Fast Leiden Algorithm for Community Detection in Shared Memory Setting

Subhajit Sahu
subhajit.sahu@research.iiit.ac.in
IIIT Hyderabad
Hyderabad, Telangana, India

Kishore Kothapalli
kkishore@iiit.ac.in
IIIT Hyderabad
Hyderabad, Telangana, India

Dip Sankar Banerjee
dipsankarb@iitj.ac.in
IIT Jodhpur
Karwar, Rajasthan, India

ABSTRACT

Community detection is the problem of identifying natural divisions in networks. Efficient parallel algorithms for identifying such divisions is critical in a number of applications, where the size of datasets have reached significant scales. This paper presents one of the most efficient implementations of the Leiden algorithm, a high quality community detection method. On a server equipped with dual 16-core Intel Xeon Gold 6226R processors, our Leiden implementation, which we term as GVE-Leiden, outperforms the original Leiden, igraph Leiden, NetworKit Leiden, and cuGraph Leiden (running on NVIDIA A100 GPU) by 436×, 104×, 8.2×, and 3.0× respectively — achieving a processing rate of 403M edges/s on a 3.8B edge graph. In addition, GVE-Leiden improves performance at an average rate of 1.6× for every doubling of threads.

CCS CONCEPTS

• **Theory of computation → Graph algorithms analysis**.

KEYWORDS

Community detection, Parallel Leiden implementation

ACM Reference Format:
Subhajit Sahu, Kishore Kothapalli, and Dip Sankar Banerjee. 2024. Fast Leiden Algorithm for Community Detection in Shared Memory Setting. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673146

1 INTRODUCTION

Community detection is the problem of identifying subsets of vertices that exhibit higher connectivity among themselves than with the rest of the network. The identified communities are intrinsic when based on network topology alone, and are disjoint when each vertex belongs to only one community. These communities, also known as clusters, shed light on the organization and functionality of the network. This problem has applications in topic discovery, protein annotation, recommendation systems, and targeted advertising [10]. One of the difficulties in the community detection problem is the lack of apriori knowledge on the number and size distribution of communities [2]. The Louvain method [2] is a popular heuristic-based approach for community detection which employs a two-phase approach, comprising a local-moving phase and an aggregation phase, to iteratively optimize the modularity metric — a measure of community quality [18].

Despite its popularity, the Louvain method has been observed to produce internally-disconnected and badly connected communities. To address these shortcomings, Traag et al. [30] propose the Leiden algorithm. It introduces an additional refinement phase between the local-moving and aggregation phases. The refinement phase allows vertices to explore and potentially form sub-communities within the communities identified during the local-moving phase. This enables the Leiden algorithm to identify well-connected communities [30].

However, applying the original Leiden algorithm to massive graphs has raised computational bottlenecks, mainly due to its inherently sequential nature — similar to the Louvain method [11]. In contexts where scalability is paramount, the development of an optimized parallel Leiden algorithm becomes imperative — especially in the multicore/shared memory setting, due to its energy efficiency and the prevalence of hardware with large memory sizes. Existing studies on parallel Leiden algorithm [19, 31] propose a number of parallelization techniques, but do not address optimization for the aggregation phase of the Leiden algorithm, which emerges as a bottleneck after the local-moving phase of the algorithm has been optimized. In addition, a number of optimization techniques that apply to the Louvain method also apply to the Leiden algorithm.

In this paper, we present our parallel multicore implementation of the Louvain algorithm[1]. It incorporates several optimizations, including parallel prefix sums, preallocated Compressed Sparse Row (CSR) data structures for community vertex identification and super-vertex graph storage during aggregation, fast collision-free per-thread hash tables for the local-moving and aggregation phases, and prevention of unnecessary aggregations — enabling it to run significantly faster than existing implementations. Additionally, we employ a greedy refinement phase where vertices optimize for delta-modularity within their community bounds, yielding improved performance and quality compared to a randomized approach. Furthermore, we utilize established techniques such as asynchronous computation, OpenMP's dynamic loop schedule, threshold-scaling optimization, and vertex pruning. To the best of our knowledge, our implementation is the most efficient implementation of Leiden algorithm on multicore CPUs to date. We conduct comprehensive comparisons with other state-of-the-art implementations, including multi-core, GPU-based implementations, detailed in Table 1. Indirect comparisons are outlined in Section 5.5.

[1] https://github.com/puzzlef/louvain-communities-openmp

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08…$15.00
https://doi.org/10.1145/3673038.3673146

Table 1: Speedup of our multicore Leiden implementation compared to other state-of-the-art implementations.

Leiden implementation	Parallelism	Our Speedup
Original Leiden [30]	Sequential	22×
igraph Leiden [5]	Sequential	50×
NetworKit Leiden [26]	Parallel	20×
cuGraph Leiden [13]	Parallel (GPU)	3.0×

2 RELATED WORK

The Louvain method, introduced by Blondel et al. [2] from the University of Louvain, is a greedy modularity-optimization based algorithm for community detection [15]. While it is favored for identifying communities with high modularity, it often results in internally disconnected communities. This occurs when a vertex, acting as a bridge, moves to another community during iterations. Further iterations aggravate the problem, without decreasing the quality function. Further, the Louvain method may identify communities that are not well connected, i.e., splitting certain communities could improve the quality score — such as modularity [30].

To address these limitations, Traag et al. [30] from the University of Leiden, propose the Leiden algorithm. It introduces a *refinement phase* after the local-moving phase, where vertices within each community undergo constrained merges in a randomized fashion proportional to the delta-modularity of the move. This allows vertices to find sub-communities within those obtained from the local-moving phase. The Leiden algorithm guarantees that the identified communities are both well separated (like the Louvain method) and well connected. When communities have converged, it is guaranteed that all vertices are optimally assigned, and all communities are subset optimal [30]. Shi et al. [25] also introduce an additional refinement phase after the local-moving phase with the Louvain method, which they observe minimizes bad clusters. It should however be noted that methods relying on modularity maximization are known to suffer from resolution limit problem, which prevents identification of communities of certain sizes [8, 30]. This can be overcome by using an alternative quality function, such as the Constant Potts Model (CPM) [29].

We now discuss a number of algorithmic improvements proposed for the Louvain method, that also apply to the Leiden algorithm. These include ordering of vertices based on importance [1], attempting local move only on likely vertices [21, 25, 34], early pruning of non-promising candidates [11, 21, 33, 34], moving vertices to a random neighbor community [27], subnetwork refinement [30], multilevel refinement [7, 20, 25], threshold cycling [9], threshold scaling [11, 16, 17], and early termination [9].

A number of parallelization techniques have been attempted for the Louvain method, that may also be applied to the Leiden algorithm. These include using adaptive parallel thread assignment [6, 17], parallelizing the costly first iteration [32], ordering vertices via graph coloring [11], performing iterations asynchronously [25], using vector based hashtables [11], and using sort-reduce [4].

A few open source implementations and software packages have been developed for community detection using Leiden algorithm. The original implementation of the Leiden algorithm [30], called `libleidenalg`, is written in C++ and has a Python interface called `leidenalg`. NetworKit [26] is a software package designed for analyzing the structural aspects of graph data sets with billions of connections. It utilizes a hybrid with C++ kernels and a Python frontend. The package features a parallel implementation of the Leiden algorithm by Nguyen [19] which uses global queues for vertex pruning, and vertex and community locking for updating communities. igraph [5] is a similar package, written in C, with Python, R, and Mathematica frontends. It is widely used in academic research, and includes an implementation of the Leiden algorithm. cuGraph [13] is a GPU-accelerated graph analytics library, part of the RAPIDS suite of data science and machine learning libraries. It leverages the computational power of NVIDIA GPUs to perform graph analytics tasks much faster than traditional CPU-based approaches. cuGraph's core is implemented in C++ with CUDA and is used primarily through its Python interface.

3 PRELIMINARIES

Consider an undirected graph $G(V, E, w)$, where V represents the set of vertices, E the set of edges, and $w_{ij} = w_{ji}$ denotes the weight associated with each edge. In the case of an unweighted graph, we assume unit weight for each edge ($w_{ij} = 1$). Additionally, the neighbors of a vertex i are denoted as $J_i = \{j \mid (i, j) \in E\}$, the weighted degree of each vertex as $K_i = \sum_{j \in J_i} w_{ij}$, the total number of vertices as $N = |V|$, the total number of edges as $M = |E|$, and the sum of edge weights in the undirected graph as $m = \sum_{i,j \in V} w_{ij}/2$.

3.1 Community detection

Disjoint community detection involves identifying a community membership mapping, $C : V \to \Gamma$, where each vertex $i \in V$ is assigned a community-id c from the set of community-ids Γ. We denote the vertices of a community $c \in \Gamma$ as V_c, and the community that a vertex i belongs to as C_i. Further, we denote the neighbors of vertex i belonging to a community c as $J_{i \to c} = \{j \mid j \in J_i \text{ and } C_j = c\}$, the sum of those edge weights as $K_{i \to c} = \sum_{j \in J_{i \to c}} w_{ij}$, the sum of weights of edges within a community c as $\sigma_c = \sum_{(i,j) \in E \text{ and } C_i = C_j = c} w_{ij}$, and the total edge weight of a community c as $\Sigma_c = \sum_{(i,j) \in E \text{ and } C_i = c} w_{ij}$.

3.2 Modularity

Modularity serves as a metric for evaluating the quality of communities identified by heuristic-based community detection algorithms. It is calculated as the difference between the fraction of edges within communities and the expected fraction if edges were randomly distributed, yielding a range of $[-0.5, 1]$, where higher values signify superior results [3]. The modularity Q of identified communities is determined using Equation 1, where δ represents the Kronecker delta function ($\delta(x, y) = 1$ if $x = y$, 0 otherwise). The *delta modularity* of moving a vertex i from community d to community c, denoted as $\Delta Q_{i:d \to c}$, can be calculated using Equation 2.

$$Q = \frac{1}{2m} \sum_{(i,j) \in E} \left[w_{ij} - \frac{K_i K_j}{2m} \right] \delta(C_i, C_j) = \sum_{c \in \Gamma} \left[\frac{\sigma_c}{2m} - \left(\frac{\Sigma_c}{2m} \right)^2 \right] \quad (1)$$

$$\Delta Q_{i:d \to c} = \frac{1}{m}(K_{i \to c} - K_{i \to d}) - \frac{K_i}{2m^2}(K_i + \Sigma_c - \Sigma_d) \quad (2)$$

3.3 Louvain algorithm

The Louvain method [2] is an agglomerative algorithm that optimizes modularity to identify high quality disjoint communities in large networks. It has a time complexity of $O(L|E|)$, where L is the total number of iterations performed, and a space complexity of $O(|V| + |E|)$ [15]. This algorithm comprises two phases: the *local-moving phase*, in which each vertex i greedily decides to join the community of one of its neighbors $j \in J_i$ to maximize the increase in modularity $\Delta Q_{i:C_i \to C_j}$ (using Equation 2), and the *aggregation phase*, where all vertices in a community are merged into a single super-vertex. These phases constitute one pass, which is repeated until there is no further increase in modularity is observed [2].

3.4 Leiden algorithm

As mentioned earlier, the Louvain method, while effective, may identify internally disconnected communities and arbitrarily badly connected ones. Traag et al. [30] proposed the Leiden algorithm to address these issues. The algorithm introduces a *refinement phase* subsequent to the local-moving phase, wherein vertices within each community undergo constrained merges to other sub-communities within their community bounds (obtained from the local-moving phase), starting from a singleton sub-community. This is performed in a randomized manner, with the probability of joining a neighboring sub-community within its community bound being proportional to the delta-modularity of the move. This facilitates the identification of sub-communities within those obtained from the local-moving phase. The Leiden algorithm not only guarantees that all communities are well separated (akin to the Louvain method), but also are well connected. Once communities have converged, it is guaranteed that all vertices are optimally assigned, and all communities are subset optimal [30]. It has a time complexity of $O(L|E|)$, where L is the total number of iterations performed, and a space complexity of $O(|V| + |E|)$, similar to the Louvain method.

4 APPROACH

4.1 Optimizations for Leiden algorithm

We extend optimization techniques, originally designed for the Louvain method [23], to the Leiden algorithm. Specifically, we implement an *asynchronous* version of the Leiden algorithm, allowing threads to operate independently on distinct sections of the graph. While this approach promotes faster convergence, it also introduces variability into the final result [25]. To ensure efficient computations, we allocate a dedicated hashtable per thread. These hashtables serve two main purposes: they keep track of the delta-modularity associated with moving to each community connected to a vertex during the local-moving/refinement phases, and they record the total edge weight between super-vertices in the aggregation phase of the algorithm [23].

Our optimizations include using preallocated Compressed Sparse Row (CSR) data structures for identifying community vertices ($G'_{C'}$ in Algorithm 4) and storing the super-vertex graph (G'' in Algorithm 4) during aggregation, utilizing parallel prefix sum (lines 3-4, 9 in Algorithm 4), employing fast collision-free per-thread hashtables that are well separated in their memory addresses (H_t in Algorithms 2, 3, and 4), and using an aggregation tolerance of 0.8 to avoid performing aggregations of minimal utility (line 10 in Algorithm 4). Additionally, we implement flag-based vertex pruning (lines 2, 6, 14 in Algorithm 2) — instead of a queue-based one [19], utilize OpenMP's *dynamic* loop scheduling, cap the number of iterations per pass at 20 (line 3 in Algorithm 2), employ a tolerance drop rate of 10 (line 15 in Algorithm 1) — threshold scaling optimization, and initiate with a tolerance of 0.01 [23].

We attempt two approaches of the Leiden algorithm. One uses a *greedy refinement phase* where vertices greedily optimize for delta-modularity (within their community bounds), while the other uses a *randomized refinement phase* (using fast *xorshift32* random number generators), where the likelihood of selection of a community to move to (by a vertex) is proportional to its delta-modularity, as originally proposed [30]. Our results, shown in Figures 1 and 2, indicate the *greedy approach* performs the best on average, both in terms of runtime and modularity. We also try medium and heavy variants for both approaches, which disables threshold scaling and aggregation tolerance (including threshold scaling) respectively, However, we do not find them to perform well overall.

We also attempt two different variations of Parallel Leiden algorithm, one where the community labels of super-vertices (upon aggregation) is based on the local-moving phase (*move-based*), and the other where the community labels of super-vertices is based on the refinement phase (*refine-based*). Our observations indicate that both approaches have roughly the same runtime and modularity on average, as indicated by Figures 3 and 4. Accordingly, we stick to the move-based approach, which is the one recommended by Traag et al. [30]. However, refine-based approach may be more suitable for the design of dynamic Leiden algorithm (for dynamic graphs).

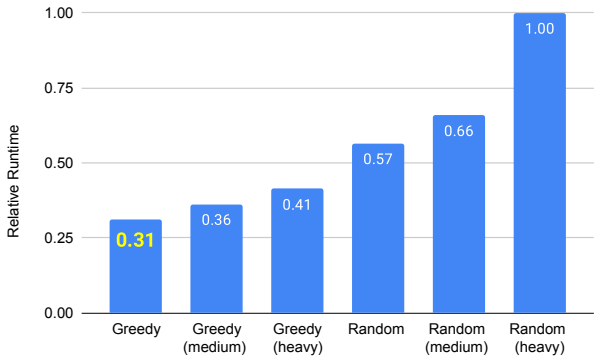


Figure 1: Average relative runtime for the *greedy* and *random* approaches (including *medium* and *heavy* variants) of parallel Leiden algorithm for all graphs in the dataset.

4.2 Our optimized Leiden implementation

We now explain the implementation of GVE-Leiden in Algorithms 1, 2, 3, and 4. Our intention is to integrate GVE-Leiden into a forthcoming command-line graph processing tool named "GVE", which simply stands for Graph(Vertices, Edges), hence the name. GVE-Leiden operates with a time complexity of $O(KM)$, where K is the total number of iterations performed, and a space complexity of

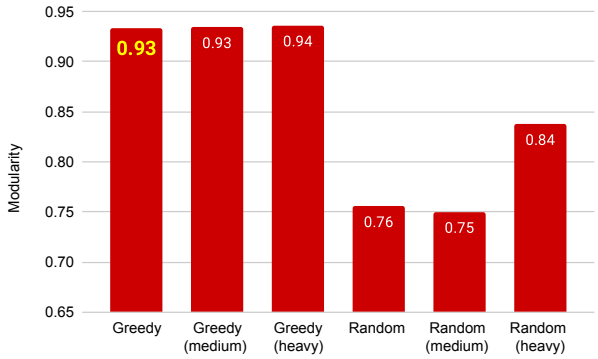


Figure 2: Average modularity for the *greedy* and *random* approaches (including *medium* and *heavy* variants) of parallel Leiden algorithm for all graphs in the dataset.

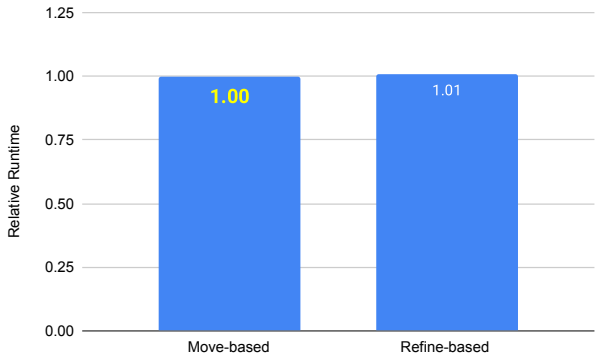


Figure 3: Average relative runtime for *move-based* and *refine-based* communities for super-vertices upon aggregation with parallel Leiden algorithm, for all graphs in the dataset.

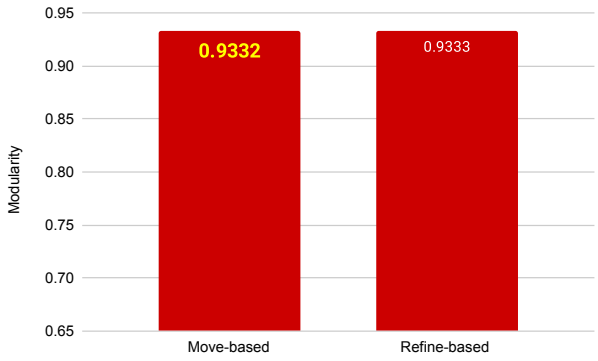


Figure 4: Average modularity for *move-based* and *refine-based* communities for super-vertices upon aggregation with parallel Leiden algorithm, for all graphs in the dataset.

$O(TN + M)$, where T represents the number of threads, and TN accounts for the collision-free hash tables H_t used per thread. Figure 5 illustrates the first pass of GVE-Leiden.

4.2.1 Main step of GVE-Leiden. The main step of GVE-Leiden (`leiden()` function) is outlined in Algorithm 1. It encompasses initialization, the local-moving phase, the refinement phase, and the aggregation phase. Here, the `leiden()` function accepts the input graph G, and returns the community membership C of each vertex. In line 2, we first initialize the community membership C for each vertex in G, and perform passes of the Leiden algorithm, limited to *MAX_PASSES* (lines 3-15). During each pass, we initialize the total edge weight of each vertex K', the total edge weight of each community Σ', and the community membership C' of each vertex in the current graph G' (line 4).

Subsequently, in line 5, we perform the local-moving phase by invoking `leidenMove()`, which optimizes community assignments. Following this, we set the *community bound* of each vertex (for the refinement phase) as the community membership of each vertex just obtained, and reset the membership of each vertex, and the total weight of each community as singleton vertices in line 6. In line 7, the refinement phase is carried out by invoking `leidenRefine()`, which optimizes the community assignment of each vertex within its community bound. If the local-moving phase converges in a single iteration, global convergence is implied and we terminate the passes (line 8). Further, if the drop in the number of communities $|\Gamma|$ is marginal, we halt the algorithm at the current pass (line 10).

If convergence has not been achieved, we proceed to renumber communities (line 11), update top-level community memberships C with dendrogram lookup (line 12), perform the aggregation phase by calling `leidenAggregate()`, and adjust the convergence threshold for subsequent passes, i.e., perform threshold scaling (line 15). The next pass commences in line 3. At the end of all passes, we perform a final update of the top-level community memberships C with dendrogram lookup (line 16), and return the top-level community membership C of each vertex in G.

4.2.2 Local-moving phase of GVE-Leiden. The pseudocode for the local-moving phase of GVE-Leiden is shown in Algorithm 2, which iteratively moves vertices between communities to maximize modularity. Here, the `leidenMove()` function takes the current graph G', community membership C', total edge weight of each vertex K' and each community Σ', the iteration tolerance τ as input, and returns the number of iterations performed l_i.

Lines 3-15 represent the main loop of the local-moving phase. In line 2, we first mark all vertices as unprocessed. Then, in line 4, we initialize the total delta-modularity per iteration ΔQ. Next, in lines 5-14, we iterate over unprocessed vertices in parallel. For each unprocessed vertex i, we mark i as processed - vertex pruning (line 6), scan communities connected to i - excluding self (line 7), determine the best community $c*$ to move i to (line 9), and calculate the delta-modularity of moving i to $c*$ (line 10). We then update the community membership of i (lines 12-13) and mark its neighbors as unprocessed (line 14) if a better community was found. In line 15, we check if the local-moving phase has converged. If so, we break out of the loop (or if *MAX_ITERATIONS* is reached). At the end, in line 16, we return the number of iterations performed l_i.

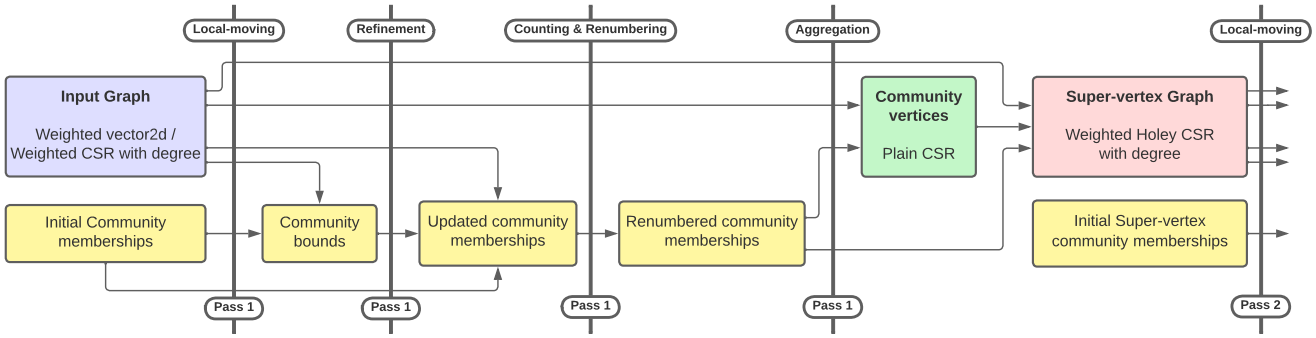


Figure 5: A flow diagram illustrating the first pass of GVE-Leiden for either a Weighted 2D-vector-based or a Weighted CSR with degree-based input graph. In the local-moving phase, vertex community memberships are updated to obtain community bounds for the refinement phase, until the cumulative delta-modularity change across all vertices reaches a specified threshold. Then, in the refinement phase, the each vertex starts in a singleton community, and community memberships are updated similarly to the local-moving phase, with vertices changing communities within their bounds. These community memberships are then counted and renumbered. In the aggregation phase, community vertices in a CSR are first obtained. This is used to create the super-vertex graph stored in a Weighted Holey CSR with degree, which is used as input graph in subsequent passes.

Algorithm 1 GVE-Leiden: Our parallel Leiden algorithm.

▷ G: Input graph
▷ C: Community membership of each vertex
▷ G': Input/super-vertex graph
▷ C': Community membership of each vertex in G'
▷ K': Total edge weight of each vertex
▷ Σ': Total edge weight of each community
□ l_i, l_j: Number of iterations performed (per pass)
□ l_p: Number of passes performed
□ τ: Per iteration tolerance
□ τ_{agg}: Aggregation tolerance

1: **function** LEIDEN(G)
2: Vertex membership: $C \leftarrow [0..|V|)$; $G' \leftarrow G$
3: **for all** $l_p \in [0..\text{MAX\_PASSES})$ **do**
4: $\Sigma' \leftarrow K' \leftarrow vertexWeights(G')$; $C' \leftarrow [0..|V'|)$
5: $l_i \leftarrow leidenMove(G', C', K', \Sigma', \tau)$ ▷ Alg. 2
6: $C'_B \leftarrow C'$; $C' \leftarrow [0..|V'|)$; $\Sigma' \leftarrow K'$
7: $l_j \leftarrow leidenRefine(G', C'_B, C', K', \Sigma')$ ▷ Alg. 3
8: **if** $l_i + l_j \leq 1$ **then break** ▷ Globally converged?
9: $|\Gamma|, |\Gamma_{old}| \leftarrow$ Number of communities in C, C'
10: **if** $|\Gamma|/|\Gamma_{old}| > \tau_{agg}$ **then break** ▷ Low shrink?
11: $C' \leftarrow$ Renumber communities in C'
12: $C \leftarrow$ Lookup dendrogram using C to C'
13: $G' \leftarrow leidenAggregate(G', C')$ ▷ Alg. 4
14: $C' \leftarrow$ Map C' to C'_B ▷ Use move-based membership
15: $\tau \leftarrow \tau/\text{TOLERANCE\_DROP}$ ▷ Threshold scaling
16: $C \leftarrow$ Lookup dendrogram using C to C'
17: **return** C

4.2.3 Refinement phase of GVE-Leiden. The pseudocode for the refinement phase of GVE-Leiden is presented in Algorithm 2. This is similar to the local-moving phase, but utilizes the obtained community membership of each vertex as a *community bound*, where

Algorithm 2 Local-moving phase of GVE-Leiden.

▷ G': Input/super-vertex graph
▷ C': Community membership of each vertex
▷ K': Total edge weight of each vertex
▷ Σ': Total edge weight of each community
□ H_t: Collision-free per-thread hashtable
□ l_i: Number of iterations performed
□ τ: Per iteration tolerance

1: **function** LEIDENMOVE($G', C', K', \Sigma', \tau$)
2: Mark all vertices in G' as unprocessed
3: **for all** $l_i \in [0..\text{MAX\_ITERATIONS})$ **do**
4: Total delta-modularity per iteration: $\Delta Q \leftarrow 0$
5: **for all** unprocessed $i \in V'$ **in parallel do**
6: Mark i as processed (prune)
7: $H_t \leftarrow scanCommunities(\{\}, G', C', i, false)$
8: ▷ Use H_t, K', Σ' to choose best community
9: $c^* \leftarrow$ Best community linked to i in G'
10: $\delta Q^* \leftarrow$ Delta-modularity of moving i to c^*
11: **if** $c^* = C'[i]$ **then continue**
12: $\Sigma'[C'[i]] -= K'[i]$; $\Sigma'[c^*] += K'[i]$ **atomically**
13: $C'[i] \leftarrow c^*$; $\Delta Q \leftarrow \Delta Q + \delta Q^*$
14: Mark neighbors of i as unprocessed
15: **if** $\Delta Q \leq \tau$ **then break** ▷ Locally converged?
16: **return** l_i

17: **function** SCANCOMMUNITIES($H_t, G', C', i, self$)
18: **for all** $(j, w) \in G'.edges(i)$ **do**
19: **if not** $self$ **and** $i = j$ **then continue**
20: $H_t[C'[j]] \leftarrow H_t[C'[j]] + w$
21: **return** H_t

Algorithm 3 Refinement phase of GVE-Leiden.

▷ G': Input/super-vertex graph
▷ C': Community membership of each vertex
▷ K': Total edge weight of each vertex
▷ Σ': Total edge weight of each community
□ H_t: Collision-free per-thread hashtable

1: **function** LEIDENREFINE($G', C'_B, C', K', \Sigma'$)
2: **for all** $i \in V'$ **in parallel do**
3: $c \leftarrow C'[i]$
4: **if** $\Sigma'[c] \neq K'[i]$ **then continue**
5: $H_t \leftarrow scanBounded(\{\}, G', C'_B, C', i, false)$
6: ▷ Use H_t, K', Σ' to choose best community
7: $c^* \leftarrow$ Best community linked to i in G' within C'_B
8: $\delta Q^* \leftarrow$ Delta-modularity of moving i to c^*
9: **if** $c^* = c$ **then continue**
10: **if** $atomicCAS(\Sigma'[c], K'[i], 0) = K'[i]$ **then**
11: $\Sigma'[c^*] += K'[i]$ **atomically ;** $C'[i] \leftarrow c^*$

12: **function** SCANBOUNDED($H_t, G', C'_B, C', i, self$)
13: **for all** $(j, w) \in G'.edges(i)$ **do**
14: **if not** $self$ **and** $i = j$ **then continue**
15: **if** $C'_B[i] \neq C'_B[j]$ **then continue**
16: $H_t[C'[j]] \leftarrow H_t[C'[j]] + w$
17: **return** H_t

18: **function** ATOMICCAS($pointer, old, new$)
19: ▷ Perform the following atomically
20: **if** $pointer = old$ **then** $pointer \leftarrow new$ **; return** old
21: **else return** $pointer$

each vertex must choose to join the community of another vertex within its community bound. At the start of the refinement phase, the community membership of each vertex is reset, such that each vertex belongs to its own community. Here, the `leidenRefine()` function takes the current graph G', the community bound of each vertex C'_B, the initial community membership C' of each vertex, the total edge weight of each vertex K', the initial total edge weight of each community Σ', and the current per iteration tolerance τ as input, and returns the number of iterations performed l_j.

Lines 2-11 represent the core of the refinement phase. In the refinement phase, we perform, what is called the constrained merge procedure [30]. The idea here is to allow vertices, within each community bound, to form sub-communities by only allowing isolated vertices (i.e., vertices belonging to their own community) to change their community membership. This procedure splits any internally-disconnected communities identified during the local-moving phase, and prevents the formation of any new disconnected communities. Here, for each isolated vertex i (line 4), we scan communities connected to i within the *same community bound* - excluding self (line 5), evaluate the best community $c*$ to move i to (line 7), and compute the delta-modularity of moving i to $c*$ (line 8). If a better community was found, we attempt to update the community membership of i if it is still isolated (lines 10-11).

Algorithm 4 Aggregation phase of GVE-Leiden.

▷ G': Input/super-vertex graph
▷ C': Community membership of each vertex
□ $G'_{C'}$: Community vertices (CSR)
□ G'': Super-vertex graph (weighted CSR)
□ $*.offsets$: Offsets array of a CSR graph
□ H_t: Collision-free per-thread hashtable

1: **function** LEIDENAGGREGATE(G', C')
2: ▷ Obtain vertices belonging to each community
3: $G'_{C'}.offsets \leftarrow countCommunityVertices(G', C')$
4: $G'_{C'}.offsets \leftarrow exclusiveScan(G'_{C'}.offsets)$
5: **for all** $i \in V'$ **in parallel do**
6: Add edge $(C'[i], i)$ to CSR $G'_{C'}$ **atomically**
7: ▷ Obtain super-vertex graph
8: $G''.offsets \leftarrow communityTotalDegree(G', C')$
9: $G''.offsets \leftarrow exclusiveScan(G''.offsets)$
10: $|\Gamma| \leftarrow$ Number of communities in C'
11: **for all** $c \in [0, |\Gamma|)$ **in parallel do**
12: $H_t \leftarrow \{\}$
13: **for all** $i \in G'_{C'}.edges(c)$ **do**
14: $H_t \leftarrow scanCommunities(H_t, G', C', i, true)$
15: **for all** $(d, w) \in H_t$ **do**
16: Add edge (c, d, w) to CSR G'' **atomically**
17: **return** G''

4.2.4 Aggregation phase of GVE-Leiden. Finally, we show the psuedocode for the aggregation phase in Algorithm 4, where communities are aggregated into super-vertices in preparation for the next pass of the Leiden algorithm (which operates on the super-vertex graph). Here, the `leidenAggregate()` function takes the current graph G' and the community membership C' as input, and returns the super-vertex graph G''.

In lines 3-4, the offsets array for the community vertices CSR $G'_{C'}.offsets$ is obtained. This is achieved by initially counting the number of vertices in each community using `countCommunityVertices()` and subsequently performing an exclusive scan on the array. In lines 5-6, a parallel iteration over all vertices is performed to atomically populate vertices belonging to each community into the community graph CSR $G'_{C'}$. Following this, the offsets array for the super-vertex graph CSR is obtained by overestimating the degree of each super-vertex. This involves calculating the total degree of each community with `communityTotalDegree()` and performing an exclusive scan on the array (lines 8-9). As a result, the super-vertex graph CSR becomes holey, featuring gaps between the edges and weights arrays of each super-vertex in the CSR.

Then, in lines 11-16, a parallel iteration over all communities $c \in [0, |\Gamma|)$ is performed. For each vertex i belonging to community c, all communities d (with associated edge weight w), linked to i as defined by `scanCommunities()` in Algorithm 2, are added to the per-thread hashtable H_t. Once H_t is populated with all communities (and associated weights) linked to community c, these are atomically added as edges to super-vertex c in the super-vertex graph G''. Finally, in line 17, we return the super-vertex graph G''.

5 EVALUATION

5.1 Experimental Setup

5.1.1 System used. We employ a server with two Intel Xeon Gold 6226R CPUs, each featuring 16 cores running at 2.90 GHz. Each core is equipped with a 1 MB L1 cache, a 16 MB L2 cache, and a 22 MB shared L3 cache. The system is configured with 376 GB RAM and set up with CentOS Stream 8. For our GPU experiments, we use a system with an NVIDIA A100 GPU (108 SMs, 64 CUDA cores per SM, 80 GB global memory, 1935 GB/s bandwidth, 164 KB shared memory per SM) and an AMD EPYC-7742 processor (64 cores, 2.25 GHz). The server has 512 GB DDR4 RAM and runs Ubuntu 20.04.

5.1.2 Configuration. We use 32-bit integers for vertex ids and 32-bit float for edge weights, but use 64-bit floats for computations and hashtable values. We utilize 64 threads to match the number of cores available on the system (unless specified). For compilation, we use GCC 8.5 and OpenMP 4.5 on the CPU-based system, and GCC 9.4, OpenMP 5.0, and CUDA 11.4 on the GPU-based system.

5.1.3 Dataset. The graphs used in our experiments are given in Table 2. These are sourced from the SuiteSparse Matrix Collection [14]. In the graphs, number of vertices vary from 3.07 to 214 million, and number of edges vary from 25.4 million to 3.80 billion. We ensure edges to be undirected and weighted with a default of 1.

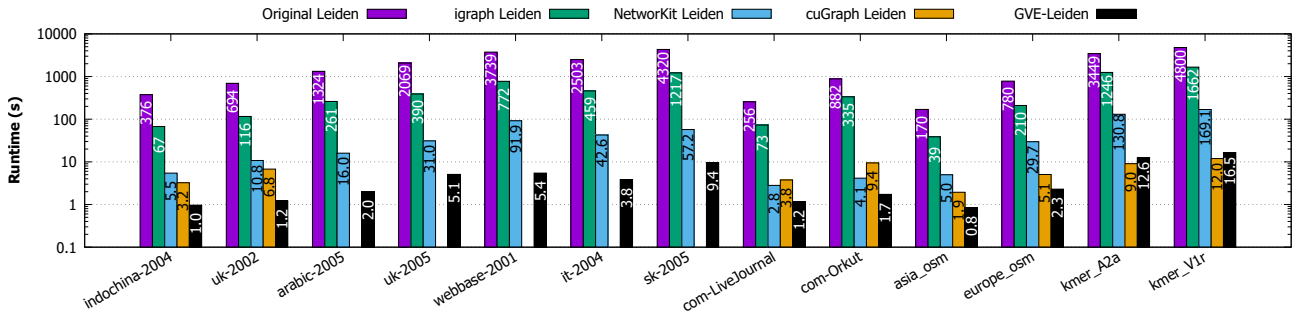
Table 2: List of 13 graphs obtained SuiteSparse Matrix Collection [14] (directed graphs are marked with $*$). Here, $|V|$ is the number of vertices, $|E|$ is the number of edges (after adding reverse edges), D_{avg} is the average degree, and $|\Gamma|$ is the number of communities obtained with *GVE-Leiden*.

Graph	$\|V\|$	$\|E\|$	D_{avg}	$\|\Gamma\|$
Web Graphs (LAW)				
indochina-2004*	7.41M	341M	41.0	2.68K
uk-2002*	18.5M	567M	16.1	41.8K
arabic-2005*	22.7M	1.21B	28.2	2.92K
uk-2005*	39.5M	1.73B	23.7	18.2K
webbase-2001*	118M	1.89B	8.6	2.94M
it-2004*	41.3M	2.19B	27.9	4.05K
sk-2005*	50.6M	3.80B	38.5	2.67K
Social Networks (SNAP)				
com-LiveJournal	4.00M	69.4M	17.4	3.09K
com-Orkut	3.07M	234M	76.2	36
Road Networks (DIMACS10)				
asia_osm	12.0M	25.4M	2.1	2.70K
europe_osm	50.9M	108M	2.1	6.13K
Protein k-mer Graphs (GenBank)				
kmer_A2a	171M	361M	2.1	21.1K
kmer_V1r	214M	465M	2.2	10.5K

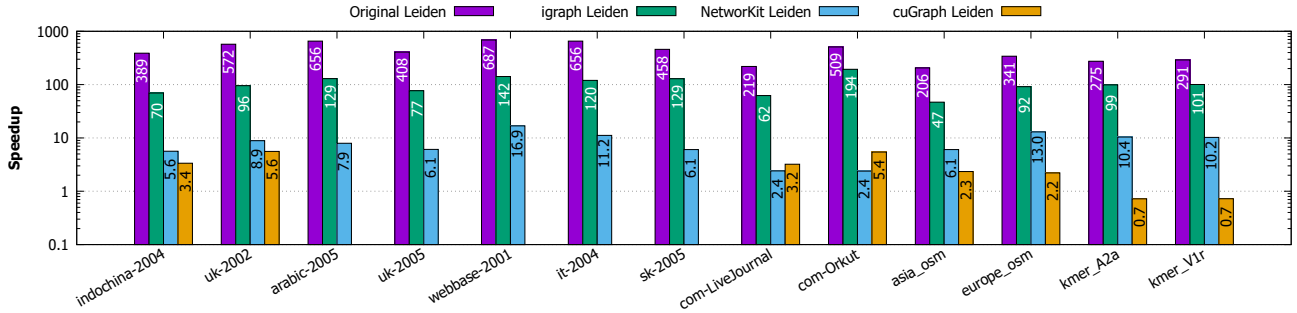
5.2 Comparing Performance of GVE-Leiden

We now compare the performance of GVE-Leiden with the original Leiden [30], igraph Leiden [5], NetworKit Leiden [26], and cuGraph Leiden [13]. The original Leiden and igraph Leiden are sequential implementations, while NetworKit Leiden and GVE-Leiden are parallel multicore implementations of the Leiden algorithm. On the other hand, cuGraph Leiden is a GPU implementation of Leiden, and is run on our GPU-based system. For the original Leiden, we use a C++ program to initialize a `ModularityVertexPartition` upon the loaded graph, and invoke `optimise_partition()` to obtain the community membership of each vertex in the graph. On graphs with a large number of edges, such as *webbase-2001* and *sk-2005*, using `ModularityVertexPartition` introduces disconnected communities due to issues with numerical precision (i.e. the improvement of separating two disconnected parts may be positive, but due to the enormous weight, this may effectively be near 0) [28]. For such graphs, we instead use `RBConfigurationVertexPartition`, which uses unscaled improvements to modularity (i.e. they do not scale with the total weight). For igraph Leiden, we use `igraph_community_leiden()` with a resolution of $1/2|E|$, a beta of 0.01, and request the algorithm to run until convergence. For NetworKit Leiden, we write a Python script to call `ParallelLeiden()`, while limiting the number of passes to 10. For cuGraph Leiden, we write a Python script to configure the Rapids Memory Manager (RMM) to use a pool allocator for fast memory allocations and run `cugraph.leiden()` on the loaded graph. For each graph, we measure the runtime of each implementation and the modularity of the communities obtained, five times, for averaging. With cuGraph, we disregard the runtime of the first run to ensure subsequent measurements only reflect RMM's pool usage without CUDA memory allocation overhead. We save the obtained community membership vector to a file and later count the disconnected communities using an algorithm given in our extended report [22]. In all instances, we use modularity as the quality function to optimize for.

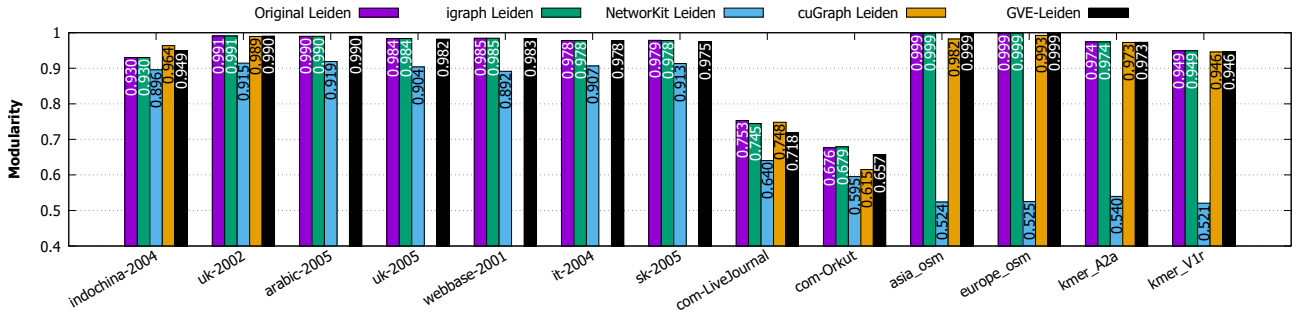
Figure 6(a) shows the runtimes of the original Leiden, igraph Leiden, NetworKit Leiden, cuGraph Leiden, and GVE-Leiden on each graph in the dataset. cuGraph Leiden fails to run on the *arabic-2005*, *uk-2005*, *webbase-2001*, *it-2004*, and *sk-2005* graphs due to out of memory issues. On the *sk-2005* graph, GVE-Leiden finds communities in 9.4 seconds, and thus achieve a processing rate of 403 million edges/s. Figure 6(b) shows the speedup of GVE-Leiden with respect to each implementation mentioned above. GVE-Leiden is on average 436×, 104×, 8.2×, and 3.0× faster than the original Leiden, igraph Leiden, NetworKit Leiden, and cuGraph Leiden respectively. Figure 6(c) shows the modularity of communities obtained with each implementation. GVE-Leiden on average obtains 0.3% lower modularity than the original Leiden and igraph Leiden, 25% higher modularity than NetworKit Leiden (especially on road networks and protein k-mer graphs), and 3.5% higher modularity that cuGraph Leiden (primarily due to cuGraph Leiden's inability to run on well-clusterable graphs). Finally, Figure 6(d) shows the fraction of disconnected communities obtained with each implementation. Here, the absence of bars indicates the absence of disconnected communities. Communities identified by NetworKit Leiden and cuGraph Leiden have on average 1.5×10^{-2} and 6.6×10^{-5} fraction of disconnected communities, respectively, while none of the communities identified by the original Leiden, igraph Leiden, and GVE-Leiden are internally-disconnected. As the Leiden algorithm guarantees the absence of disconnected communities [30], those observed with NetworKit Leiden and cuGraph Leiden are likely due to implementation issues.



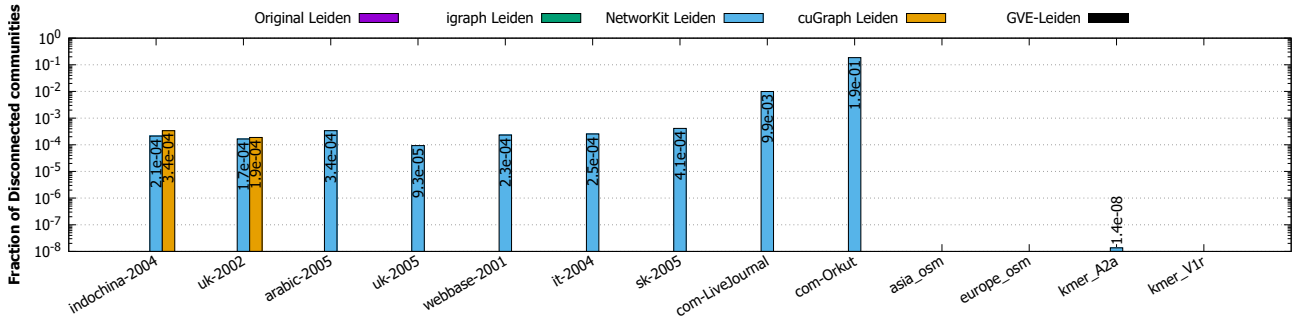
(a) Runtime in seconds (logarithmic scale) with *Original Leiden*, *igraph Leiden*, *NetworKit Leiden*, *cuGraph Leiden*, and *GVE-Leiden*



(b) Speedup of *GVE-Leiden* (logarithmic scale) with respect to *Original Leiden*, *igraph Leiden*, *NetworKit Leiden*, and *cuGraph Leiden*.

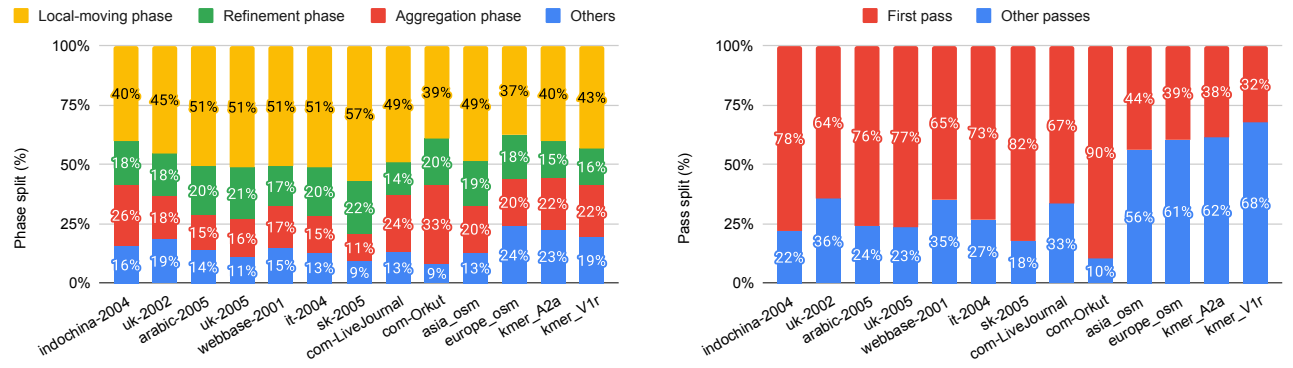


(c) Modularity of communities obtained with *Original Leiden*, *igraph Leiden*, *NetworKit Leiden*, *cuGraph Leiden*, and *GVE-Leiden*.



(d) Fraction of disconnected communities (logarithmic scale) with *Original Leiden*, *igraph Leiden*, *NetworKit Leiden*, *cuGraph Leiden*, and *GVE-Leiden*.

Figure 6: Runtime in seconds (log-scale), speedup (log-scale), modularity, and fraction of disconnected communities (log-scale) with *Original Leiden*, *igraph Leiden*, *NetworKit Leiden*, *cuGraph Leiden*, and *GVE-Leiden* for each graph in the dataset.



(a) Phase split (b) Pass split

Figure 7: Phase split of *GVE-Leiden* shown on the left, and pass split shown on the right for each graph in the dataset.

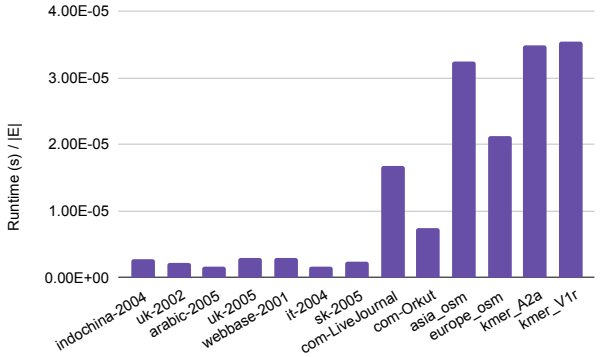


Figure 8: Runtime $/|E|$ factor with *GVE-Leiden* for each graph.

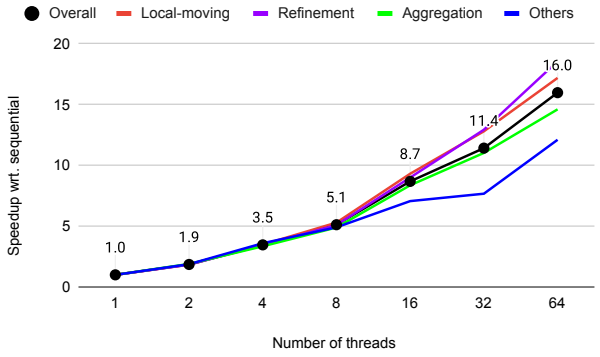


Figure 9: Overall speedup of *GVE-Leiden*, and its various phases (local-moving, refinement, aggregation, others), with increasing number of threads (in multiples of 2).

5.3 Analyzing Performance of GVE-Leiden

We now analyze the performance of GVE-Leiden. The phase-wise and pass-wise split of GVE-Leiden is shown in Figures 7(a) and 7(b) respectively. Figure 7(a) reveals that GVE-Leiden devotes a significant portion of its runtime to the local-moving and refinement phases on *web graphs*, *road networks*, and *protein k-mer graphs*, while it dedicates majority of its runtime in the aggregation phase on *social networks*. The pass-wise split, shown in Figure 7(b), indicates that the first pass is time-intensive for high-degree graphs (*web graphs* and *social networks*), while subsequent passes take precedence in execution time on low-degree graphs.

On average, GVE-Leiden spends 46% of its runtime in the local-moving phase, 19% in the refinement phase, 20% in the aggregation phase, and 15% in other steps (initialization, renumbering communities, dendrogram lookup, and resetting communities). Further, 63% of the runtime is consumed by the first pass of the algorithm, which is computationally demanding due to the size of the original graph (subsequent passes operate on super-vertex graphs). We also observe that graphs with lower average degree (*road networks* and *protein k-mer graphs*) and those with poor community structure (e.g., `com-LiveJournal` and `com-Orkut`) exhibit a higher runtime/$|E|$ factor, as shown in Figure 8.

5.4 Strong Scaling of GVE-Leiden

Finally, we assess the strong scaling performance of GVE-Leiden. In this analysis, we vary the number of threads from 1 to 64 in multiples of 2 for each input graph, and measure the total time taken for GVE-Leiden to identify communities, encompassing its phase splits (local-moving, refinement, aggregation, and others), repeated five times for averaging. The results are shown in Figure 9. With 32 threads, GVE-Leiden achieves an average speedup of 11.4× compared to a single-threaded execution, indicating a performance increase of 1.6× for every doubling of threads. Nevertheless, scalability is restricted due to the sequential nature of steps/phases in the algorithm. At 64 threads, GVE-Leiden is affected by NUMA effects, resulting in a speedup of only 16.0×.

5.5 Indirect Comparison with State-of-the-art

We now perform an indirect comparison of our multicore Leiden algorithm implementation's performance against other state-of-the-art implementations. Please treat the reported speedups as rough estimates. Hu et al. [12] introduce ParLeiden, a parallel Leiden

implementation for distributed environments, which uses thread locks and efficient buffers, to resolve community joining conflicts and reduce communication overheads. They refer to their single node version of ParLeiden as ParLeiden-S, and their distributed version as ParLeiden-D. On a cluster with 8 nodes, with each node being equipped with a 48 core CPU, Hu et al. observe a speedup of 12.3×, 9.9×, and 1.32× for ParLeiden-S, ParLeiden-D, and a baseline Leiden implemented on KatanaGraph, on the *com-LiveJournal* graph, with respect to original Leiden [30] (refer to Table 2 in their paper [12]). In contrast, on the same graph, we observe a speedup of 219× relative to original Leiden. Consequently, GVE-Leiden outperforms ParLeiden-S, ParLeiden-D, and KatanaGraph Leiden by approximately 18×, 22×, and 166× respectively.

6 CONCLUSION

In conclusion, this study addresses the design of the most optimized multicore implementation of the Leiden algorithm [30], to the best of our knowledge. Here, we extend optimizations from our implementation of the Louvain algorithm [23], and use a greedy refinement phase where vertices greedily optimize for delta-modularity within their community bounds, which we observe, offers both better performance and quality than a randomized approach. On a system with two 16-core Intel Xeon Gold 6226R processors, our implementation, referred to as GVE-Leiden, attains a processing rate of $403M$ edges per second on a $3.8B$ edge graph. It outperforms the original Leiden implementation, igraph Leiden, NetworKit Leiden, and cuGraph Leiden (run on an NVIDIA A100 GPU) by 436×, 104×, 8.2×, and 3.0× respectively. GVE-Leiden identifies communities of equivalent quality to the first two implementations, and 25% / 3.5% higher quality than NetworKit / cuGraph. Doubling the threads results in a performance scaling of 1.6× for GVE-Leiden.

ACKNOWLEDGMENTS

We extend our sincere thanks to Vincent Traag for his invaluable assistance in debugging our Leiden implementation, identifying issues related to discovering disconnected communities, and aiding in benchmarking the original Leiden implementation. We also thank Fabian Nguyen for directing us to NetworKit Leiden, and to Chuck Hastings and Rick Ratzel for their guidance in properly benchmarking cuGraph Leiden. This work was partially supported by a grant from the Department of Science and Technology (DST), India, under the National Supercomputing Mission (NSM) R&D in Exascale initiative vide Ref. No: DST/NSM/R&D_Exascale/2021/16. To promote reproducibility of the results, our computational artifact is available in the Zenodo repository [24].

REFERENCES

[1] A. Aldabobi, A. Sharieh, and R. Jabri. 2022. An improved Louvain algorithm based on Node importance for Community detection. *Journal of Theoretical and Applied Information Technology* 100, 23 (2022), 1–14.

[2] V. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (Oct 2008), P10008.

[3] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner. 2007. On modularity clustering. *IEEE transactions on knowledge and data engineering* 20, 2 (2007), 172–188.

[4] C. Cheong, H. Huynh, D. Lo, and R. Goh. 2013. Hierarchical Parallel Algorithm for Modularity-Based Community Detection Using GPUs. In *Proceedings of the 19th International Conference on Parallel Processing* (Aachen, Germany) *(Euro-Par'13)*. Springer-Verlag, Berlin, Heidelberg, 775–787.

[5] G. Csardi, T. Nepusz, et al. 2006. The igraph software package for complex network research. *InterJournal, complex systems* 1695, 5 (2006), 1–9.

[6] M. Fazlali, E. Moradi, and H. Malazi. 2017. Adaptive parallel Louvain community detection on a multicore platform. *Microprocessors and microsystems* 54 (Oct 2017), 26–34.

[7] O. Gach and J. Hao. 2014. Improving the Louvain algorithm for community detection with modularity maximization. In *Artificial Evolution: 11th International Conference, Evolution Artificielle, EA , Bordeaux, France, October 21-23, . Revised Selected Papers 11*. Springer, Springer, Bordeaux, France, 145–156.

[8] S. Ghosh, M. Halappanavar, A. Tumeo, and A. Kalyanarainan. 2019. Scaling and quality of modularity optimization methods for graph clustering. In *IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[9] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarria-Miranda, A. Khan, and A. Gebremedhin. 2018. Distributed louvain algorithm for graph community detection. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Vancouver, British Columbia, Canada, 885–895.

[10] S. Gregory. 2010. Finding overlapping communities in networks by label propagation. *New Journal of Physics* 12 (10 2010), 103018. Issue 10.

[11] M. Halappanavar, H. Lu, A. Kalyanaraman, and A. Tumeo. 2017. Scalable static and dynamic community detection using Grappolo. In *IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA USA, 1–6.

[12] Y. Hu, J. Wang, C. Zhao, Y. Liu, C. Chen, X. Cong, and C. Li. [n. d.]. ParLeiden: Boosting Parallelism of Distributed Leiden Algorithm on Large-scale Graphs. ([n. d.]).

[13] S. Kang, C. Hastings, J. Eaton, and B. Rees. 2023. cuGraph C++ primitives: vertex/edge-centric building blocks for parallel graph computing. In *IEEE International Parallel and Distributed Processing Symposium Workshops*. 226–229.

[14] S. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. Davis, M. Henderson, Y. Hu, and R. Sandstrom. 2019. The SuiteSparse matrix collection website interface. *The Journal of Open Source Software* 4, 35 (Mar 2019), 1244.

[15] A. Lancichinetti and S. Fortunato. 2009. Community detection algorithms: a comparative analysis. *Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics* 80, 5 Pt 2 (Nov 2009), 056117.

[16] H. Lu, M. Halappanavar, and A. Kalyanaraman. 2015. Parallel heuristics for scalable community detection. *Parallel computing* 47 (Aug 2015), 19–37.

[17] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo. 2017. Community detection on the GPU. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Orlando, Florida, USA, 625–634.

[18] M. Newman. 2006. Finding community structure in networks using the eigenvectors of matrices. *Physical review E* 74, 3 (2006), 036104.

[19] Fabian Nguyen. [n. d.]. *Leiden-Based Parallel Community Detection*. Bachelor's Thesis. Karlsruhe Institute of Technology, 2021 (zitiert auf S. 31).

[20] R. Rotta and A. Noack. 2011. Multilevel local search algorithms for modularity clustering. *Journal of Experimental Algorithmics (JEA)* 16 (2011), 2–1.

[21] S. Ryu and D. Kim. 2016. Quick community detection of big graph data using modified louvain algorithm. In *IEEE 18th International Conference on High Performance Computing and Communications (HPCC)*. IEEE, Sydney, NSW, 1442–1445.

[22] S. Sahu. 2023. GVE-Leiden: Fast Leiden Algorithm for Community Detection in Shared Memory Setting. *arXiv preprint arXiv:2312.13936* (2023).

[23] S. Sahu. 2023. GVE-Louvain: Fast Louvain Algorithm for Community Detection in Shared Memory Setting. *arXiv preprint arXiv:2312.04876* (2023).

[24] S. Sahu, K. Kothapalli, and D.S. Banerjee. 2024. *Artifact of the paper: Fast Leiden Algorithm for Community Detection in Shared Memory Setting*. https://doi.org/10.XXXX/zenodo.XXXXXXXX

[25] J. Shi, L. Dhulipala, D. Eisenstat, J. Łącki, and V. Mirrokni. 2021. Scalable community detection via parallel correlation clustering.

[26] C.L. Staudt, A. Sazonovs, and H. Meyerhenke. 2016. NetworKit: A tool suite for large-scale complex network analysis. *Network Science* 4, 4 (2016), 508–530.

[27] V. Traag. 2015. Faster unfolding of communities: Speeding up the Louvain algorithm. *Physical Review E* 92, 3 (2015), 032801.

[28] V. Traag. 2024. Personal communication. (2024).

[29] V. Traag, P. Dooren, and Y. Nesterov. 2011. Narrow scope for resolution-limit-free community detection. *Physical Review E* 84, 1 (2011), 016114.

[30] V. Traag, L. Waltman, and N. Eck. 2019. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports* 9, 1 (Mar 2019), 5233.

[31] G. Verweij. [n. d.]. *Faster Community Detection Without Loss of Quality: Parallelizing the Leiden Algorithm*. Master's Thesis. Leiden University, 2020.

[32] C. Wickramaarachchi, M. Frincu, P. Small, and V. Prasanna. 2014. Fast parallel algorithm for unfolding of communities in large graphs. In *IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, IEEE, Waltham, MA USA, 1–6.

[33] Y. You, L. Ren, Z. Zhang, K. Zhang, and J. Huang. 2022. Research on improvement of Louvain community detection algorithm. In *2nd International Conference on Artificial Intelligence, Automation, and High-Performance Computing (AIAHPC)*, Vol. 12348. SPIE, Zhuhai, China, 527–531.

[34] J. Zhang, J. Fei, X. Song, and J. Feng. 2021. An improved Louvain algorithm for community detection. *Mathematical Problems in Engineering* 2021 (2021), 1–14.