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

Yongmin Hu *[1], Jing Wang *[1,2], Cheng Zhao✉[1], Yibo Liu[1,2], Cheng Chen[1], Xiaoliang Cong[1], Chao Li[2]
{huyongmin,zhaocheng.127,chencheng.sg,congxiaoliang}@bytedance.com, {jing618, liuyib}@sjtu.edu.cn, lichao@cs.sjtu.edu.cn
ByteDance[1], Shanghai Jiao Tong University[2]

Introduction & Background

High-quality community searching and clustering on large graph datasets has become a significant concern in both industry [1,2,3,4] and academia[3,5,6].

- Leiden[4] is a SOTA algorithm based on Louvain[3, 6], a more efficient community detection method with well-connected communities.
- The distributed parallel design of the Leiden algorithm becomes crucial as the size of graph data grows much larger.

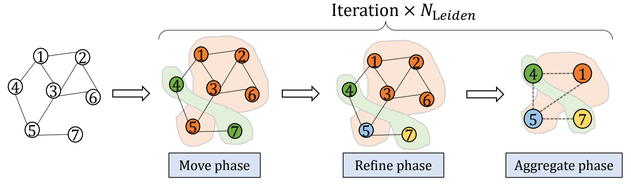


Fig 1: The execution workflow of Leiden Algorithm.

We design **ParLeiden**, a fast parallel Leiden strategy that processes large-scale graph data with multiple threads on distributed platforms.

Challenges

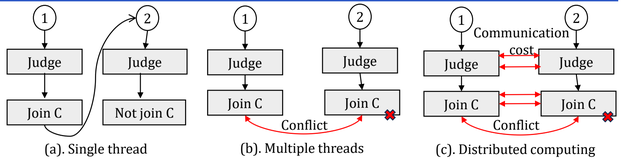


Fig 2: The main challenges of parallel Leiden: the data conflict and the communication cost.

- Community size limitation[5] is an essential requirement in risk control and anomaly detection scenarios to avoid results with overly large communities.
- However, community limit brings significant challenge when parallelizing Leiden.

Parallel challenges of Leiden Algorithm

1). Challenge 1: The correctness

- Concurrent workers face **conflict** upon reaching the limit of community size.
- Basic thread locks **cannot ensure** correctness without sacrificing the parallelism.

2). Challenge 2: The overhead

- Concurrent community updating always requires the latest quality value computed by other threads and servers, causing massive **waiting overhead**.
- Random community data access and synchronization across servers in parallel environment causes unacceptable data **communication cost**.

Novelty1: ***We design threads locks and shared cache to lift the competition of multiple vertices joining the community with correctness assurance.***

Novelty2: ***We adopt centralized computing model and optimized processing order on distributed servers to reduce communication overhead with consistency guarantee.***

ParLeiden Design

Monolithic Thread-level parallelism Design

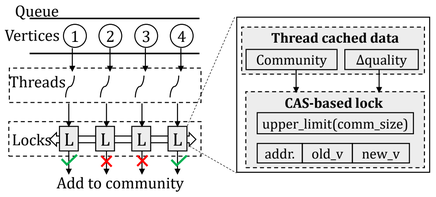


Fig 3: CAS-based thread locks with cached intermediate data.

- We design CAS-based thread locks to handle the joining conflict of joining the same community.
- We use shared memory to cache the community and quality information of each traversed node.

Distributed Server-level parallelism Design

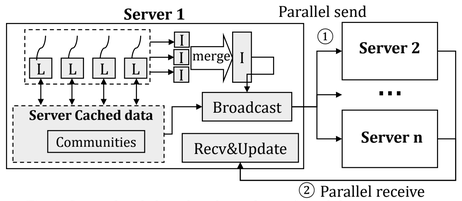


Fig 4: Server-level shared cache and transferred data merging.

- We adopt centralized distributed computing model including message merging and parallel broadcast.
- We optimize the processing order of vertices to reduce the cycle synchronization cost across servers.

ParLeiden Workflow

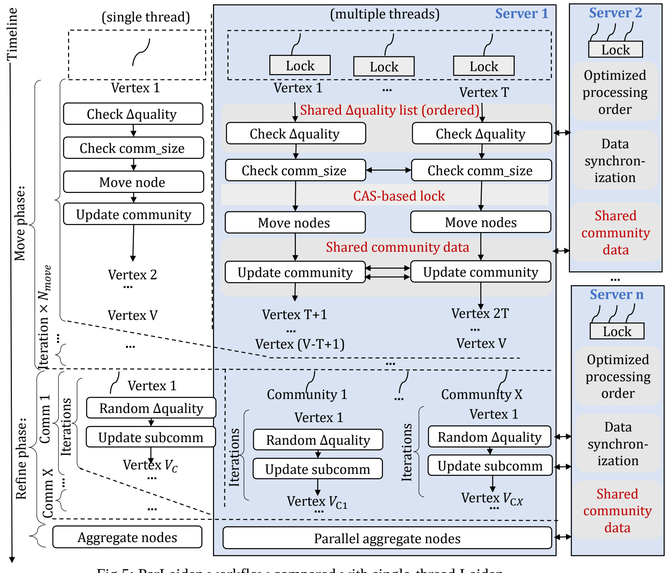


Fig 5: ParLeiden workflow compared with single-thread Leiden.

(1). Move phase: We use multiple threads with proposed locks to parallel process each node's movement to the neighbors. We calculate the quality difference of the leaving and joining community to decide the community labels. We update each node label iteratively till convergence.

(2). Refine phase: We re-traversal each community that is generated in the move node phase with different threads. The refinement will search new communities only inside of each community so the execution is totally in parallel.

(3). Aggregate phase: We merge the nodes in sub-communities in the refine phase into a single vertex and merge the inner and outer edges. We use the basic parallel APIs to aggregate the new graph.

(4). Repeat till convergence: We search communities iteratively on the new graph until there's no quality improvement on every community.

Experimental Evaluation

Datasets are from http://snap.stanford.edu/data/index.html

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

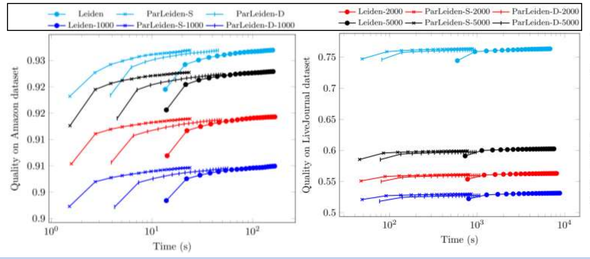
The evaluated baselines:

- **Leiden[5]**: Original Leiden algorithm
- **KatanaGraph[2]**: State-of-the-art parallel Leiden implementation
- **ParLeiden**: Our parallel Leiden algorithm

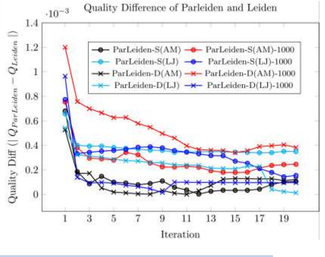
ParLeiden-S: single node version
ParLeiden-D: distributed version

Baseline-number (e.g. Leiden-1000): The number is community size upper limit.

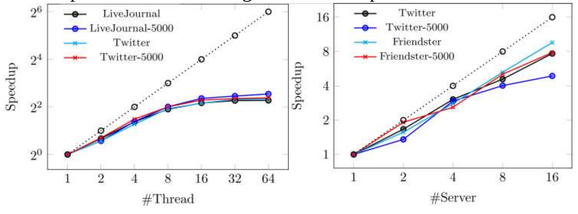
Result 1: We can achieve almost equivalent quality compared with **Leiden** on different **community size limits** and different **datasets.**



Result 2: The quality difference is **negligible** and **decreases** over iterations on ParLeiden-S(96 threads) and ParLeiden-D(2 servers).



Result 3: ParLeiden shows good speedup when scaling up with multiple threads and scaling out with multiple servers.



Result 4: The speedup of **ParLeiden** over baselines on each iteration**.**

Datasets	Baselines	Speedup
LiveJournal	Leiden	7.75×
	Leiden-5000	9.81×
	KatanaGraph	9.25×
Soc-Pokec	Leiden	7.56×
	Leiden-5000	8.89×
	KatanaGraph	9.58×

ParLeiden can handle significantly larger graphs, surpassing the capacity of the baselines. We achieve performance speedup on up to 9.8× than single thread baseline[3] with closed quality and 9.6× than KatanaGraph[4] in each iteration.

Conclusion

- In this paper we explore the parallelism design of Leiden algorithm on multi-thread monolithic server and distributed servers.
- We design thread locks and efficient buffers to solve community joining conflicts and reduce communication overheads.
- Our system ParLeiden can handle significantly larger graphs and achieves performance speedup on up to 9.8× than baseline methods.

Reference

[1] igraph, http://igraph.sourceforge.net/.
[2] 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.

∗These authors contributed equally to this work.