

# GVE-Leiden: Fast Leiden Algorithm for Community Detection in Shared Memory Setting

Subhajit Sahu

subhajit.sahu@research.iiit.ac.in

IIIT Hyderabad

Hyderabad, Telangana, 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 technical report presents one of the most efficient parallel implementation 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, and NetworKit Leiden by 436 $\times$ , 104 $\times$ , and 8.2 $\times$  respectively - achieving a processing rate of 403M edges/s on a 3.8B edge graph. Compared to GVE-Louvain, our parallel Louvain implementation, GVE-Leiden achieves a total elimination of disconnected communities, with only a 13% increase in runtime. In addition, GVE-Leiden improves performance at an average rate of 1.6 $\times$  for every doubling of threads.

## KEYWORDS

Community detection, Parallel Leiden implementation

## 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. It is an NP-hard problem with applications in topic discovery, protein annotation, recommendation systems, and targeted advertising [11]. The Louvain method [3] is a popular heuristic-based approach for community detection. It 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 [20].

Despite its popularity, the Louvain method has been observed to produce internally-disconnected and badly connected communities. To address these shortcomings, Traag et al. [33] 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 [33].

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 [12]. 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 [21, 34] 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<sup>1</sup>. 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. 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, multi-node implementations, detailed in Table 1. Both direct and indirect comparisons are provided, with further details outlined in Sections 5.2 and A.3, respectively.

**Table 1: Speedup of our multicore implementation of Leiden algorithm compared to other state-of-the-art implementations. Direct comparisons entail running the given implementation on our server, while indirect comparisons (marked with a \*, explained in Section A.3) involve comparing results relative to a common reference.**

| Leiden implementation      | Published | Our Speedup    |
|----------------------------|-----------|----------------|
| Original Leiden [33]       | 2019      | 22 $\times$    |
| igraph Leiden [6]          | 2006      | 50 $\times$    |
| NetworKit Leiden [29]      | 2016      | 20 $\times$    |
| KatanaGraph Leiden [1]     | 2023      | 166 $\times^*$ |
| ParLeiden-S [13]           | 2023      | 22 $\times^*$  |
| ParLeiden-D (8 nodes) [13] | 2023      | 18 $\times^*$  |

<sup>1</sup><https://github.com/puzzlef/leiden-communities-openmp>

## 2 RELATED WORK

The Louvain method, introduced by Blondel et al. [3] 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 [33].

To address these limitations, Traag et al. [33] 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 [33]. Shi et al. [28] 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 [9, 33]. This can be overcome by using an alternative quality function, such as the Constant Potts Model (CPM) [32].

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 [2], attempting local move only on likely vertices [22, 25, 28, 38], early pruning of non-promising candidates [12, 25, 37, 38], moving vertices to a random neighbor community [30], subnetwork refinement [33, 35], multilevel refinement [8, 24, 28], threshold cycling [10], threshold scaling [12, 17, 19], and early termination [10]. A number of parallelization techniques have been also attempted for the Louvain method, that may also be applied to the Leiden algorithm. These include using adaptive parallel thread assignment [7, 18, 19, 27], parallelizing the costly first iteration [36], ordering vertices via graph coloring [12], performing iterations asynchronously [23, 28], and using sort-reduce instead of hashing [5].

A few open source implementations and software packages have been developed for community detection using Leiden algorithm. The original implementation of the Leiden algorithm [33], called `liblabeledalg`, is written in C++ and has a Python interface called `leidenalg`. `NetworkKit` [29] 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 [21] which uses global queues for vertex pruning, and vertex and community locking for updating communities. `igraph` [6] 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.

## 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,  $C : V \rightarrow \Gamma$ , where each vertex  $i \in V$  is assigned a community-id  $c$  from the set of community-ids  $\Gamma$ . 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  $J_{i \rightarrow c} = \{j \mid j \in J_i \text{ and } C_j = c\}$ , the sum of those edge weights as  $K_{i \rightarrow c} = \sum_{j \in J_{i \rightarrow c}} w_{ij}$ , 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  $\Sigma_c = \sum_{(i,j) \in E \text{ and } C_i=c} w_{ij}$  [16].

### 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 [4]. The modularity  $Q$  of identified communities is determined using Equation 1, where  $\delta$  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  $\Delta Q_{i:d \rightarrow c}$ , can be calculated using Equation 2.

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

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

### 3.3 Louvain algorithm

The Louvain method [3] 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  $\Delta Q_{i:C_i \rightarrow C_j}$  (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 [3, 16].

### 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. [33] 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. Once communities have converged, it is guaranteed that all vertices are optimally assigned, and all communities are subset optimal [33]. 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 our optimization techniques, originally designed for the Louvain method [26], 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 [28]. 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 [26].

Our optimizations include utilizing OpenMP’s *dynamic* loop scheduling, capping the number of iterations per pass at 20, employing a tolerance drop rate of 10 (threshold scaling), initiating with a tolerance of 0.01, using an aggregation tolerance of 0.8 to avoid performing aggregations of minimal utility, implementing flag-based vertex pruning (instead of a queue-based one [21]), utilizing parallel prefix sum, and using preallocated CSRs for identifying community vertices and storing the super-vertex graph during aggregation. Additionally, we employ fast collision-free per-thread hashtables, well separated in their memory addresses [26].

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 [33]. 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.

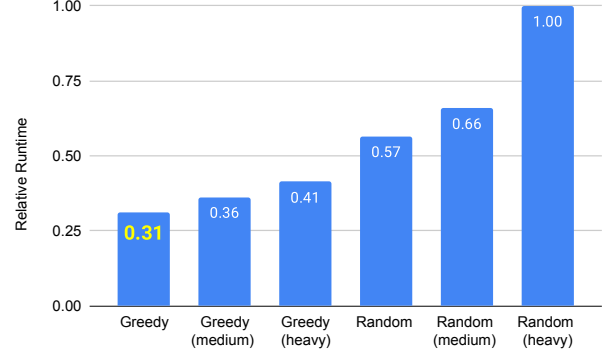


Figure 1: Average relative runtime for the *greedy* and *random* approaches (including *medium* and *heavy* variants) of parallel Leiden algorithm for all graphs in the dataset.

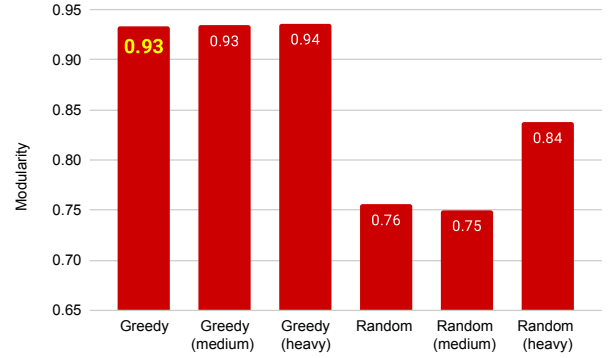


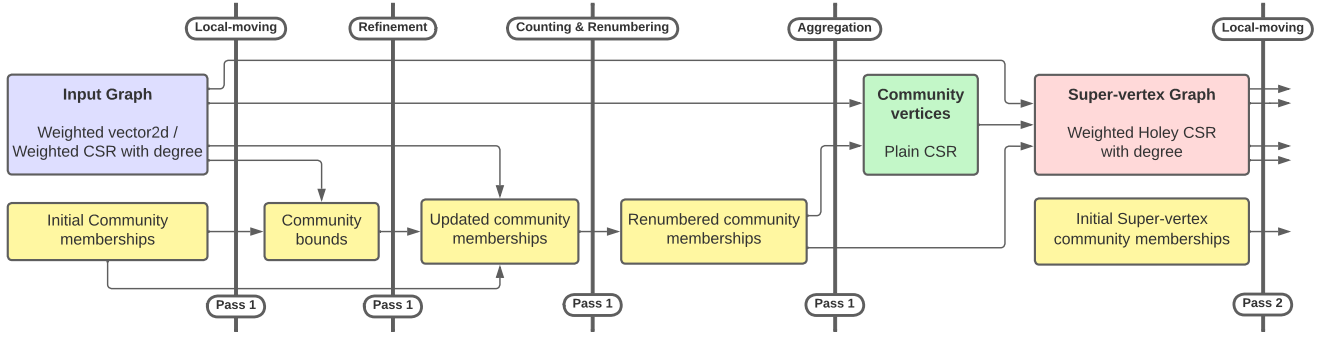
Figure 2: Average modularity for the *greedy* and *random* approaches (including *medium* and *heavy* variants) of parallel Leiden algorithm for all graphs in the dataset.

### 4.2 Our optimized Leiden implementation

We now explain the implementation of GVE-Leiden in Algorithms 1, 2, 3, and 4. A flow diagram illustrating the first pass of GVE-Leiden is shown in Figure 3.

**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  $\Sigma'$ , 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



**Figure 3: 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 Holy CSR with degree. Subsequent passes use a Weighted Holy CSR with degree and initial community memberships for super-vertices from the previous pass as input.**

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  $\Sigma'$ , the iteration tolerance  $\tau$  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  $\Delta Q$ . 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

---

**Algorithm 1** GVE-Leiden: Our parallel Leiden algorithm.

---

```

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

```

---

**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
▷  $\Sigma'$ : Total edge weight of each community
□  $G'_C$ : Community vertices (CSR)
□  $H_t$ : Collision-free per-thread hashtable
□  $l_i$ : Number of iterations performed
□  $\tau$ : Per iteration tolerance

1: function LEIDENMOVE( $G', C', K', \Sigma', \tau$ )
2:   Mark all vertices in  $G'$  as unprocessed
3:   for all  $l_i \in [0..MAX\_ITERATIONS]$  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:        $H_t \leftarrow scanCommunities(\{i\}, G', C', i, false)$ 
8:       ▷ Use  $H_t, K', \Sigma'$  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:       $\Sigma'[C'[i]] - = K'[i]; \Sigma'[c^*] + = K'[i]$  atomic
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  $H_t$ 

```

---

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  $l_i$ .

**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 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  $\Sigma'$ , and the current per iteration tolerance  $\tau$  as input, and returns the number of iterations performed  $l_j$ .

Lines 2-12 represent the core of the refinement phase. In the refinement phase, we perform, what is called the constrained merge procedure [33]. The idea here is to allow vertices, within each community bound, to form sub-communities by only allowing

**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
▷  $\Sigma'$ : Total edge weight of each community
□  $G'_C$ : Community vertices (CSR)
□  $H_t$ : Collision-free per-thread hashtable
□  $\tau$ : Per iteration tolerance

1: function LEIDENREFINE( $G', C'_B, C', K', \Sigma', \tau$ )
2:   for all  $i \in V'$  in parallel do
3:      $c \leftarrow C'[i]$ 
4:     if  $\Sigma'[c] \neq K'[i]$  then continue
5:      $H_t \leftarrow scanBounded(\{i\}, G', C'_B, C', i, false)$ 
6:     ▷ Use  $H_t, K', \Sigma'$  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
12:       $C'[i] \leftarrow c^*$ 

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

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

```

---

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 10-12).

**4.2.4 Aggregation phase of GVE-Leiden.** Finally, we show the pseudocode 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 `countCommunityVert`

**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:   $|\Gamma| \leftarrow \text{Number of communities in } C'$ 
11:  for all  $c \in [0, |\Gamma|)$  in parallel do
12:    if degree of  $c$  in  $G'_{C'}$  = 0 then continue
13:     $H_t \leftarrow \{\}$ 
14:    for all  $i \in G'_{C'}.edges(c)$  do
15:       $H_t \leftarrow scanCommunities(H, G', C', i, true)$ 
16:    for all  $(d, w) \in H_t$  do
17:      Add edge ( $c, d, w$ ) to CSR  $G''$  atomically
18:  return  $G''$ 

```

---

ices() 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 11-17, 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 18, we return the super-vertex graph  $G''$ .

## 5 EVALUATION

### 5.1 Experimental Setup

**5.1.1 System used.** We employ a server equipped with two Intel Xeon Gold 6226R processors, each featuring 16 cores running at a clock speed of 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.

**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 otherwise). For compilation, we use GCC 8.5 and OpenMP 4.5.

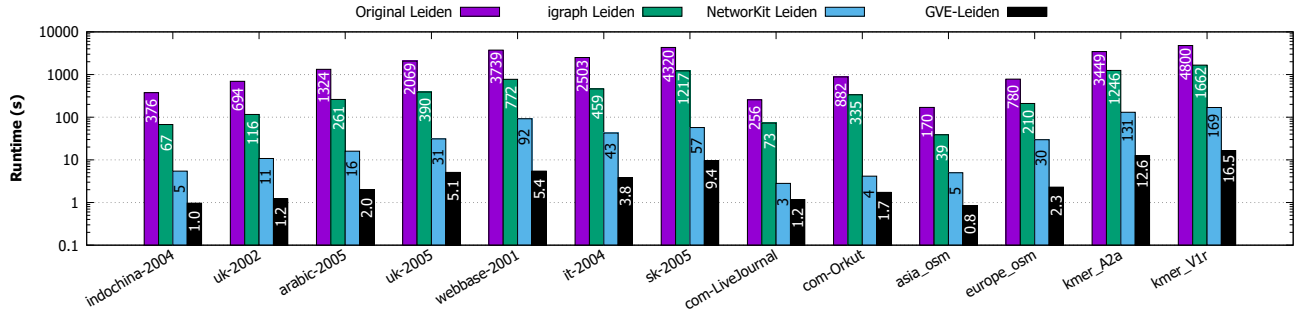
**5.1.3 Dataset.** The graphs used in our experiments are given in Table 2. These are sourced from the SuiteSparse Matrix Collection [14]. In the graphs, number of vertices vary from 3.07 to 214 million, and number of edges vary from 25.4 million to 3.80 billion. We ensure edges to be undirected and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

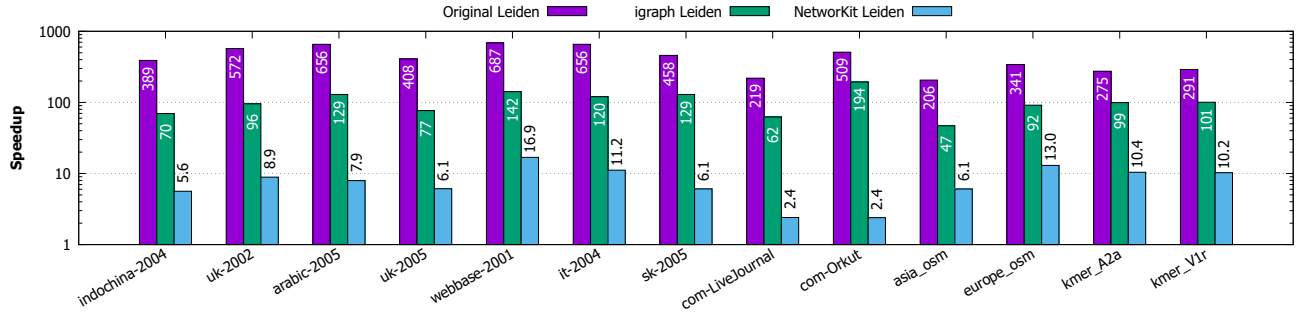
| Graph                                 | $ V $ | $ E $ | $D_{avg}$ | $ \Gamma $ |
|---------------------------------------|-------|-------|-----------|------------|
| <b>Web Graphs (LAW)</b>               |       |       |           |            |
| indochina-2004*                       | 7.41M | 341M  | 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      |
| <b>Social Networks (SNAP)</b>         |       |       |           |            |
| com-LiveJournal                       | 4.00M | 69.4M | 17.4      | 3.09K      |
| com-Orkut                             | 3.07M | 234M  | 76.2      | 36         |
| <b>Road Networks (DIMACS10)</b>       |       |       |           |            |
| asia_osm                              | 12.0M | 25.4M | 2.1       | 2.70K      |
| europa_osm                            | 50.9M | 108M  | 2.1       | 6.13K      |
| <b>Protein k-mer Graphs (GenBank)</b> |       |       |           |            |
| kmer_A2a                              | 171M  | 361M  | 2.1       | 21.1K      |
| kmer_V1r                              | 214M  | 465M  | 2.2       | 10.5K      |

### 5.2 Comparing Performance of GVE-Leiden

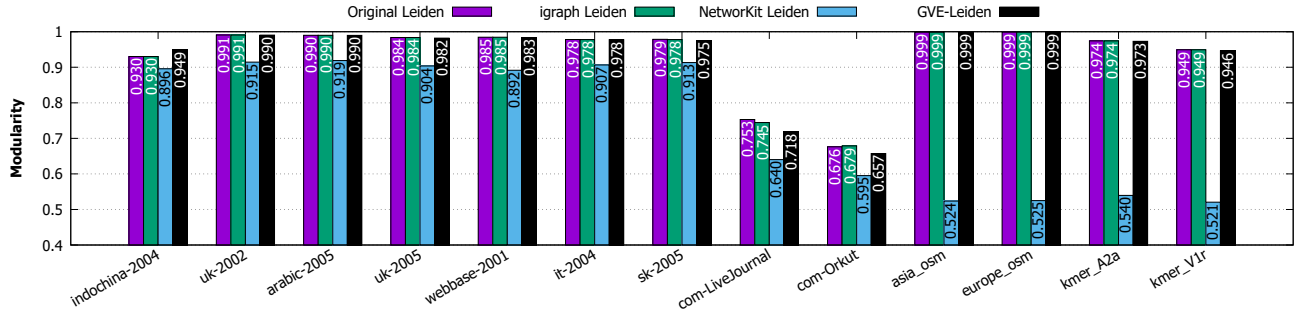
We now compare the performance of GVE-Leiden with the original Leiden [33], igraph Leiden [6], and NetworkKit Leiden [29]. For the original Leiden, we use a C++ program to initialize a `ModularityVertexPartition` upon the loaded graph, and invoke `optimise_partition()` to obtain the community membership 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) [31]. 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 NetworkKit Leiden, we write a



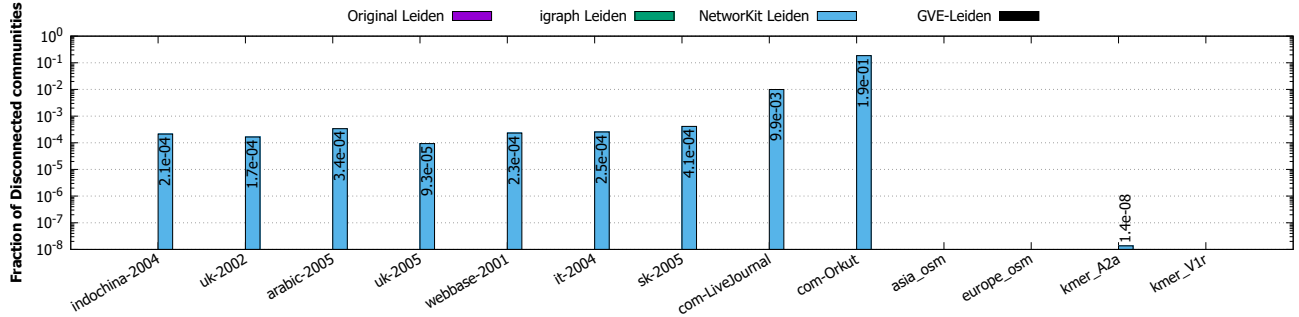
(a) Runtime in seconds (logarithmic scale) with *Original Leiden*, *igraph Leiden*, *NetworkKit Leiden*, and *GVE-Leiden*



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

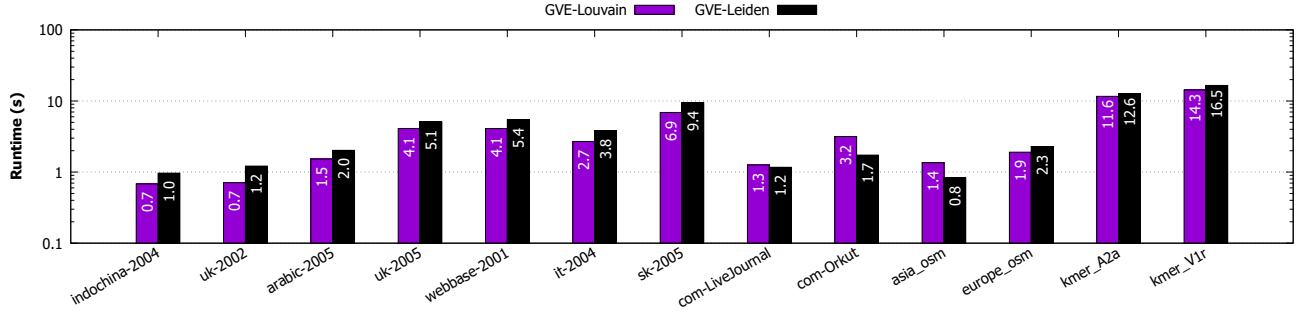


(c) Modularity of communities obtained with *Original Leiden*, *igraph Leiden*, *NetworkKit Leiden*, and *GVE-Leiden*.

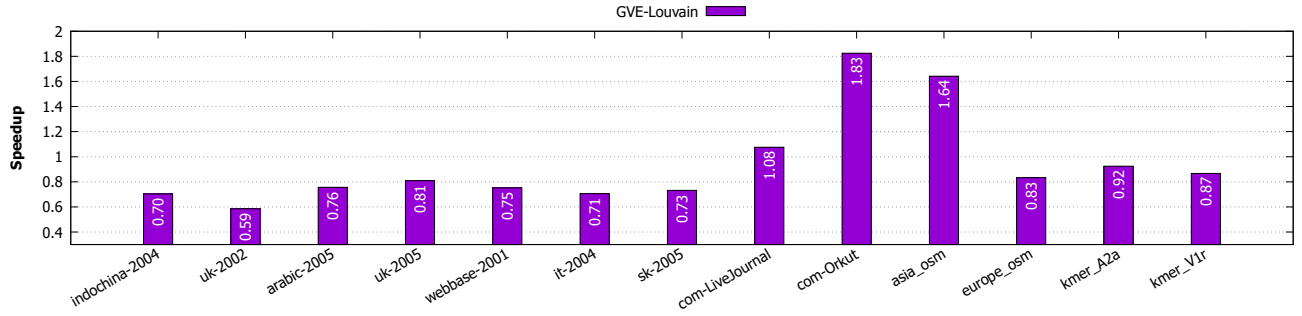


(d) Fraction of disconnected communities (logarithmic scale) with *Original Leiden*, *igraph Leiden*, *NetworkKit Leiden*, and *GVE-Leiden*.

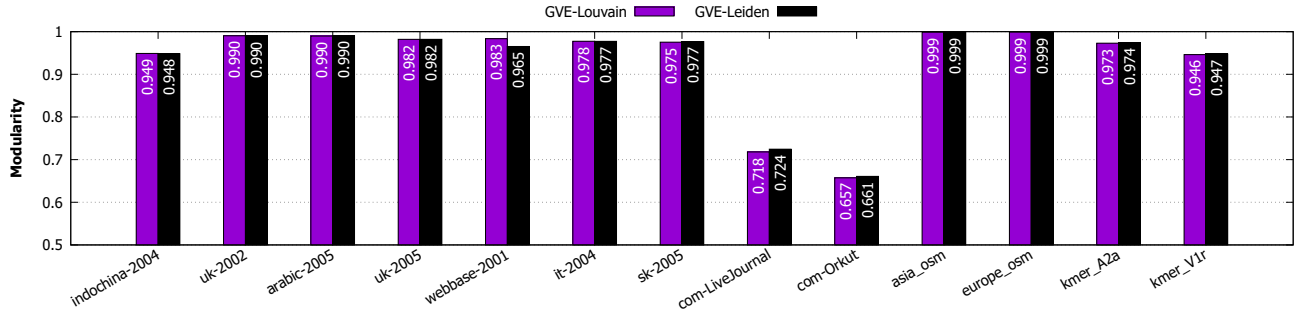
**Figure 4: Runtime in seconds (log-scale), speedup (log-scale), modularity, and fraction of disconnected communities (log-scale) with *Original Leiden*, *igraph Leiden*, *NetworkKit Leiden*, and *GVE-Leiden* for each graph in the dataset.**



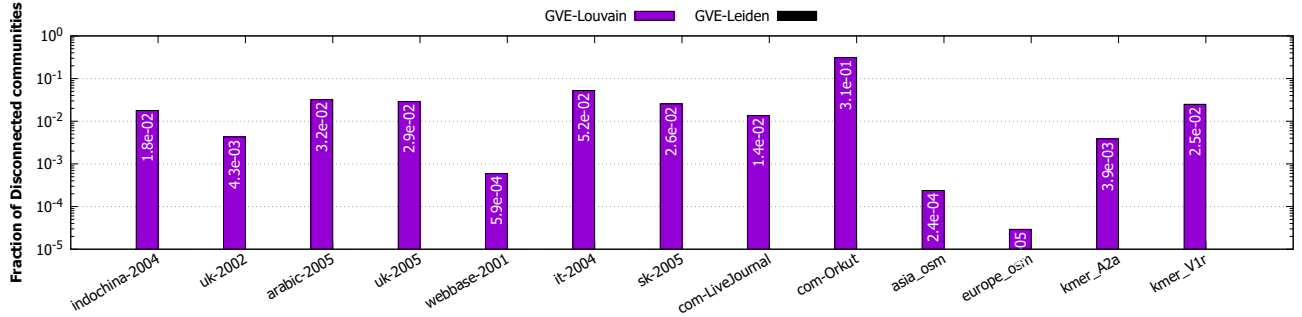
(a) Runtime in seconds (logarithmic scale) with GVE-Louvain and GVE-Leiden



(b) Speedup of GVE-Leiden with respect to GVE-Louvain. GVE-Leiden is generally slower (speedup &lt; 1) because of additional refinement phase.



(c) Modularity of communities obtained with GVE-Louvain and GVE-Leiden.



(d) Fraction of disconnected communities (logarithmic scale) with GVE-Louvain and GVE-Leiden.

**Figure 5: Runtime in seconds (log-scale), speedup, modularity, and fraction of disconnected communities (log-scale) with GVE-Louvain and GVE-Leiden for each graph in the dataset.**



Python script to call `ParallelLeiden()`, while limiting the number of passes to 10. For each graph, we measure the runtime of each implementation and the modularity of the communities obtained, five times, for averaging. We also save the community membership vector to a file and later count the number of disconnected components using Algorithm 5. In all instances, we use modularity as the quality function to optimize for.

Figure 4(a) shows the runtimes of the original Leiden, igraph Leiden, NetworKit Leiden, and GVE-Leiden on each graph in the dataset. 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 4(b) shows the speedup of GVE-Leiden with respect to each implementation mentioned above. GVE-Leiden is on average 436 $\times$ , 104 $\times$ , and 8.2 $\times$  faster than the original Leiden, igraph Leiden, and NetworKit Leiden respectively. Figure 4(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, and 25% higher modularity than NetworKit Leiden (especially on road networks and protein k-mer graphs). Finally, Figure 4(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 have on average  $1.5 \times 10^{-2}$  fraction of disconnected communities, 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 [33], those observed with NetworKit Leiden are likely due to implementation issues.

Next, we compare the performance of GVE-Leiden with GVE-Louvain [26], our parallel implementation of the Louvain method. As above, for each graph in the dataset, we run both algorithms 5 times to minimize measurement noise. Figure 5(a) shows the runtimes of GVE-Louvain and GVE-Leiden on each graph in the dataset. Figure 5(b) shows the speedup of GVE-Leiden with respect to GVE-Louvain. GVE-Leiden is on average 13% slower than GVE-Louvain. This increase in computation time is a trade-off for identifying communities that are not internally-disconnected, as given below. Figure 5(c) shows the modularity of communities obtained with GVE-Louvain and GVE-Leiden. GVE-Leiden on average obtains the same modularity as GVE-Louvain. Finally, Figure 5(d) shows the fraction of internally-disconnected communities obtained. Communities identified by GVE-Louvain on average have 4.0% disconnected communities, while GVE-Leiden has none.

### 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 6(a) and 6(b) respectively. Figure 6(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 6(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 (*road networks* and *protein k-mer 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 7.

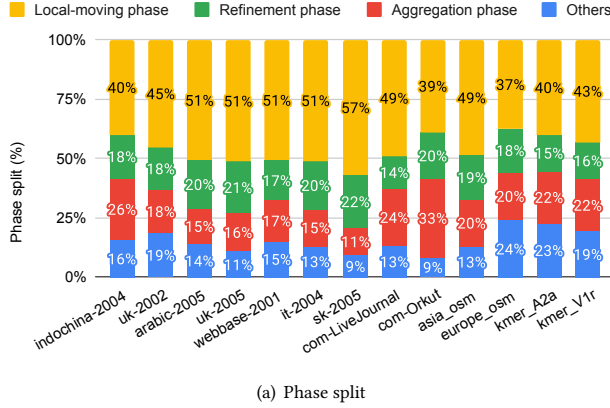
### 5.4 Strong Scaling of GVE-Leiden

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 8. With 32 threads, GVE-Leiden achieves an average speedup of 11.4 $\times$  compared to a single-threaded execution, indicating a performance increase of 1.6 $\times$  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 $\times$ .

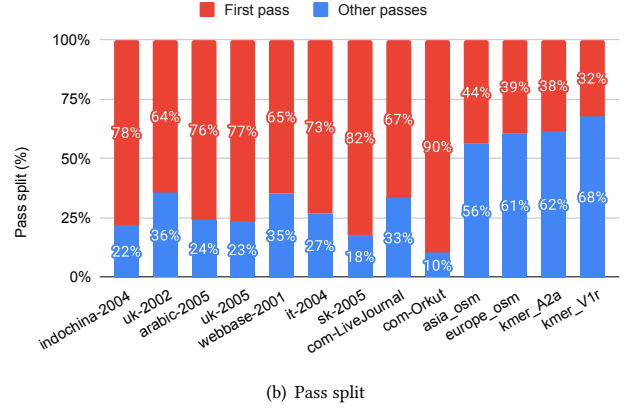
## 6 CONCLUSION

In conclusion, this study addresses the design of the most optimized multicore implementation of the Leiden algorithm [33], to the best of our knowledge. On a system equipped with two 16-core Intel Xeon Gold 6226R processors, our implementation of the Leiden algorithm, referred to as GVE-Leiden, attains a processing rate of 403M edges per second on a 3.8B edge graph. It surpasses the original Leiden implementation, igraph Leiden, and NetworKit Leiden by factors of 436 $\times$ , 104 $\times$ , and 8.2 $\times$  respectively. GVE-Leiden identifies communities of equivalent quality to the first two implementations, and 25% higher quality than NetworKit. Doubling the number of threads results in an average performance scaling of 1.6 $\times$  for GVE-Leiden. In comparison to GVE-Louvain (our parallel Louvain implementation) [26], the original Leiden, igraph Leiden, and NetworKit Leiden, GVE-Leiden completely eliminates internally-disconnected communities.

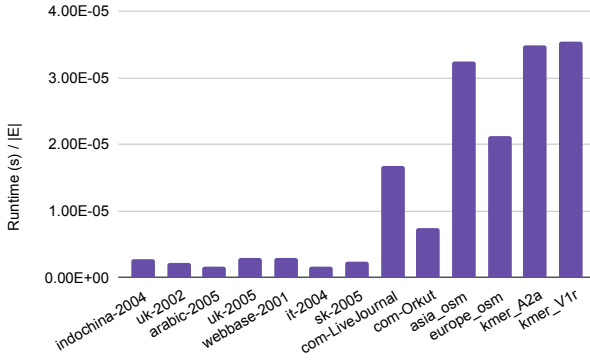
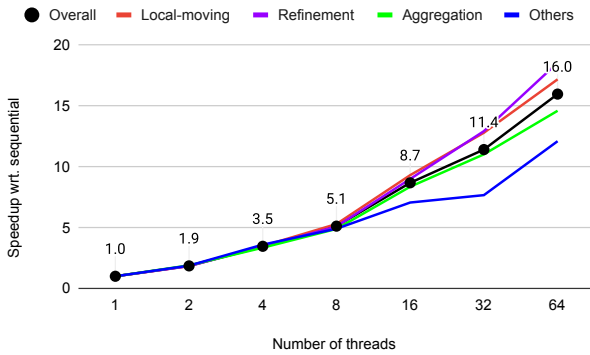
In a previous version of this report, we implemented the refinement phase of the Leiden algorithm utilizing a *constrained move* procedure, which does not guarantee the absence of disconnected communities. In this current version of the report, we have transitioned to employing the *constrained merge* procedure alongside atomics to ensure no internally-disconnected communities. We also addressed issues in measuring disconnected communities for the original Leiden and igraph Leiden, which arose due to the number of vertices in a graph varying between the Matrix Market and the Edgelist formats (which does not have isolated vertices), and used the `RBCConfigurationVertexPartition` with the original Leiden for large graphs (i.e., *webbase-2001* and *sk-2005*).



(a) Phase split



(b) Pass split

Figure 6: Phase split of *GVE-Leiden* shown on the left, and pass split shown on the right for each graph in the dataset.Figure 7: Runtime /|E| factor with *GVE-Leiden* for each graph in the dataset.Figure 8: Overall speedup of *GVE-Leiden*, and its various phases (local-moving, refinement, aggregation, others), with increasing number of threads (in multiples of 2).

## ACKNOWLEDGMENTS

I would like to thank Prof. Kishore Kothapalli, Prof. Dip Sankar Banerjee, Vincent Traag, Geerten Verweij, and Fabian Nguyen for their support.

## REFERENCES

- [1] 2023. KatanaGraph. <https://katanagraph.ai/>
- [2] 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.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] G. Csardi, T. Nepusz, et al. 2006. The igraph software package for complex network research. *InterJournal, complex systems* 1695, 5 (2006), 1–9.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] S. Gregory. 2010. Finding overlapping communities in networks by label propagation. *New Journal of Physics* 12 (10 2010), 103018. Issue 10.
- [12] 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.
- [13] Yongmin Hu, Jing Wang, Cheng Zhao, Yibo Liu, Cheng Chen, Xiaoliang Cong, and Chao Li. [n. d.]. ParLeiden: Boosting Parallelism of Distributed Leiden Algorithm on Large-scale Graphs. ([n. d.]).
- [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] J. Leskovec. 2021. CS224W: Machine Learning with Graphs | 2021 | Lecture 13.3 - Louvain Algorithm. <https://www.youtube.com/watch?v=0zuiLBOIcsrw>
- [17] H. Lu, M. Halappanavar, and A. Kalyanaraman. 2015. Parallel heuristics for scalable community detection. *Parallel computing* 47 (Aug 2015), 19–37.
- [18] M. Mohammadi, M. Fazlali, and M. Hosseinzadeh. 2020. Accelerating Louvain community detection algorithm on graphic processing unit. *The Journal of supercomputing* (Nov 2020).

- [19] 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.
- [20] M. Newman. 2006. Finding community structure in networks using the eigenvectors of matrices. *Physical review E* 74, 3 (2006), 036104.
- [21] Fabian Nguyen. [n. d.]. *Leiden-Based Parallel Community Detection*. Bachelor’s Thesis. Karlsruhe Institute of Technology, 2021 (zitiert auf S. 31).
- [22] N. Ozaki, H. Tezuka, and M. Inaba. 2016. A simple acceleration method for the Louvain algorithm. *International Journal of Computer and Electrical Engineering* 8, 3 (2016), 207.
- [23] X. Que, F. Checconi, F. Petrini, and J. Gunnels. 2015. Scalable community detection with the louvain algorithm. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE, IEEE, Hyderabad, India, 28–37.
- [24] R. Rotta and A. Noack. 2011. Multilevel local search algorithms for modularity clustering. *Journal of Experimental Algorithmics (JEA)* 16 (2011), 2–1.
- [25] 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.
- [26] Subhajit Sahu. 2023. GVE-Louvain: Fast Louvain Algorithm for Community Detection in Shared Memory Setting. *arXiv preprint arXiv:2312.04876* (2023).
- [27] N. Sattar and S. Arifuzzaman. 2019. Overcoming MPI Communication Overhead for Distributed Community Detection. In *Software Challenges to Exascale Computing*, A. Majumdar and R. Arora (Eds.). Springer Singapore, Singapore, 77–90.
- [28] J. Shi, L. Dhulipala, D. Eisenstat, J. Łącki, and V. Mirrokni. 2021. Scalable community detection via parallel correlation clustering.
- [29] 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.
- [30] V. Traag. 2015. Faster unfolding of communities: Speeding up the Louvain algorithm. *Physical Review E* 92, 3 (2015), 032801.
- [31] V. Traag. 2024. Personal communication. (2024).
- [32] V. Traag, P. Dooren, and Y. Nesterov. 2011. Narrow scope for resolution-limit-free community detection. *Physical Review E* 84, 1 (2011), 016114.
- [33] V. Traag, L. Waltman, and N. Eck. 2019. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports* 9, 1 (Mar 2019), 5233.
- [34] Geerten Verweij. [n. d.]. *Faster Community Detection Without Loss of Quality: Parallelizing the Leiden Algorithm*. Master’s Thesis. Leiden University, 2020.
- [35] L. Waltman and N. Eck. 2013. A smart local moving algorithm for large-scale modularity-based community detection. *The European physical journal B* 86, 11 (2013), 1–14.
- [36] 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.
- [37] 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.
- [38] 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.

## A APPENDIX

### A.1 Community labels of super-vertices

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 9 and 10. Accordingly, we stick to the move-based approach, which is the one recommended by Traag et al. [33]. However, refine-based approach may be more suitable for the design of dynamic Leiden algorithm (for dynamic graphs).

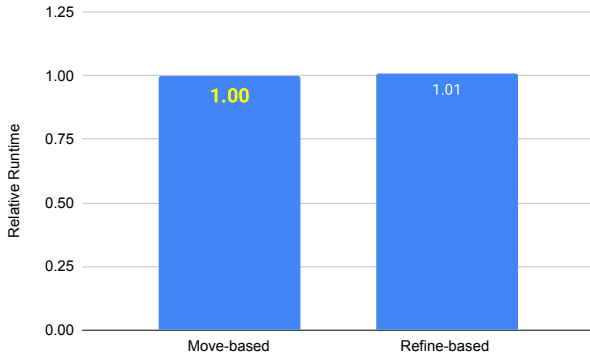


Figure 9: Average relative runtime for *move-based* and *refine-based* communities for super-vertices upon aggregation with parallel Leiden algorithm, for all graphs in the dataset.

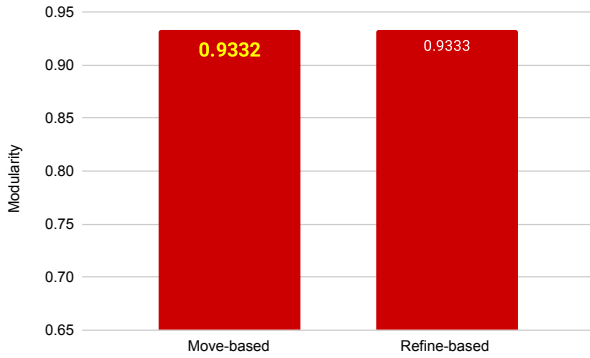


Figure 10: Average modularity for *move-based* and *refine-based* communities for super-vertices upon aggregation with parallel Leiden algorithm, for all graphs in the dataset.

### A.2 Finding disconnected communities

We now outline our parallel algorithm for identifying disconnected communities, given the original graph and the community membership of each vertex. The core concept involves determining the

size of each community, selecting a vertex from each community, traversing within the community from that vertex (avoiding adjacent communities), and marking a community as disconnected if all its vertices cannot be reached. We explore four distinct approaches, differing in the use of parallel Depth-First Search (DFS) or Breadth-First Search (BFS) and whether per-thread or shared *visited* flags are employed. If shared visited flags are used, each thread scans all vertices but processes only its assigned community based on the community ID. Our findings suggest that utilizing parallel BFS traversal with a shared flag vector yields the fastest results. As this is not a heuristic algorithm, all approaches produce identical outcomes. Algorithm 5 illustrates the pseudocode for this approach. Here, the `disconnectedCommunities()` function takes the input graph  $G$  and the community membership  $C$  as input, and it returns the disconnected flag  $D$  for each community.

---

#### Algorithm 5 Finding disconnected communities in parallel.

---

```

▷  $G$ : Input graph
▷  $C$ : Community membership of each vertex
□  $D$ : Disconnected flag for each community
□  $S$ : Size of each community
□  $f_{if}$ : Perform BFS to vertex  $j$  if condition satisfied
□  $f_{do}$ : Perform operation after each vertex is visited
□  $reached$ : Number of vertices reachable from  $i$  to  $i$ 's community
□  $work_t$ : Work-list of current thread

1: function DISCONNECTEDCOMMUNITIES( $G, C$ )
2:    $D \leftarrow \{\}$ ;  $vis \leftarrow \{\}$ 
3:    $S \leftarrow communitySizes(G, C)$ 
4:   for all threads in parallel do
5:     for all  $i \in V$  do
6:        $c \leftarrow C[i]$ ;  $reached \leftarrow 0$ 
7:       ▷ Skip if community  $c$  is empty, or
8:       ▷ does not belong to work-list of current thread.
9:       if  $S[c] = 0$  or  $c \notin work_t$  then continue
10:       $f_{if} \leftarrow (j) \Rightarrow C[j] = c$ 
11:       $f_{do} \leftarrow (j) \Rightarrow reached \leftarrow reached + 1$ 
12:       $bfsVisitForEach(vis, G, i, f_{if}, f_{do})$ 
13:      if  $reached < S[c]$  then  $D[c] \leftarrow 1$ 
14:       $S[c] \leftarrow 0$ 
15:   return  $D$ 

```

---

We now explain Algorithm 5 in detail. First, in line 2, the disconnected community flag  $D$ , and the visited vertices flags  $vis$  are initialized. In line 3, the size of each community  $S$  is obtained in parallel using the `communitySizes()` function. Subsequently, each thread processes each vertex  $i$  in the graph  $G$  in parallel (lines 5-14). In line 6, the community membership of  $i$  ( $c$ ) is determined, and the count of vertices reached from  $i$  is initialized to 0. If community  $c$  is empty or not in the work-list of the current thread  $work_t$ , the thread proceeds to the next iteration (line 9). If however the community  $c$  is non-empty and in the work-list of the current thread  $work_t$ , BFS is performed from vertex  $i$  to explore vertices in the same community, using lambda functions  $f_{if}$  to conditionally perform BFS to vertex  $j$  if it belongs to the same community, and  $f_{do}$  to update the count of *reached* vertices after each vertex is visited during BFS (line 12). If

the number of vertices *reached* during BFS is less than the community size  $S[c]$ , the community  $c$  is marked as disconnected (line 13). Finally, the size of the community  $S[c]$  is updated to 0, indicating that the community has been processed (line 14). Note that the work-list  $work_t$  for each thread with ID  $t$ , is defined as a set containing communities  $[t\chi, t(\chi+1)) \cup [T\chi+t\chi, T\chi+t(\chi+1)) \cup \dots$ , where  $\chi$  is the chunk size, and  $T$  is the number of threads. We use a chunk size of  $\chi = 1024$ .

### A.3 Indirect Comparison with State-of-the-art Leiden Implementations

Finally, we conduct an indirect comparison of the performance of our multicore implementation of the Leiden algorithm with other similar state-of-the-art implementations, as listed in Table 1. Hu

et al. [13] 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 $\times$ , 9.9 $\times$ , and 1.32 $\times$  for ParLeiden-S, ParLeiden-D, and a baseline Leiden implemented on KatanaGraph, on the *com-LiveJournal* graph, with respect to original Leiden [33] (refer to Table 2 in their paper [13]). In contrast, on the same graph, we observe a speedup of 219 $\times$  relative to original Leiden. Consequently, our Leiden implementation outperforms ParLeiden-S, ParLeiden-D, and KatanaGraph Leiden by approximately 18 $\times$ , 22 $\times$ , and 166 $\times$  respectively.