Shared-Memory Parallel Maximal Biclique Enumeration

Apurba Das*
National University of Singapore
apurba@comp.nus.edu.sg

Srikanta Tirthapura
Iowa State University
snt@iastate.edu

***Abstract*—We present shared memory parallel algorithms for maximal biclique enumeration (MBE), the task of enumerating all complete dense subgraphs (maximal bicliques) from a bipartite graph, which is widely used in the analysis of social, biological, and transactional networks. Since MBE is computationally expensive, it is necessary to use parallel computing to scale to large graphs. Our parallel algorithm `ParMBE` efficiently uses the power of multiple cores that share memory. From a theoretical view, `ParMBE` is work-efficient with respect to a state-of-the-art sequential algorithm. Our experimental evaluation shows that `ParMBE` scales well up to 64 cores, and is significantly faster than current parallel algorithms. Since `ParMBE` was yielding a super-linear speedup compared to the sequential algorithm on which it was based (`MineLMBC`), we develop an improved sequential algorithm `FMBE`, through "sequentializing" `ParMBE`.**

I. Introduction

We study the problem of Maximal Biclique Enumeration (MBE) from a bipartite graph, which requires to enumerate all maximal bicliques (complete bipartite graphs). A biclique $B = (B_L, B_R)$, $B_L \subseteq L$, $B_R \subseteq R$ is a dense bipartite subgraph of the original bipartite graph $G = (L, R, E)$ where every vertex in B_L is connected to every vertex in B_R. MBE is a computationally hard problem since the number of maximal bicliques can be of exponential order [1]. However, the number of maximal bicliques in real world graphs is typically small and therefore we can hope for enumerating them all in reasonable amount of time. Sequential algorithm for solving the MBE problem has been studied for more than a decade. Eppstein [2] proposed a linear time sequential algorithm for enumerating all maximal biclique from a simple undirected graph with bounded arboricity using a technique called *acyclic orientation*. Alexe et al. [3] proposed an output sensitive algorithm based on consensus technique where large bicliques are constructed by combining small bicliques starting with stars. There are many other works on designing sequential algorithms for solving MBE on static graph [4] [5] [6] [7] [8]. Liu et al. [9] develop a output sensitive branch and bound algorithm `MineLMBC` for solving the same problem which is also efficient in practice compared to the other algorithms.

The runtime of sequential algorithms for solving MBE can be high on large graphs. For example, `MineLMBC` takes approximately 11 hours to enumerate 5.2 million maximal bicliques in a bipartite **`IMDB`** network with 1.2 million vertices and 3.8 million edges and more than 8 hours to enumerate around 54 million maximal bicliques from **`BookCrossing`** with approximately 445 thousand vertices and 1.1 million edges. Clearly, sequential algorithms are not suitable for enumerating large number of maximal bicliques and this motivates us in designing parallel algorithms.

In this work we develop shared memory parallel algorithms for MBE. We choose shared memory parallelism because (1) the graph can reside on a single shared global memory, thus no need to distribute it across the nodes and (2) there is no network communication overhead.

In this work we make the following contributions:

Theoretically Efficient Parallel Algorithm `ParLMBC`: We present a shared-memory parallel algorithm `ParLMBC` that takes as input a bipartite graph G and enumerates all maximal bicliques in G. `ParLMBC` is a parallelization of a state-of-the-art sequential algorithm `MineLMBC`, due to Liu, Sim, and Li [9]. Our analysis of `ParLMBC` using a work-depth model of computation [10] shows that it is work-efficient and has a low parallel depth.

Faster Parallel Algorithm `ParMBE`: We design a practically efficient shared memory parallel algorithm `ParMBE` that builds on `ParLMBC` and yields substantially improved practical performance. The high level idea is to create cluster (subproblem) for each vertex v with its 2-neighborhood vertices and run `ParLMBC` as a subroutine for enumerating all the maximal bicliques from each cluster in parallel. This approach significantly reduces the parallel enumeration time compared with `ParLMBC` because the computational cost for enumerating the bicliques is directly related to the size of the candidate set (for the exploration of the search space) and its adjacent neighborhood which is much smaller in each subproblem in `ParMBE` than that of executing `ParLMBC` directly on the input graph. Thus, the size of the problem instances is reduced in `ParMBE` while keeping the total number of recursive calls same as that of `ParLMBC` as each recursive call is followed by the generation of a maximal biclique. If we simply enumerate all maximal bicliques from each subproblem then a maximal biclique will be enumerated more than one. We prevent this by assuming an ordering of the vertices using a `rank` function so that a highly ranked vertex will contain more maximal biclique than a low ranked vertex. Clearly, there will be imbalance of the load if we enumerate the maximal bicliques from the subproblems corresponding to the highly ranked vertices.

* This work was done while the author was at Iowa State University

2640-0316/19/$31.00 ©2019 IEEE
DOI 10.1109/HiPC.2019.00016

We distribute the load by delegating the task of enumerating maximal bicliques to the lower ranked vertices adjacent to a highly ranked vertex. For doing this, we create subproblems in a manner that all maximal bicliques enumerated from the subproblem for vertex w will have w as the least ranked vertex. However, computing the exact rank of the vertices beforehand is difficult and therefore we heuristically consider degree of a vertex for computing the rank. This way, in our optimized algorithm `ParMBE` we ensure (1) non duplicate enumeration of all maximal bicliques and (2) load distribution.

Experimental Evaluation: We empirically evaluate all our algorithms and the experiment shows that `ParLMBC` yields **3x-19x** parallel speedup and `ParMBE` yields **21x-345x** parallel speedup when compared with `MineLMBC`, the state-of-the-art sequential algorithm for MBE on a multicore machine with 64 cores in it. We also show that the parallel speedup of `ParMBE` is almost a linear function of the number of processors - the speedup increases with the increase in the number of the processor cores. Next we implement the state of the art MapReduce algorithm `CDFS` in a shared memory setting that we call `MCoreCDFS` and show that it gives magnitude of order speedup over the sequential algorithm `MineLMBC`. We also show that our parallel algorithm `ParMBE` is upto **3x** faster than `MCoreCDFS`.

Efficient Sequential Algorithm: The super-linear speedup of `ParMBE` over sequential `MineLMBC` (sometimes 345x on 64 cores) shows that it must be possible to develop a better sequential algorithm than `MineLMBC`. Leveraging this observation, we present an efficient sequential algorithm `FMBE` through "sequentializing" `ParMBE`, i.e. executing the steps of `ParMBE` sequentially. `FMBE` is simple to implement, often significantly faster than `MineLMBC`, and always at least matches the performance of `MineLMBC`.

Roadmap: The rest of the paper is organized as follows. We present the preliminaries and backgrounds in Section III followed by the description of our parallel algorithms in Section IV. We present experimental evaluations of all our parallel algorithms in Section V and we conclude in Section VI.

II. Prior and Related Works

Parallel Algorithms: Previous works on parallel algorithms for MBE consider both the shared memory setting [11] and distributed memory setting [12]. The shared memory algorithm [11] does not consider load balancing across threads and also has not been evaluated on large graphs – the largest graph considered there is with 500 vertices and 9K edges. There is a parallel algorithm in the Map-Reduce [12] that does scale to large graphs. When compared with this work, our algorithm employs a greater degree of parallelism. We compare with an adaptation of the [12] algorithm to the shared-memory model in our experimental section.

Sequential Algorithms: Alexe et al. [3] present an algorithm for MBE from a static graph based on the "consensus method", whose time complexity is proportional to the size of the output (number of maximal bicliques in the graph) - termed as an *output-sensitive algorithm*. Damaschke [13] present an algorithm for MBE from bipartite graphs with a skewed degree distribution. Gély et al. [14] present an algorithm for MBE through a reduction to maximal clique enumeration (MCE). However, in their work, the number of edges in the graph used for enumeration increases significantly compared to the original graph. Makino and Uno [5] present an algorithm based on matrix multiplication, which provides the current best theoretical time complexity for dense graphs. Eppstein [2] presented a linear time algorithm for MBE when the input graph has an arboricity that is bounded by a constant. Other works on sequential algorithms for MBE on a static graph include [15], [16], [17], [18]. Li et al. [6] show a correspondence between closed itemsets in a transactional database and maximal bicliques in an appropriately defined graph. Das et al. [19] present an algorithm for MBE from a dynamic graph that is changing due to the addition/deletion of edges. Practically, the most efficient sequential algorithm for MBE from a static graph seems to be due to Liu et al. [9], based on depth-first-search.

In a prior work [20], we presented shared memory parallel algorithms for maximal clique enumeration (MCE) from an undirected graph. There are some significant differences between the problems of MCE and MBE. A maximal clique lies within the 1-hop neighborhood of each vertex in the clique, where as a maximal biclique does not lie within the 1-neighborhood of a vertex. We have to consider the 2-hop neighborhood of vertices to reach all vertices within a maximal biclique. This makes the subproblems generated during MBE larger than the ones during MCE. The sequential algorithms used in MCE are also different from the ones used for MBE and so are the pruning and exploration methods. Other works on parallel MCE include distributed algorithm due to Xu et al. [21], and MapReduce algorithm due to Svendsen et al. [22].

III. Preliminaries

We consider simple undirected bipartite graph $G = (L, R, E)$ where L and R are two partitions and $E \subseteq L \times R$. The set of vertices adjacent to a vertex v is denoted by $\Gamma(v)$ and the set of vertices common to all the vertices in the set X is denoted by $\Gamma(X)$. Mathematically, $\Gamma(v) = \{u|(u, v) \in E\}$ and $\Gamma(X) = \{u|\forall x \in X, (u, x) \in E\}$. We denote by $\Gamma_2(v)$ all the vertices reachable in 2 hops from v and by $\deg(v)$ the number of vertices adjacent to v. Let d denote the maximum degree of the graph G, M denote the number of maximal biclique in G, and M_v denote the number of maximal bicliques in G containing a particular vertex v.

Sequential Algorithm `MineLMBC`: The algorithm `MineLMBC` enumerates all maximal bicliques of a simple undirected graph G by exploring the graph in a depth-first manner. Each node in the search tree generates a maximal biclique and spawns child nodes by adding the vertices to the current set X (which is a partition of the current maximal biclique) one at a time from the set $\texttt{tail}(X)$ which is a set of candidate vertices

for generating maximal bicliques from X often called tail vertices of X. For generating all maximal bicliques when G is a bipartite graph, $\texttt{tail}(X)$ is initialized with the smaller partition, $\Gamma(X)$ with the larger partition, X with an empty set, and minimum size $ms = 1$. MineLMBC is formally described in Algorithm 1.

Algorithm 1: $\texttt{MineLMBC}(X, \Gamma(X), \texttt{tail}(X), ms)$

Input: X - vertex set, $\Gamma(X)$ - adjacency list of X
$\texttt{tail}(X)$ - tail vertices of X
ms - minimum size threshold.
Output: B - Set of all maximal bicliques containing X.

```
1  for v ∈ tail(X) do
2      if (|Γ(X ∪ {v})| < ms) then
3          tail(X) ← tail(X) \ {v}
4  if |X| + |tail(X)| < ms then
5      return
6  sort vertices of tail(X) into ascending order of
     |Γ(X ∪ {v})|
7  for v ∈ tail(X) do
8      tail(X) ← tail(X) \ {v}
9      if |X ∪ {v}| + |tail(X)| > ms then
10         Y ← Γ(Γ(X ∪ {v}))
11         if Y \ (X ∪ {v}) ⊆ tail(X) then
12             if |Y| ≥ ms then
13                 B ← B∪ < Y, Γ(X ∪ {v}) >
14             MineLMBC(Y, Γ(X ∪ {v}), tail(X) \ Y, ms)
```

The time complexity of generating all maximal bicliques in a bipartite graph G using MineLMBC is $O(ndM)$ where n is the size of the smaller partition of G and other notations carry their usual meaning.

Parallel Cost Model: We analyze our shared memory parallel MBE algorithm assuming CRCW PRAM model [10], a model of shared memory parallel computation that assumes concurrent read and concurrent writes. Our parallel algorithm is also suitable for other parallel computation model such as EREW PRAM (Exclusive Read Exclusive Write) at a cost of logarithmic factor increase in both the work and the parallel depth. For measuring the efficiency of our parallel algorithm, we use *work-depth* model [10] where the "work" of a parallel algorithm is the cumulative cost of all the operations and "depth" is the length of the longest chain of dependent computations also denoted as the *parallel time* or *span*.

We assume parallel insertions and finding of elements using a concurrent hashtable using the following result. We use this result in showing the work and depth of the parallel operations in our analysis of the parallel algorithms.

Theorem 1 (Theorem 3.15 [23])**.** *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.*

IV. Parallel MBE Algorithm

In this section, we design two shared memory parallel algorithms for MBE. The first algorithm ParLMBC is inspired by the state-of-the-art output sensitive algorithm MineLMBC and the second algorithm ParMBE is inspired by the state-of-the-art distributed algorithm CDFS and our parallel algorithm ParLMBC. Although ParLMBC is a theoretically work-efficient parallel algorithm, algorithm ParMBE is practically much faster than ParLMBC because it subdivide the problem into multiple tasks per vertex and reduces the overall time by significantly reducing the size of the tail set per task basis as opposed to the larger tail set in the parallel recursive calls in ParLMBC. We will discuss about ParMBE followed by the discussion on ParLMBC which is the subroutine for enumerating maximal bicliques in tasks per vertex in the algorithm ParMBE.

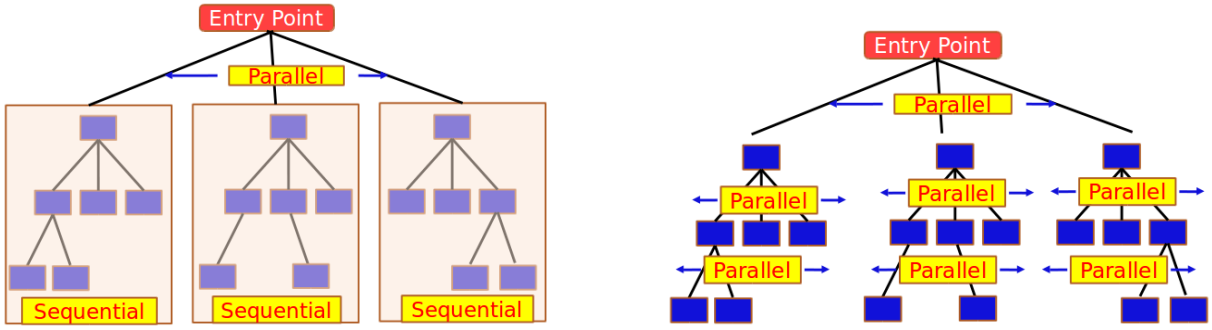
A. Algorithm ParLMBC

Let us first discuss about the high level idea in designing the parallel algorithm ParLMBC before going into the technical details. The parallel design is based on introducing parallelism at the recursive call levels so that branching out from a recursive call (usual scenario in a recursive backtracking algorithm) can be performed in parallel. This way, many recursive calls at each level can be processed by individual threads simultaneously (see Figure 1). However, it is not always straightforward to call recursive procedures in parallel (that are called iteratively in sequential procedure) due to a sequential barrier caused by the iterative dependence of data structures used by the recursive calls. In that case, in parallel design, we modify the update process of the data structure in each iteration such that the state of the data structures in each iteration used by the recursive calls remains independent. This allows us to call the recursive procedures in parallel.

ParLMBC consists of three main components: (1) pruning of the tail set (Lines 1-3), (2) sorting the vertices in the tail set (Line 6), and (3) recursive exploration of the search space in depth-first order (Lines 7-14) where each iteration corresponds to the exploration of a sub search space. Now we explain how we parallelize each of these components:

Parallel pruning of the tail set: Within a single call to MineLMBC, pruning on the tail vertex set is performed in parallel in a straight forward manner: iterate over the vertices in the tail set in parallel and remove those that fails to satisfy the threshold size criteria as in Line 2 of Algorithm 1. The total work of this step is $O(nd)$ following the analysis in the sequential algorithm description and the depth is $O(1)$ following the $O(1)$ cost of inserting an element and $O(1)$ cost of searching an element following Theorem 1.

Parallel sorting the vertices in the tail set: Sorting the vertices in a set using a parallel sorting algorithm is difficult to achieve because parallel sorting algorithm works on list



(a) parallelism in prior work [12]

(b) parallelism in this work

Fig. 1: demonstration of addition of more parallelism in search tree exploration in this work.

data structure such as array or vector where the elements are indexed in the data structure which is not the case when the elements are in an unordered set such as the vertices in the `tail` set in our situation. We overcome this difficulty by putting the elements of `tail` set in an array (assume the array is A) in parallel with identity mapping meaning that $A[v] = v$ only if v is in `tail` set and $A[v] = 0$ otherwise. Next we apply parallel filter operation on the array to compact it (an array containing only the elements in the pruned `tail` set) and assume that the resulting array is A'. Next we apply parallel sorting algorithm to sort the elements in A' with comparison on $\Gamma(\cdot)$ instead of the absolute values of the vertices in the tail set. Finally we generate the sorted array consisting of the vertices in pruned `tail` set. The total work of this step is $O(n \log n)$ which is a combination of array A' construction step with total work $O(n)$ and sorting step with total work $O(n \log n)$ and the depth is $O(\log n)$ which consists of the $O(\log n)$ depth for the construction of A' and $O(\log n)$ depth for parallel sorting.

Parallel unrolling the iterative recursion: We first observe that there is a sequential dependency in the iterations because of the update process of the `tail` vertex set as in Line 8 of Algorithm 1. Therefore, it is not straightforward to execute on each vertex (at Line 7 of Algorithm 1) in parallel. Also, we observed that we can run the iterations in parallel if the dependency of `tail` set can be removed. We exactly do this by creating a local `tail` set for each iteration and initializing it with the vertices from index $i+1$ till κ where κ is the size of `tail` set before the iterations begin and i is the current iteration number if presented sequentially. The total work of this step is $O(n + d^2)$ which is a combination of (1) updating the `tail` set with total work $O(n)$, (2) constructing the Y with total work $O(d^2)$, and (2) subset check as in Line 11 of Algorithm 1 with total work $O(n)$. The depth of this step is $O(d^2)$ which is a combination of the depths of these 3 aforementioned components: $O(1)$ for updating `tail` set, $O(d^2)$ for computing Y, and $O(1)$ for subset check.

Along with the parallelization of the main steps of the enumeration algorithm, once one partition of a new maximal biclique is settled, generation of the other partition of the same maximal biclique is a costly operation because it involves computation of intersection of unordered sets. Note that the generation of the other partition is required for ensuring non duplicate generation of the maximal bicliques. Our approach for parallelizing the intersection computation is the following:

Parallel intersection computation of unordered sets: We compute $\Gamma(\Gamma(X \cup \{v\}))$ by iterating on each vertex in $\Gamma(X \cup \{v\})$ in parallel. For each vertex u adjacent to some vertex in $\Gamma(X \cup \{v\})$ we use an `atomic` counter and increment by one for each vertex w in $\Gamma(X \cup \{v\})$ adjacent to u. For doing this, we use a hashmap with vertices as `key` and its counter as the `value`. Finally, we consider those vertices from the map in the set Y with counter value $|\Gamma(X \cup \{v\})|$. More details are in Section V.

The formal description of the parallel techniques is presented in Algorithm 2 and we present the work and depth analysis of `ParLMBC` in the following theorem:

Theorem 2. *Given a bipartite graph $G = (L, R, E)$ with $n = |L| \leq |R|$, the number of maximal bicliques M, and the maximum degree d, the total work of* `ParLMBC` *is $O(ndM)$ and the depth of the algorithm is $O(d(d^2 + \log n))$.*

Proof. From the discussion on the parallel steps, it is easy to see that the total work of `ParLMBC` is $O(ndM)$.

To show the depth of the algorithm, note that the overall depth is the depth of a single recursive call multiplied by the depth of the search tree. From the previous discussion of the depth of the individual parallel steps, it is clear to see that the depth of a single recursive call is $O(d^2 + \log n)$. Now the depth of the search tree is the maximum degree of the graph d. This is because, the size of X increases by at least 1 when the depth of the search tree is increased by 1 (Line 11 of Algorithm 2). For contradiction assume that the depth of the search tree is $d + 1$. Then there will be an X at depth $d + 1$ with at least $d + 1$ vertices. This is a contradiction because the maximum degree of the graph d. Thus, the depth of the algorithm follows. □

Algorithm 2: ParLMBC$(X, \Gamma(X), \texttt{tail}(X), ms)$

Input: X - vertex set, $\Gamma(X)$ - adjacency list of X $\texttt{tail}(X)$ - tail vertices of X, ms - minimum size threshold.
Output: B - Set of all maximal bicliques containing X.

1 **for** $v \in \texttt{tail}(X)$ **do in parallel**
2 **if** $(|\Gamma(X \cup \{v\})| < ms)$ **then**
3 $\texttt{tail}(X) \leftarrow \texttt{tail}(X) \setminus \{v\}$
4 **if** $|X| + |\texttt{tail}(X)| < ms$ **then**
5 **return**
6 **parallel** sort vertices of $\texttt{tail}(X)$ into ascending order of $|\Gamma(X \cup \{v\})|$
7 Let the elements of sorted $\texttt{tail}(X)$ are presented in the order $0..\kappa$
8 **for** $i \in [0..\kappa]$ **do in parallel**
9 $\texttt{ntail}(X) \leftarrow tail[i+1..\kappa]$
10 **if** $|X \cup \{v\}| + |\texttt{ntail}(X)| > ms$ **then**
11 $Y \leftarrow \Gamma(\Gamma(X \cup \{v\}))$ in **parallel**
12 **if** $Y \setminus (X \cup \{v\}) \subseteq \texttt{ntail}(X)$ **then**
13 **if** $|Y| \geq ms$ **then**
14 $B \leftarrow B \cup < Y, \Gamma(X \cup \{v\}) >$
15 ParLMBC$(Y, \Gamma(X \cup \{v\}), \texttt{ntail}(X) \setminus Y, ms)$

B. Algorithm ParMBE

Note that the work efficiency of ParLMBC comes at a cost of additional sequential work of generation of tail set for each parallel recursive call (Line 9 of Algorithm 2) that remains hidden under the cost of updating the tail set to prune the search space (Line 1-3 of Algorithm 2) because of asymptotically higher work complexity of this pruning step. An approach to reduce this sequential overhead is to reduce the size of the candidate set. One way of reducing the candidate set size (tail set in this algorithm) is to create subproblem for each vertex and only consider the vertices in the neighborhood (or 2-neighborhood as needed) in creating the candidate set. We take this approach and design another algorithm ParMBE with techniques close to the distributed algorithm CDFS. ParMBE works in the following way: For each vertex $v \in V(G)$, we create a subgraph G_v consisting of the vertices in the set $\Gamma_2(v)$ and enumerate all maximal bicliques from G_v using our parallel algorithm ParLMBC. While working on the subproblems, it is important not to enumerate a maximal bicliques more than once. We ensure this by assuming a total ordering of the vertices and initializing the tail set for each subproblem with the vertices that comes after v in that ordering and belongs to the partition of v. We create this ordering by defining a rank function based on which we order the vertices. In this work our rank function is based on the degree of the vertices where for any two vertices u and v, $\texttt{rank}(u) > \texttt{rank}(v)$ if $\deg(u) > \deg(v)$ and when $\deg(u) = \deg(v)$, $\texttt{rank}(u) > \texttt{rank}(v)$ if the absolute value of u is greater than the absolute value of v. ParMBE is presented formally in Algorithm 3

Algorithm 3: ParMBE(G, ms)

Input: $G = (L, R, E)$ - input graph, ms - minimum size threshold.
Output: B - Set of all maximal bicliques containing X.

1 **for** $v \in L$ **do in parallel**
2 $X \leftarrow \{v\}$
3 $\Gamma(X) \leftarrow \Gamma(v)$
4 $\texttt{tail}(X) \leftarrow \emptyset$
5 **for** $w \in \Gamma(v)$ **do in parallel**
6 **for** $y \in \Gamma(w)$ **do in parallel**
7 **if** $\texttt{rank}(y) > \texttt{rank}(v)$ **then**
8 $\texttt{tail}(X) \leftarrow \texttt{tail}(X) \cup \{y\}$
9 ParLMBC$(X, \Gamma(X), \texttt{tail}(X), ms)$

Discussion: Note that the high level idea of ParMBE has a close resemblance with our recent work on parallel algorithm for maximal clique enumeration [20]. But unlike maximal cliques, only the vertices adjacent to the candidate set are not sufficient for the exploration of the search space. For maximal bicliques, we need to focus on the 2-neighborhood of the vertices in the candidate set. Also, lowest rank among the vertices in one partition does not necessarily imply lowest rank in the entire vertex set of the maximal biclique which is unlike the case of maximal cliques. Therefore, we need to take special care while creating the candidate sets for maximal biclique enumeration to ensure that all maximal bicliques are enumerated without any duplication. We ensure both of these through a careful design in ParMBE.

Next, we can use our previous work [20] here for parallel maximal biclique enumeration. However that is inefficient from the space cost perspective. If we want to apply parallel maximal clique enumeration algorithm for enumerating all maximal bicliques, we need to transform the original bipartite graph where we need to add many edges to the original graph such that each partition becomes a clique in the transformed graph. It is then easy to see that a maximal clique in the transformed graph corresponds to a maximal bicliques in the original bipartite graph.

V. Experiments

We empirically evaluate our parallel algorithms ParLMBC and ParMBE and compare with the state-of-the-art sequential and parallel algorithms MineLMBC and CDFS respectively on real world bipartite networks to show the parallel speedup and scalability of our parallel algorithms. We show that our parallel algorithms show significant speedup over the state of the art sequential and parallel algorithms. We evaluate all the experiments in a multicore computer equipped with 3TB RAM and 64 core processor (four 16-core Intel 6130 processors).

A. Datasets

We use 6 real world static bipartite networks from publicly available repository KONECT [24] for the experiments. The summary of the dataset is presented in Table I. We consider two networks **DBpedia locations** and **Marvel** with small number of maximal bicliques for comparison with a prior sequential algorithm iMBEA.

TABLE I: Bipartite Networks used for evaluation, and their properties.

Dataset	#Vertices	#Edges	#Maximal Bicliques
DBpedia locations	225,486	293,697	75,360
Marvel	19,428	96,662	206,135
YouTube	124,325	293,360	1,826,587
IMDB	1,199,919	3,782,463	5,160,061
Stack Overflow	641,873	1,301,942	3,320,824
BookCrossing	445,801	1,149,739	54,458,953

B. Implementation of the algorithms

In the implementations of ParLMBC and ParMBE, we use parallel_for and parallel_for_each constructs from Intel TBB parallel library [25] for the implementation of the parallel for loop. For the atomic operations on hashtable we use concurrent_hash_map, for the atomic operations on unordered set we use concurrent_unordered_set, and for atomic operations on the dynamic array we use concurrent_vector. We implement $\Gamma(\Gamma(X \cup \{v\}))$ in parallel in ParLMBC. For doing this, we use a concurrent_hash_map from vertex as the key and the the number of vertices in the set $\Gamma(X \cup \{v\})$ it is adjacent to as the value. Then we iterate on the vertices of $\Gamma(X \cup \{v\})$ in parallel and update the frequency of the vertices adjacent to each vertex in $\Gamma(X \cup \{v\})$. Finally, we generate the set Y with the vertices in the concurrent_hash_map whose frequency is $|\Gamma(X \cup \{v\})|$. For the parallel sort we use parallel_sort. All of these are provided by TBB. We use C++11 for the implementation of the algorithms and compile the sources using Intel ICC compiler version 18.0.3 with optimization level '-O3'. System level load balancing is performed using a dynamic work stealing scheduler [25] built inside TBB.

For the comparison with prior works, we implement state of the art MapReduce algorithm CDFS in shared memory setting that we call MCoreCDFS. We also implement a more recent sequential algorithm iMBEA [8].

C. Discussion of the Results

Now we present and interpret the results of the empirical evaluations of our parallel algorithms. First we will show the parallel speedup (with respect to MineLMBC) and scalability of ParLMBC and ParMBE to show that the performance of the parallel algorithms improve when the number of core is increased. Next we compare our parallel algorithms with MCoreCDFS to show that ParMBE performs better than MCoreCDFS on all the input graph and then we compare with the sequential algorithm iMBEA and show that we get magnitude of order speedup over this algorithm. This is as expected because, the performance of iMBEA is much worse than the performance of MineLMBC.

Parallel Speedup: We show the parallel speedup of ParLMBC and ParMBE in Table II. The result clearly shows the substantially better performance of ParMBE over ParLMBC and magnitude of order parallel speedup of ParMBE compared with MineLMBC. However, more than $64\times$ speedup of ParMBE in a 64 core machine clearly indicates that the sequential algorithm MineLMBC is not the most efficient one.

TABLE II: Runtime (in sec.) of MineLMBC, ParLMBC, and ParMBE on 64 cores. Numbers in the parenthesis indicates the parallel speedup.

Dataset	MineLMBC	ParLMBC	ParMBE
YouTube	305	85 (**3.6x**)	8.3 (**36.7x**)
IMDB	41476	2187 (**19x**)	120 (**345.6x**)
Stack Overflow	23323	1220 (**19x**)	892 (**26x**)
BookCrossing	18569	4259 (**4.3x**)	863 (**21.5x**)

Scalability: We show the scalability of our parallel algorithms ParLMBC and ParMBE in Figure 3. The result shows that ParMBE scale up almost linearly as we increase the degree of parallelism by increasing the number of threads. The x axis is the number of the threads used and the y axis is the parallel speedup which is a function of the number of threads. We also see that ParLMBC does not scale as we increase the number of threads. This is because the additional overhead in the parallelization of processing the candidate sets is large compared to the sequential algorithm MineLMBC due to the large candidate sets in the recursive calls compared with the candidate sets in the recursive calls in ParMBE. Moreover, the speedup achieved with 64 threads is not always maximum (for example Figure 3(a)) especially when the problem size is small.

Comparison with prior works: We compare our parallel algorithms ParLMBC and ParMBE with a distributed parallel algorithm CDFS [12] in a shared memory setting by reusing the methods of cluster construction and maximal biclique enumeration and eliminating the need to communicate the subgraphs by storing a single copy of the graph in a global shared memory. From Table III we see that ParMBE is upto **3x** faster than MCoreCDFS. This speedup is due to the use of parallel algorithm ParLMBC in ParMBE for enumerating the maximal biclique from the subproblems instead of MineLMBC as in CDFS. We also evaluate a prior sequential algorithm iMBEA and it appears that MineLMBC is significantly faster over iMBEA. For an instance, on **Marvel** graph, MineLMBC takes around 10 second to enumerate around 206K maximal bicliques where as iMBEA takes more than 30 minutes to enumerate those maximal bicliques. In another example on **DBpedia locations** graph, MineLMBC takes around 450 sec. to enumerate around 75K maximal bicliques where as iMBEA takes more than an hour for exactly doing the same job.

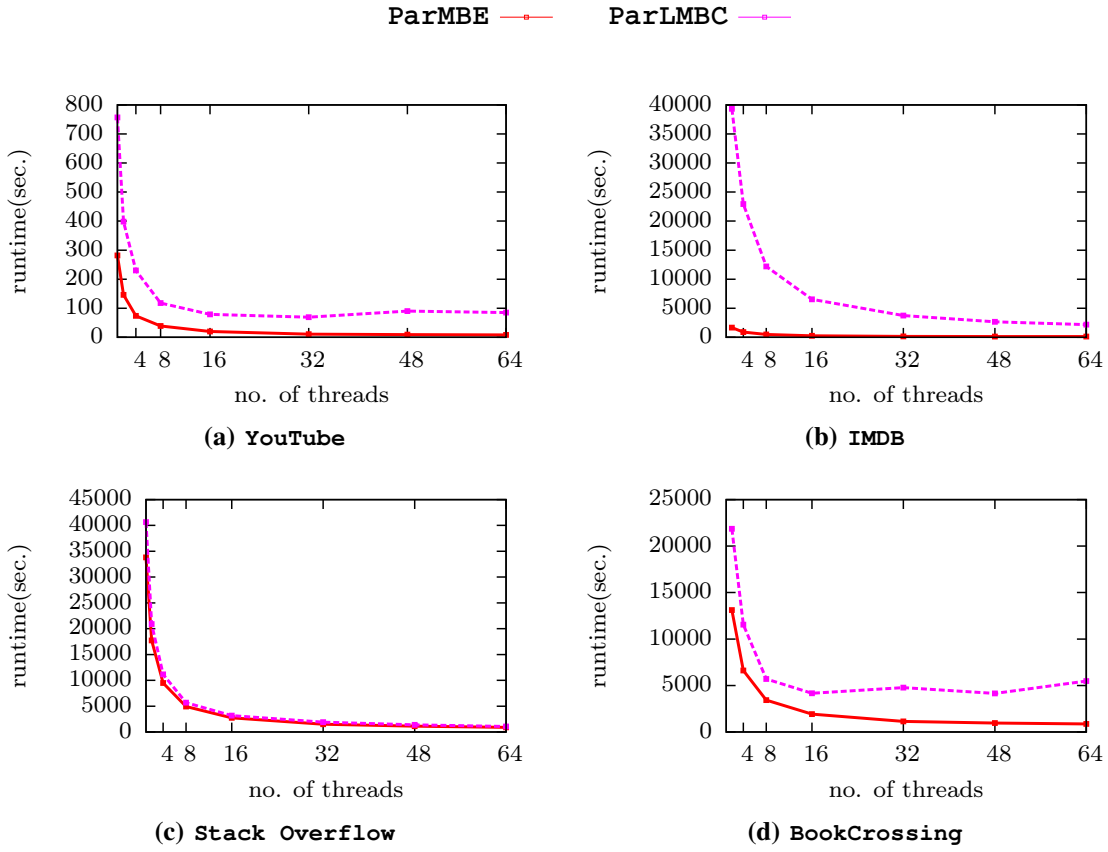


Fig. 2: Runtime of ParMBE and ParLMBC as a function of number of threads.

TABLE III: Comparison of runtime (in sec.) of ParMBE with CDFS on 64 cores.

Dataset	MCoreCDFS	ParMBE
YouTube	18	**8.3**
IMDB	247	**120**
Stack Overflow	2694	**892**
BookCrossing	2190	**863**

D. New Sequential Algorithm FMBE

Based on the observation from Table II the parallel speedup of ParMBE compared with MineLMBC is magnitude of order better than the number of cores (64) in the multicore machine where we execute all the parallel algorithms. Clearly MineLMBC is not the optimized sequential algorithm and it indicates a gap between the possibility of a better sequential algorithm and the algorithm MineLMBC. We fill the gap by designing a new sequential algorithm FMBE where we execute all the procedures in ParMBE sequentially. Surprisingly, we find that the time complexity of FMBE is better than the time complexity of MineLMBC in both the theory and in practice. We formally present FMBE in Algorithm 4.

Typically the size of the tail set for each subproblem becomes significantly smaller than the size of the tail set for the entire graph. Intuitively the significant reduction in the runtime is related to the reduction in the size of the tail set in the argument of MineLMBC in algorithm FMBE. The following lemma shows a better time complexity of FMBE than MineLMBC when the maximum degree d of the graph is much smaller than the number of vertices n.

Lemma 1. *Given a bipartite graph $G = (L, R, E)$ with $n = |L| \leq |R|$, the number of maximal bicliques M, and the maximum degree d, the time complexity of* FMBE *is $O(d^4M)$.*

Proof. First we show that if $b = (b_L, b_R)$ is a maximal biclique of G where $b_L \subseteq L$ and $b_R \subseteq R$, it will be enumerated from G_v only where v is the least ranked vertex among all the vertices in b_L. Suppose this is not the case, and assume that b is enumerated from another subgraph G_w for some $w \in L$. Then, v is not the least ranked vertex among the vertices of b_L based on the construction of G_w. This is a contradiction.

Now for each vertex $v \in L$, the the size of G_v is $O(d^2)$ because, in constructing G_v, we consider the vertex v, the vertices in $\Gamma(v)$ and the vertices adjacent to each of the vertex in $\Gamma(v)$. The time complexity of MineLMBC on the instance of G_v is $O(d^3M_v)$ where M_v is the number of maximal bicliques in G_v and d is the maximum degree of G_v which is same as the maximum degree of the original graph G. Thus the overall time complexity is $\sum_{v \in L} O(d^3M_v)$ which is $O(d^4M)$ because

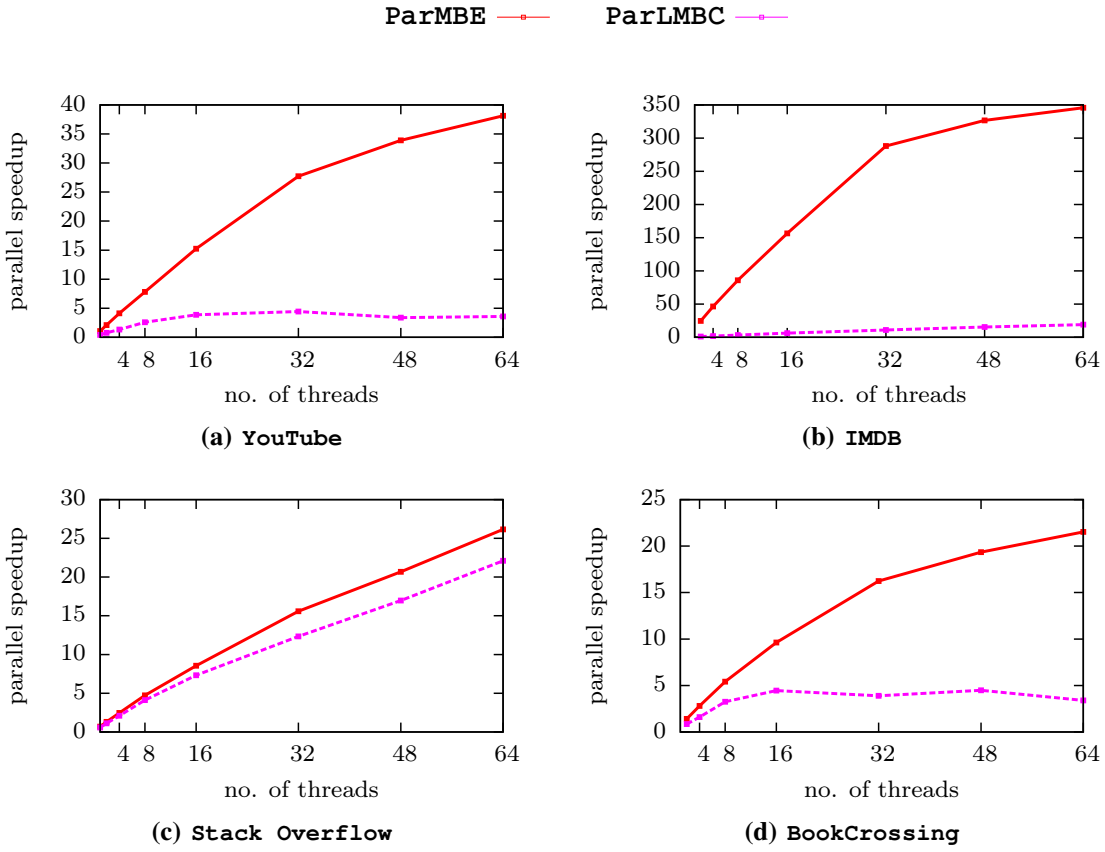


Fig. 3: Parallel speedup (with respect to MineLMBC) of ParMBE and ParLMBC as a function of number of threads.

each maximal cliques will be enumerated at most d times by d different subproblems. This completes the proof. □

In Figure 4 we show that the runtime of FMBE is significantly smaller than the runtime of MineLMBC in almost all the input graphs. This shows that FMBE is a significantly better sequential algorithm than MineLMBC.

Algorithm 4: FMBE(G)

Input: $G = (L, R, E)$ - input graph
Output: B - Set of all maximal bicliques containing X.

```
1 for v ∈ L do
2     X ← {v}
3     Γ(X) ← Γ(v)
4     tail(X) ← ∅
5     for w ∈ Γ(v) do
6         for y ∈ Γ(w) do
7             if rank(y) > rank(v) then
8                 tail(X) ← tail(X) ∪ {y}
9     MineLMBC(X, Γ(X), tail(X), 1)
```

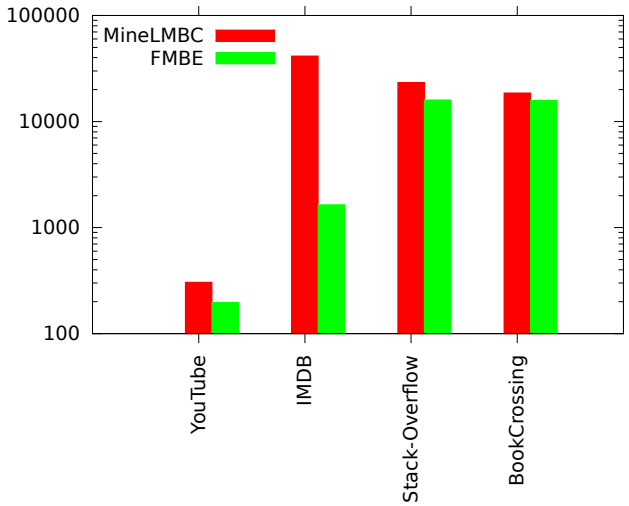


Fig. 4: Comparison of runtimes (in sec.) of MineLMBC and FMBE with the input graphs in the x-axis and the runtime in the y-axis in logscale.

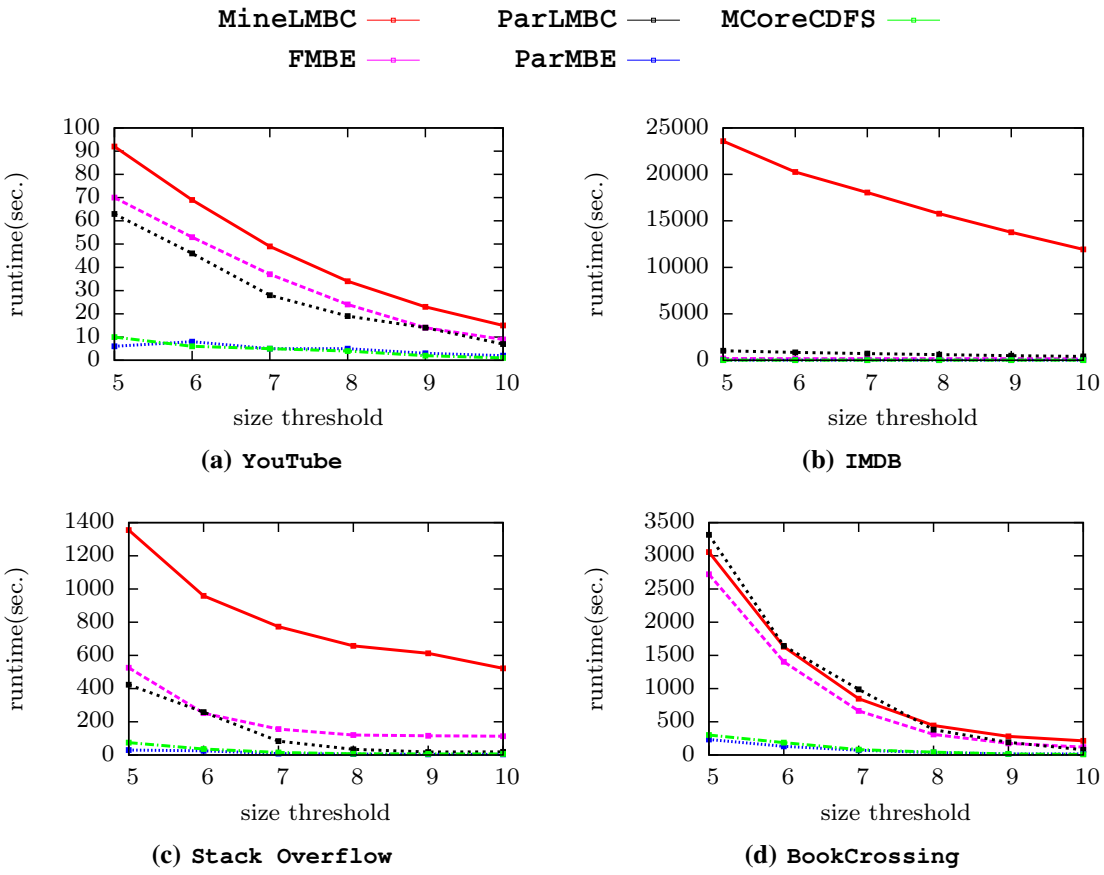


Fig. 5: Runtime as a function of the size threshold.

E. Enumeration of Large Maximal Bicliques

We also consider enumerating maximal bicliques with size of each partition exceeding a threshold ms. This is important when we do not require all maximal bicliques but some of them with larger size. Note that enumerating all maximal bicliques is a special case when $ms = 1$. We try to address the following question in this study: Can we enumerate large maximal bicliques faster than enumerating all maximal bicliques? We conduct experimental evaluation to show in Figure 5 that indeed the runtime decreases with the increase in the threshold size. This clearly follows the intuition that the search space is shrunk when we increase the threshold size by disregarding the vertices with smaller degrees (than the threshold) from being in the candidate set for the enumeration process. Also, it is clear from the plots that the relative time of the algorithms is preserved. Intuitively, this ensures the shrinking of search space in each of the algorithms due to the increase in the threshold size.

VI. Conclusion

In this work we design shared memory parallel algorithm for maximal biclique enumeration (MBE) based on the state-of-the-art sequential algorithm `MineLMBC`. We provide theoretical guarantee of work-efficiency and low depth of our parallel algorithms and demonstrate through experimental evaluation that our parallel algorithms are efficient and scalable - scales linearly with the increase in the number of processor. We also so that our practical efficient parallel algorithm `ParMBE` provides more than **2x** speedup over the shared memory implementation of state-of-the-art MapReduce based parallel algorithm `CDFS`. In addition to that, we develop with theoretical guarantee a faster (than `MineLMBC`) sequential algorithm inspired by the observation that the speedup achieved by `ParMBE` is much higher than the maximum number of available cores in the system.

Acknowledgment: This work is supported in part by the US National Science Foundation through grants 1527541, 1725702, and 1632116.

References

[1] E. Prisner, "Bicliques in graphs i: Bounds on their number," *Combinatorica*, vol. 20, no. 1, pp. 109–117, 2000.

[2] D. Eppstein, "Arboricity and bipartite subgraph listing algorithms," *Information processing letters*, vol. 51, no. 4, pp. 207–211, 1994.

[3] G. Alexe, S. Alexe, Y. Crama, S. Foldes, P. L. Hammer, and B. Simeone, "Consensus algorithms for the generation of all maximal bicliques," *Discrete Applied Mathematics*, vol. 145, no. 1, pp. 11–21, 2004.

[4] M. J. Zaki and C.-J. Hsiao, "Charm: An efficient algorithm for closed itemset mining," in *Proceedings of the 2002 SIAM international conference on data mining*. SIAM, 2002, pp. 457–473.
[5] K. Makino and T. Uno, "New algorithms for enumerating all maximal cliques," in *Scandinavian Workshop on Algorithm Theory*. Springer, 2004, pp. 260–272.
[6] J. Li, H. Li, D. Soh, and L. Wong, "A correspondence between maximal complete bipartite subgraphs and closed patterns," in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2005, pp. 146–156.
[7] R. A. Mushlin, A. Kershenbaum, S. T. Gallagher, and T. R. Rebbeck, "A graph-theoretical approach for pattern discovery in epidemiological research," *IBM systems journal*, vol. 46, no. 1, pp. 135–149, 2007.
[8] Y. Zhang, C. A. Phillips, G. L. Rogers, E. J. Baker, E. J. Chesler, and M. A. Langston, "On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types," *BMC bioinformatics*, vol. 15, no. 1, p. 1, 2014.
[9] G. Liu, K. Sim, and J. Li, "Efficient mining of large maximal bicliques," in *Data warehousing and knowledge discovery*. Springer, 2006, pp. 437–448.
[10] G. E. Blelloch and B. M. Maggs, "Parallel algorithms," in *Algorithms and theory of computation handbook*, 2010, pp. 25–25.
[11] R. Nataraj and S. Selvan, "Parallel mining of large maximal bicliques using order preserving generators," *International Journal of Computing*, vol. 8, no. 3, pp. 105–113, 2014.
[12] 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.
[13] P. Damaschke, "Enumerating maximal bicliques in bipartite graphs with favorable degree sequences," *Information Processing Letters*, vol. 114, no. 6, pp. 317–321, 2014.
[14] A. Gély, L. Nourine, and B. Sadi, "Enumeration aspects of maximal cliques and bicliques," *Discrete applied mathematics*, vol. 157, no. 7, pp. 1447–1459, 2009.
[15] V. M. Dias, C. M. De Figueiredo, and J. L. Szwarcfiter, "Generating bicliques of a graph in lexicographic order," *Theoretical Computer Science*, vol. 337, no. 1, pp. 240–248, 2005.
[16] V. M. Dias, C. M. de Figueiredo, and J. L. Szwarcfiter, "On the generation of bicliques of a graph," *Discrete Applied Mathematics*, vol. 155, no. 14, pp. 1826–1832, 2007.
[17] 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.
[18] A. P. Mukherjee, P. Xu, and S. Tirthapura, "Enumeration of maximal cliques from an uncertain graph," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 3, pp. 543–555, 2017.
[19] A. Das and S. Tirthapura, "Incremental maintenance of maximal bicliques in a dynamic bipartite graph," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 231–242, 2018.
[20] A. Das, S. Sanei-Mehri, and S. Tirthapura, "Shared-memory parallel maximal clique enumeration," in *25th IEEE International Conference on High Performance Computing, HiPC 2018, Bengaluru, India, December 17-20, 2018*, 2018, pp. 62–71.
[21] Y. Xu, J. Cheng, A. W.-C. Fu, and Y. Bu, "Distributed maximal clique computation," in *IEEE BigData Congress*, 2014, pp. 160–167.
[22] M. Svendsen, A. P. Mukherjee, and S. Tirthapura, "Mining maximal cliques from a large graph using mapreduce: Tackling highly uneven subproblem sizes," *J. Parallel Distrib. Comput.*, vol. 79-80, pp. 104–114, 2015.
[23] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *Journal of the ACM (JACM)*, vol. 53, no. 3, pp. 379–405, 2006.
[24] J. Kunegis, "Konect: the koblenz network collection," in *WWW*. ACM, 2013, pp. 1343–1350.
[25] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.