ParLeiden: Boosting Parallelism of Distributed Leiden Algorithm on Large-scale Graphs

Yongmin Hu*† Jing Wang*†‡ Cheng Zhao† Yibo Liu†‡ Cheng Chen† Xiaoliang Cong† Chao Li‡
{huyongmin,zhaocheng.127,chencheng.sg,congxiaoliang}@bytedance.com,
{jing618, liuyib,}@sjtu.edu.cn, lichao@cs.sjtu.edu.cn
Douyin Vision Co., Ltd.† Shanghai Jiao Tong University‡

ABSTRACT

Leiden algorithm has demonstrated superior efficacy compared to traditional Louvain algorithms in the field of community detection. However, parallelizing the Leiden algorithm while imposing community size limitations brings significant challenges in big data processing scenarios. We present **ParLeiden**, a pioneering parallel Leiden strategy designed for distributed environments. By thread locks and efficient buffers, we effectively resolve community joining conflicts and reduce communication overheads. We can run Leiden algorithm on large-scale graphs and achieve performance speedup on up to 9.8× than baselines.

KEYWORDS

Leiden, parallelism, graph processing, distributed computing

1 INTRODUCTION

High-quality community searching and clustering on large graph datasets has become a significant concern in both academia and industry, in fields of social network analysis(SNA), network centralityand biology area[3, 4, 6]. Traag et al. proposed Leiden[5] in 2019 based on Louvain[3, 6], a more efficient algorithm with more accurate community uncovering. Leiden guarantees the internal community connection by adding refinement steps to search for new sub-community in the move node phase. In scenarios of risky community mining, Leiden supports the community size limitation to narrow search scopes. With the explosion of data scale and network complexity, distributed and multi-thread design of Leiden algorithm becomes crucial to scaling the algorithm to larger graphs.

However, parallelizing Leiden is never simple. The main difficulty is that the naive parallel implementation can not guarantee the equivalence with the original execution flow, which depends on serial computation to ensure correctness, as shown in Figure 1-(a). Existing works[1, 2] also try to design parallel community detection for high performance. However, their system performs unstable speedup even sometimes has worse performance compared with the single-thread design of the original Leiden[5]. Naively adopting parallel interfaces may not bring performance benefits.

Conflict issues across multiple threads. Flexible maximum community size limit in Leiden is designed to narrow the searching scale but brings great challenges for parallelism design. When a community cap is given, multiple nodes may apply to join the community at the same time, which involves conflicts of community updates, as shown in Figure 1-(b). The community updating procedure in the parallel workflow can be totally different from the hidden logic of node selection that is guaranteed by the serial computing flow. In the original Leiden algorithm, each node transfers all its neighbors by order of quality difference and selects the neighbor with maximum quality improvement of joining the neighbor's community. Multiple threads generate isolated ordered values of neighbors thus requiring thread locks to handle the joining conflict. However, simply adopting signal-based thread locks cause unacceptable performance degradation. Worsely, they can not ensure the quality difference comparison in each atomic operation.

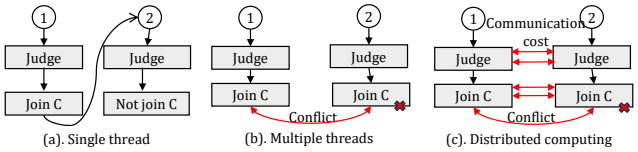

Figure 1: The challenge of parallel Leiden algorithm.

Data synchronization problem on distributed servers. Furthermore, distributed parallel Leiden also faces the problem of data consistency based on unacceptable communication costs, as shown in Figure 1-(c). Parallelizing the Leiden algorithm involves communication and synchronization operations between multiple computing nodes. Vertices on different servers may need to join the same community with size limitations and then update community information on each server. This involves multiple quality difference comparisons and data synchronization across servers. The distribution correctness requires the inclusion of distributed locks, but generic distributed locks perform poorly on data synchronization.

In this work, we proposed parallel Leiden strategy **ParLeiden** for high-performance distributed and multi-thread community detection on large graphs. 1) We design thread locks and shared cache to lift the competition of multiple vertices joining the community with correctness assurance. 2) We adopt parallel data caching, message merging, and broadcast synchronizing to reducing communication overhead with distributed data consistency guarantee.

2 PARLEIDEN DESIGN

Each iteration of Leiden algorithm consists of three phases: (1) local moving of each vertex to label the to-be-joined community, (2) refinement of node inside each community and generating sub-communities, (3) aggregating the nodes in each refined sub-community to construct a new graph. The move phase and refine phase take most of the iteration that can be parallelized. Our framework consists of two parts: monolithic thread-level parallelism and distributed server-level parallelism. In Figure 2, The left shows the execution flow on a single thread while the right shows the parallel design with thread locks and shared-data cache on multiple threads.

*These authors contribute equally to this work.

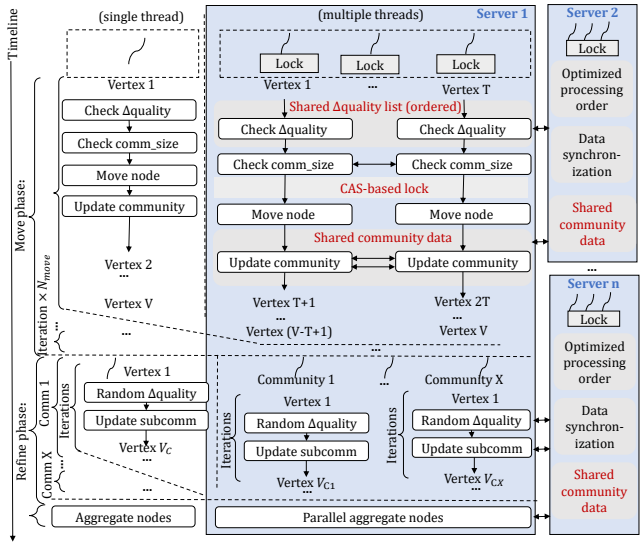

Figure 2: The comparison between single-thread Leiden algorithm and multi-thread ParLeiden design.

Monolithic Thread-level parallelism. We design thread locks to satisfy the original data updating order as much as possible. At the *move node* phase, We design CAS locks with upper bounds to resolve community joining conflicts in each thread under the condition of maximum community size limitation. We use shared memory to cache quality difference values and community updates for multi-thread data synchronization during node-level parallelism. At the *refine node* phase, each node re-searches the neighbors and joins the community that is still under the limited community size. We use multi-threads to directly process multiple communities for community-level parallelism, and we use shared memory caching to synchronize community update data.

Distributed Server-level parallelism. On distributed servers, We adopt hot data caching to improve execution efficiency. We use a Gemini-like approach to slice the edge data to multiple machines, keep a copy of community information related to each machine (node state, community information, etc.), and distribute new data to other machines in parallel after the current machine generates new data. The traversal order of the machines is randomly determined by MPI. The inner-server data conflict and consistency are guaranteed by monolithic thread-level parallelism design.

At the *move node* phase, we use hot data caching with parallel broadcast synchronization to optimize the distributed execution process. We cache the vertices that will join the community and their quality information in a single machine to support fast computation. We traverse each machine sequentially and synchronize the updated community information to other machines in parallel to ensure that the information on each machine is up-to-date. At the *refine node* phase, we optimize data transfer efficiency and reduce communication overheads by combining transferred data and parallel broadcast synchronization. We optimize the processing order of vertices to reduce the cycle synchronization in BSP (Bulk Synchronous Parallel) computing model. We combine the small data into aggregated data chunks to further reduce the communication cost. We also synchronize updated community information to other machines by parallel broadcasting.

Graph Dataset	Abbr.	$\|V\|$	$\|E\|$	Size (GB)
Amazon	AM	334863	925872	0.1
Soc-Pokec	SP	1632803	30622564	0.41
LiveJournal	LJ	3,997,962	34,681,189	0.66
Twitter	TW	17,069,982	476,553,560	18.08
Friendster	FT	65,608,366	1,806,067,135	27.16

Table 1: Evaluated graph datasets.

Dataset	Leiden	KatanaGraph	Parleiden-S	Parleiden-D
Amazon	13.58	4.93	1.53	6.87
Soc-Pokec	312.39	282.41	29.47	25.74
Livejournal	593.99	448.38	48.46	59.77
Twitter*	>24h	>24h	686.73	152.76
Friendster*	>24h	>24h	1268.99	279.83
Amazon-5k	13.90	-	1.54	14.19
Soc-Pokec-5k	496.93	-	36.25	96.28
Livejournal-5k	729.41	-	45.29	111.82
Twitter-5k*	>24h	-	719.68	200.491
Friendster-5k*	>24h	-	1174.03	284.50

Table 2: Executing time(s) on different graph datasets. The content within the parentheses after the dataset indicates the limit on community size (CS), while no parentheses indicate an unlimited CS. KatanaGraph does not support limiting CS by marking '-'. Over '*' datasets, quality differences in the move node phase are set as 0.01, while the others are set as 0.

3 EVALUATION

In the evaluation, we use a cluster with 8 real machines to evaluate the performance of our design. Each server is equipped with 48 core CPU to handle 96 threads. We evaluate 5 graph datasets from SNAP as shown in Table 1. We compare **ParLeiden** (single node version **ParLeiden-S** and distributed version **ParLeiden-D**)with the Original Leiden[5] implementation (without parallelism) **Leidenalg** and the implementation version of the baseline design on **KatanaGraph**[1] framework. We can achieve equivalent quality under different community sizes. Table 2 presents the detailed latency of our work and the baselines with and without community size limitations. Overall, we can support large graphs to search communities. In each iteration, we perform up to 9.8 × faster than single-thread baseline and 9.6 × faster than KanataGraph in monolithic and distributed scenarios. Our distributed design uses 8 servers and further speeds up the single-node design on large graphs. For more details on the evaluated results, please refers to our poster.

Conclusion. We improve the overall performance of Leiden algorithm on large-scale graphs by fine-grained parallelism design. We will continue to improve our design in the future work.

REFERENCES

[1] 2023. igraph. http://igraph.sourceforge.net/.
[2] 2023. KatanaGraph. https://katanagraph.ai/.
[3] Sayan Ghosh et al. 2018. Distributed louvain algorithm for graph community detection. In *IPDPS*.
[4] Thomas Magelinski et al. 2021. Measuring node contribution to community structure with modularity vitality. *IEEE TNSE* (2021).
[5] Vincent A Traag et al. 2019. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific reports* (2019).
[6] Jianping Zeng and Hongfeng Yu. 2018. A Scalable Distributed Louvain Algorithm for Large-Scale Graph Community Detection. In *CLUSTER*.