Advancements in Parallel Graph Algorithms for Data Science: Scalable, Fast and Space-Efficient Solutions

Letong Wang
University of California, Riverside
Advisors: Yan Gu, Yihan Sun
lwang323@ucr.edu

ABSTRACT

Living in the world of big data, we are challenged by how to process the high volume of data efficiently. Running time and space usage are the two main challenges preventing many existing solutions from running on large real-world data. Parallel algorithms and efficient data structures are the keys to solving the challenges. With the prevalence of multi-core CPUs, designing efficient algorithms and software is now of huge practical relevance and significant theoretical interest. However, programming in a parallel setting is much harder than that in a sequential setting. We summarized four main challenges in designing efficient parallel algorithms. We explain the challenges and how we overcome them with examples of popular data science problems, such as Influence Maximization (IM) and graph traversal algorithms (Strongly Connected Component (SCC), Bi-Connectivity (BCC)). Compared with the state-of-the-art implementations on a 96-core machine with 1.5 TB main memory, our SCC and BCC are 2.7 × and 3.1 × faster than the best existing baseline on average, respectively. Our IM algorithm is the first one that can process the ClueWeb graph with 978M vertices and 75B edges in about 2 hours. The techniques we proposed in the paper are not limited to specific problems but potentially work with a large range of problems in data science.

VLDB Workshop Reference Format:
Letong Wang. Advancements in Parallel Graph Algorithms for Data Science: Scalable, Fast and Space-Efficient Solutions. VLDB 2024 Workshop: VLDB Ph.D. Workshop.

1 INTRODUCTION

Graph algorithms are fundamental to various applications, including social network analysis, bioinformatics, and geographical information systems. As the volume of data continues to grow, there is a pressing need for scalable and efficient algorithms that can handle massive graphs. Parallel computing is essential for handling the increasing size and complexity of modern datasets. The core idea is to let multiple processors work together to reduce computation time. However, programming in a parallel setting is much harder than that in a sequential setting. Four challenges that are commonly faced in parallelizing sequential algorithms are listed here:

Challenge 1: The *iterative* essence of sequential algorithms. Many nice algorithmic ideas work iteratively. How can we parallelize them if all operations logically depend on the previous?

Challenge 2: *Data structures* are more complicated. Many classic data structures only considered sequential algorithms and interfaces. How can data structures benefit parallel algorithms as they do in the sequential setting?

Challenge 3: Algorithms sacrifice *space* for parallelism. Parallel algorithms need more space to avoid data race or deal with other issues, but large space limits solvable problem size.

Challenge 4: Thread *synchronization* is expensive. Thread synchronization cost (usually not reflected in theory) is expensive, which may make parallel algorithms run even slower than sequential algorithms in some cases.

These challenges are very common in designing scalable parallel algorithms. I take three concrete problems as examples: Influence Maximization (IM), Strongly Connected Component (SCC), and Bi-Connectivity (BCC). Challenge 1 is observed in IM et al. [10, 14]. Challenge 2 is observed in IM and SCC. Challenge 3 is observed in BCC and IM. Challenge 4 is observed in SCC and BCC. My research work revolves around overcoming these challenges. *PaC-IM* [24] trades off between space usage and running time for the preprocessing step and parallelizes the *iterative* seed selection step, taking advantage of *parallel data structures.* PASGAL-SCC [11, 25] improves the performance of existing SCC on large-diameter graphs. We propose the Vertical Granularity Control (VGC) idea to reduce global *synchronizations* and use *parallel hash bags* to implement it. PASGAL-BCC [11, 12] solves the *synchronization-efficient* challenge by improving theoretical bounds, and it is efficient in practice. Note that some techniques proposed in concrete algorithms can be extended to other algorithms with similar challenges. For example, the technique for parallelizing iterative seed selection in IM can be extended to other iterative algorithms for solving optimization problems if the problem has the submodular property, and the VGC and hash bags proposed in SCC can be used in other graph traversal algorithms to improve their performance on large-diameter graphs. I will introduce *PaC-IM*, PASGAL-SCC, and PASGAL-BCC in detail in Sec. 3 to 5. Then, I will discuss future work in Sec. 6 and summarize the paper in Sec. 7.

2 PRELIMINARIES

We use the fork-join parallelism [7], and the work-span analysis [4]. We assume a set of threads that access a shared memory. A thread can `fork` two child software threads to work in parallel. When both children complete, the parent process continues. A parallel for-loop can be simulated by recursive `fork`s in logarithmic levels. The ***work*** of an algorithm is the total number of instructions, and the ***span*** is the length of the longest sequence of dependent instructions. We can execute the computation using a randomized work-stealing scheduler [2] in practice. We use an *atomic* operation

Proceedings of the VLDB Endowment. ISSN 2150-8097.

WriteMax(t, v_{new}) to write value v_{new} at the memory location t if v_{new} is larger than the current value in t. We use compare-and-swap to implement WriteMax.

3 INFLUENCE MAXIMIZATION

Influence Maximization (IM) is a crucial problem in data science. The goal is to find a fixed-size set of highly influential *seed* vertices on a network to maximize the spread of influence along the edges. For example, in viral marketing, the company may choose to send free samples to a small set of users in the hope of triggering a large cascade of further adoptions through the "word-of-mouth" effects. Given a graph $G = (V, E)$ and a stochastic *diffusion model* to specify how influence spreads along edges, we use $n = |V|$, $m = |E|$, and $\sigma(S)$ to denote the expected influence spread on G using the seed set $S \subseteq V$. The IM problem aims to find a seed set S with size k to maximize $\sigma(S)$.

Although IM is NP-hard, a greedy algorithm can achieve $(1-1/e)$-approximation [16]. Given the current seed set S, the greedy algorithm selects the next seed as the vertex with the highest *marginal gain*, where $\Delta(v \mid S) \leftarrow \sigma(S \cup \{v\}) - \sigma(S)$. However, the challenge lies in estimating the influence of a seed set $\sigma(\cdot)$.

Existing work either uses a **simulation**-based or a **memoizing**-based approach to compute σ. The simulation-based approach uses Monte-Carlo (MC) experiments by simulating R' rounds of influence diffusion [16, 17], but the solution quality relies on a high value of R' (usually around 10^4). This method is very expensive in time. To avoid MC simulations, the memoizing-based approach uses ***sketches*** by pre-storing R sampled graphs as the results of MC experiments and/or memorizing auxiliary information to accelerate influence computation, such as connectivity or strong connectivity. An existing study [6] shows that using $R \approx 200$ sketches achieves a similar solution quality to $R' = 10^4$ MC experiments. This method is very expensive in memory usage.

3.1 Sketch Compression

Our first contribution is a **compression scheme for sketches** on undirected graphs, which allows for user-defined compression ratios (α). Similar to existing work, *PaC-IM* memoizes connected components (CC) of the sketches but *avoids the $O(Rn)$ space to store per-vertex information.* Our idea is a combination (and thus a tradeoff) of memoization and simulation. The idea is to memoize the CC information only for *centers* $C \subseteq V$, where $|C| = \alpha n$, and $\alpha \in [0, 1]$ is a user-defined compression ratio. When we evaluate the spread of a vertex, we average the size of CC that the vertex belongs to on R sketches. On a certain sketch, a local simulation from the vertex will retrieve the CC size by either encountering a center (the CC size is the same as the center), or traversing the whole CC (the CC size is the search size). As such, the sketch saves the space by a factor of α at the cost of increasing the work by $O(\min\{T, 1/\alpha\})$, where T is the number of reachable vertices in a full simulation. The parameter α provides a trade-off between the simulation and memorization: when $\alpha = 0$, it is a simulation-based algorithm; when $\alpha = 1$, it is a memorization-based algorithm. The comparison of our algorithm and existing algorithms in space and time complexity is shown in Tab. 1. The influence of α for sketch-construction is shown in Fig. 1. *Win-Tree* and *P-tree* are two IM algorithms that

	Ours	**Simulation**	**Memoization**
space	$O((1+\alpha R)n)$	O(n)	O(Rn)
work for σ	$(R \cdot \min(1/\alpha, T))$	$O(RT)$	$O(R)$
parameter	$\alpha \in [0, 1]$	$\alpha = 0$	$\alpha = 1$

Table 1: Compare the complexity of different approaches to estimating influence by R spread experiments. Our algorithm can trade-off between simulation (running time) and memorization (memory usage) by a parameter α. The existing pure simulation way and memorization way can be viewed as the extreme case when α is 0 and 1, respectively.

use different data structures for seed selection, but both use the same sketch compression algorithm for preprocessing (and store the same sketch information). As the α decreases (compress more), the memory usage for both decreases.

3.2 Seed Selection Parallelization

Even with sketch compression, the Greedy algorithm itself is expensive because it needs to re-evaluate all the vertices to select one seed. Many SOTA solutions use the CELF [17] optimization to reduce the number of calls for re-evaluation. In a nutshell, CELF is an iterative approach that lazily evaluates the marginal gain of vertices in seed selection, one at a time. While laziness reduces the number of vertices to evaluate, CELF is *inherently sequential* because the current round relies on the results of the previous round. Therefore, CELF is hard to parallelize, and existing parallel solutions ([13, 19]) only parallelize the computation for re-evaluating one vertex but still re-evaluate vertices one by one.

Our second contribution is **two new parallel data structures** to reduce running time for seed selection. To overcome this challenge, we proposed two novel solutions that achieve high parallelism for CELF. The challenge is evaluating more vertices in parallel while avoiding unpromising vertices as in CELF. Our first approach gives a theoretically efficient solution based on the *P-tree* [3] (a parallel balanced BST), and the work- and span-efficiency is approached by a prefix-doubling scheme. More formally, we have a sophisticated analysis to show that the P-tree solution evaluates no more than twice as many vertices as sequential CELF but brings in high parallelism. Our seed selection based on *P-tree* will select the same seed set as CELF with $O(n \log n + W_{CELF})$ work and $\tilde{O}(kD_{\Delta})$ span, where k is the number of seeds, W_{CELF} is the work (time complexity) by CELF, and D_{Δ} is the span to evaluate one vertex. To further improve the practical performance, we showed our second solution, referred to as *Win-Tree*, based on a winning tree (aka. tournament tree) that does not keep the total ordering of the objects as in the BSTs, and thus avoids the $O(n \log n)$ work term for sorting. It is highly asynchronous and has lower space usage compared with *P-tree*, leading to slightly better overall performance. However, due to the asynchrony, *Win-Tree* does not have strong worst-case bounds as *P-tree*. The comparison between *P-tree* and *Win-Tree* in memory usage can be found in Fig. 1. *Win-Tree* always has smaller memory usage than *P-tree*.

We tested the scalability of systems and baseline algorithms: Infuser ([13]) and Ripples ([19]). Our algorithm scales well, and we run faster when the number of cores increases. The baseline algorithms run even slower than the number of cores increases because they did not parallel the seed selection well. When the number of cores increases, the improvement brought by parallelism is limited and can not overcome the increased parallelism overhead.

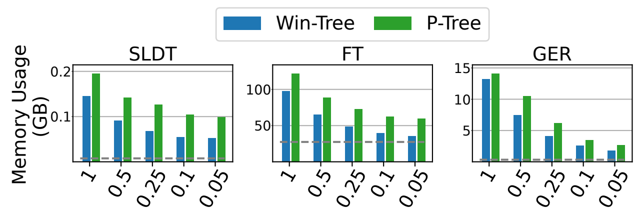

Figure 1: The influence of compression rate α in memory usage, and the comparison between *P-tree* and *Win-Tree* in memory usage. The x-axis represents the α. The y-axis shows the total memory usage. The gray horizontal line represents the basic memory we need to load the graph.

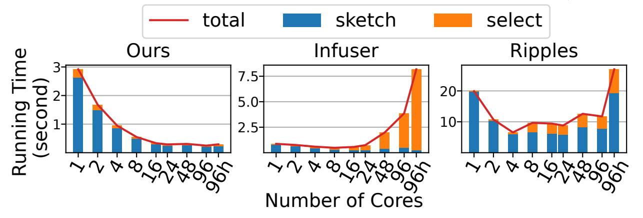

Figure 2: The scalability for different IM algorithms on graph SLDT. The x-axis shows core counts, and the y-axis shows running time.

Note that the two parallel seed selection solutions based on *P-tree* and *Win-Tree* work on both directed and undirected graphs and most diffusion models. They are also potentially extendable to many other iterative algorithms for optimization problems (if they have the submodular property).

4 STRONG CONNECTIVITY

Given a directed graph $G = (V, E)$, we denote $n = |V|$ and $m = |E|$, and use D as the diameter of G. We say v is ***reachable*** to u if there is a path from v to u. Two vertices v and u are ***strongly connected*** if v is reachable to u and u is reachable to v. An SCC is a maximal set of vertices in G that are strongly connected.

As a fundamental graph algorithm, SCC is widely studied. Sequentially, Kosaraju's algorithm [1] and Tarjan's algorithm [22] can compute SCC in $O(m)$ cost. Although there exist many parallel implementations, most of them are optimized for graphs having a low diameter and/or one large SCC. When either of the assumptions is not satisfied, these algorithms may have unsatisfactory performance. In our experiments, we found that existing algorithms on a 96-core machine can even be slower than Tarjan's sequential algorithm on many k-NN and lattice graphs (see Fig. 3 (c)). Unfortunately, they have many applications. For example, the k-NN graphs are used in unsupervised learning and lattice graphs are used in computational chemistry.

Most existing parallel SCC algorithms are based on reachability search. Existing implementations use Breadth-First Search (BFS) for reachability, which needs $O(D)$ rounds of synchronization, where D is the diameter of the graphs. The synchronization cost is high and is not shown in the theory bound. We propose a novel idea to reduce rounds of synchronization in reachability searches and thus reducing the parallelism overhead. The novel idea is referred to as ***vertical granularity control*** (VGC) optimization. Furthermore, to maintain the frontier more efficiently (and correctly) in VGC, we propose a novel data structure called the ***parallel hash bag***. It supports efficient insertion and extract-all operations for an unordered set structure and dynamic resizing in an efficient manner.

4.1 Vertical Granularity Control (VGC)

For the reachability queries, unlike parallel BFS, which only visits the direct neighbors of the vertices in the frontier, we maintain a ***local queue*** to do "local BFS" from each vertex in the frontier until τ edges have been touched. Such optimization effectively works as a *granularity control*: each thread is guaranteed to perform a reasonably large amount of work, which hides the scheduling overhead. Therefore, each round likely pushes the frontier by more than one hop, which reduces the overall number of rounds. This approach saves the number of synchronization rounds by 3-200× than BFS, and improves the performance, especially on large-diameter graphs (up to 14.7× speedup).

4.2 Parallel Hash Bag

When knowing the upper bound of the element size (n for reachability), our hash bag supports current insertion with $O(1)$ cost in expectation and listing all inserted elements (packing the vertices in the frontier together) with $O(s)$ work and $O(\log s)$ span, where s is the size of inserted elements. By supporting hash bags to maintain frontiers, our new algorithm avoids visiting the edges for the second time, and improves the performance of the existing GBBS ([9]) implementation by 1.5-4.3×.

We tested the scalability of our algorithm and baseline algorithms (GBBS [9], iSpan [15], MultiStep [21]). The results are shown in Fig. 4. We find that our algorithm is better than existing algorithms mainly because we have better scalability. On large diameter graphs, such as SQR', GL5, and COS5, the speedup of existing algorithms does not increase and even decreases as the number of cores increases.

Our techniques, VGC and parallel hash bags, have the potential to improve the performance of all the traversing-based algorithms on large-diameter graphs. We extend them to LE-List, Connectivity search, and BFS. The performance of BFS compared to other baseline algorithms is shown in Fig. 3(a). We implemented a series of graph traversal algorithms, including SSSP, BFS, SCC, BCC, and so on. We publish them in the open-source library PASGAL[11].

5 BI-CONNECTIVITY

Even though the techniques VGC and parallel hash bags are efficient in practice for many graph traversal algorithms, such as SCC, BFS, and Connectivity, they do not improve the theory bound of the algorithms. For the Bi-connectivity (BCC) problem, we proposed an algorithm that is efficient in theory and practice.

Graph biconnectivity is one of the most fundamental graph problems. Given an *undirected* graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, a ***connected component (CC)*** is a maximal subset in V such that every two vertices in it are connected by a path. A ***biconnected component (BCC)*** (or blocks) is a maximal subset $C \subseteq V$ such that C is connected and remains connected after removing any vertex $v \in C$. In this paper, we use BCC (or CC) for both the biconnected (or connected) component in the graph and the problem of computing all BCCs (or CCs). BCC has extensive applications such as planarity testing [5], centrality computation [20], and network analysis [18].

The performance bottleneck of previous BCC algorithms either comes from the use of BFS that requires $O(D)$ rounds of global

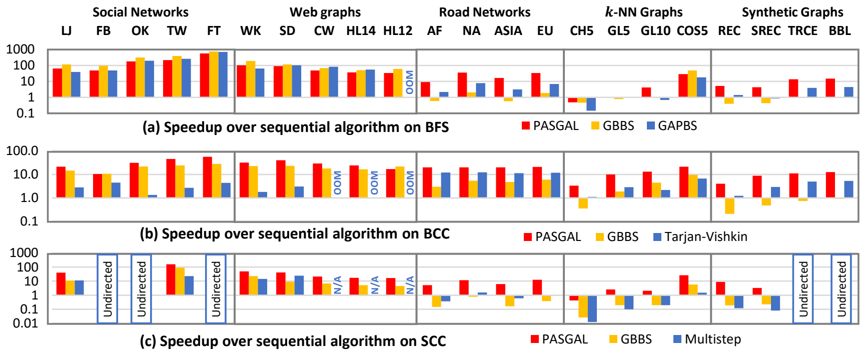

Figure 3: The y-axis shows speedup over the sequential algorithm. Greater is better.

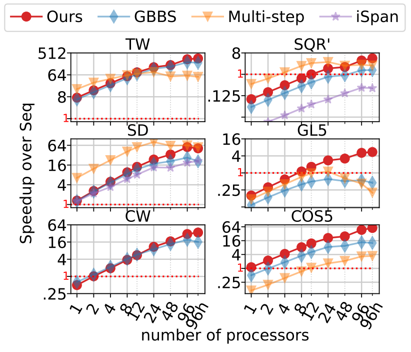

Figure 4: Scalability of SCC. The y-axis shows speedup over the sequential algorithm.

synchronizations (e.g., GBBS [8]), or requires $O(m)$ auxiliary space and does not scale to large graphs (e.g., Tarjan-Vishkin [23]). We redesigned the BCC algorithm to avoid the used of BFS. Our algorithm creates a skeleton graph based on any spanning tree of the input graph. The key idea is to carefully identify some *fence* edges, which indicate the "boundaries" of the BCCs. Our FAST-BCC achieves $O(n+m)$ work (total number of operations), polylogarithmic span (longest parallel dependency chain), and $O(n)$ auxiliary space. We carefully analyze the correctness of our algorithm, which is highly non-trivial. The comparison of BCC with other baselines is shown in Fig. 3 (b). FAST-BCC is the fastest on all tested 27 graphs. On average (geometric means), FAST-BCC is 3.1× faster than the best existing baseline on each graph.

6 FUTURE WORK

There are numerous interesting directions for future work. 1) Parallelize other Influence Maximization algorithms. Instead of the forward sketch-based algorithm we parallelized, another backward sketch-based algorithm can sample sketches adaptively for different graphs and generalize to both directed and undirected graphs. 2) Parallelizing multi-source BFS algorithms and their applications. In many applications, we need to run more than one single BFS. In addition to exploiting thread-level parallelism, multi-BFS can exploit bit-level parallelism. We need to carefully design so that both of them can be fully utilized and simple to adapt to different real-world applications.

7 SUMMARY

The papers highlight several common challenges and innovative solutions in parallel graph algorithms, which are parallelizing iterative algorithms, space efficiency, parallel data structures, and synchronization efficiency. By addressing these themes, researchers can contribute to the development of more efficient and scalable algorithms on data science problems, such as Influence Maximization and finding Strong Connectivity, enhancing the ability to analyze and interpret large-scale datasets.

REFERENCES

[1] V Aho Alfred, E Hopcroft John, D Ullman Jeffrey, V Aho Alfred, H Bracht Glenn, D Hopkin Kenneth, C Stanley Julian, Brachu Jean-Pierre, Brown A Samler, Brown A Peter, et al. 1983. *Data structures and algorithms.* USA: Addison-Wesley.
[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. 34, 2 (01 Apr 2001).
[3] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. 9, 2 (2022), 1–41.
[4] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. 46, 5 (1999), 720–748.
[5] John M Boyer and Wendy J Myrvold. 2006. Simplified $o(n)$ planarity by edge addition. 5 (2006), 241.
[6] Suqi Cheng, Huawei Shen, Junming Huang, Guoqing Zhang, and Xueqi Cheng. 2013. Staticgreedy: solving the scalability-accuracy dilemma in influence maximization. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management.* 509–518.
[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition).* MIT Press.
[8] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. 8, 1 (2021), 1–70.
[9] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E Blelloch, and Julian Shun. 2020. The graph based benchmark suite (GBBS). In *International Workshop on Graph Data Management Experiences & Systems (GRADES).* 1–8.
[10] Xiangyun Ding, Yan Gu, and Yihan Sun. 2024. Parallel and (Nearly) Work-Efficient Dynamic Programming. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures.* 219–232.
[11] Xiaojun Dong, Yan Gu, Yihan Sun, and Letong Wang. 2024. Brief Announcement: PASGAL: Parallel And Scalable Graph Algorithm Library.
[12] Xiaojun Dong, Letong Wang, Yan Gu, and Yihan Sun. 2023. Provably Fast and Space-Efficient Parallel Biconnectivity. 52–65.
[13] Gökhan Göktürk and Kamer Kaya. 2020. Boosting parallel influence-maximization kernels for undirected networks with fusing and vectorization. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2020), 1001–1013.
[14] Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. 2023. Parallel Longest Increasing Subsequence and van Emde Boas Trees.
[15] Yuede Ji, Hang Liu, and H Howie Huang. 2018. ispan: Parallel identification of strongly connected components with spanning trees. IEEE, 731–742.
[16] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. 137–146.
[17] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. 2007. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining.* 420–429.
[18] MEJ Newman and Gourab Ghoshal. 2008. Bicomponents and the robustness of networks to failure. *Physical review letters* 100, 13 (2008), 138701.
[19] Diana Popova, Naoto Ohsaka, Ken-ichi Kawarabayashi, and Alex Thomo. 2018. Nosingles: a space-efficient algorithm for influence maximization. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management.* 1–12.
[20] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit Ç atalyiirek. 2013. Incremental algorithms for closeness centrality. In *IEEE International Conference on Big Data.* IEEE, 487–492.
[21] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. IEEE, 550–559.
[22] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. 1, 2 (1972), 146–160.
[23] Robert E Tarjan and Uzi Vishkin. 1985. An efficient parallel biconnectivity algorithm. 14, 4 (1985), 862–874.
[24] Letong Wang, Xiangyun Ding, Yan Gu, and Yihan Sun. 2023. Fast and Space-Efficient Parallel Algorithms for Influence Maximization. arXiv:2311.07554 [cs.DS]
[25] Letong Wang, Xiaojun Dong, Yan Gu, and Yihan Sun. 2023. Parallel Strong Connectivity Based on Faster Reachability. 1, 2 (2023), 1–29.