Towards Dynamic Community Detection with Leiden Algorithm

Firstname Lastname
Company
Street, City, Country, PIN.
firstname.lastname@email.com

***Abstract*—Community detection is the process of grouping nodes in a network into clusters. However, real-world graphs often evolve over time, making repeated community detection on such graphs expensive. In this paper, we extend three dynamic approaches, namely, Naive-dynamic (ND), Delta-screening (DS), and Dynamic Frontier (DF), to a fast multicore implementation of the Leiden algorithm — an algorithm known for its high-quality community detection — using subset renumbering, selective refinement, and load balancing of the aggregation phase. This is, to the best of our knowledge, the first attempt at applying such dynamic approaches to the Leiden algorithm.**

***Index Terms*—Dynamic graphs, Parallel Community detection, Leiden algorithm**

I. Introduction

Driven by the ability of graphs to represent complex real-world data and capture intricate relationships among entities, research in graph-structured data has been rapidly growing. A core focus of this field is community detection, which involves decomposing a graph into densely connected groups — revealing the natural structure inherent in the data. Discovering hidden communities in social networks, analyzing regional retail landscapes [1], partitioning large graphs for machine learning [2] and automated microservice decomposition [3] are all applications of community detection.

A challenge in community detection is the absence of prior knowledge about the number and size distribution of communities. To address this, researchers have developed numerous heuristics for finding communities [4], [5]. The quality of identified communities is often measured using fitness metrics such as the modularity score proposed by Newman et al. [6]. These communities are intrinsic when identified based solely on network topology, and they are disjoint when each vertex belongs to only one community [5].

The Louvain method, proposed by Blondel et al. [4], is one of the most popular community detection algorithms [7]. This greedy algorithm employs a two-phase approach, consisting of an iterative local-moving phase and an aggregation phase, to iteratively optimize the modularity metric over several passes [4]. However, it has been observed to produce internally disconnected and poorly connected communities. To address these issues, Traag et al. [8] proposed the **Leiden algorithm**, which introduces a refinement phase between the local-moving and aggregation phases. During this phase, vertices can explore and form sub-communities within the communities identified in the local-moving phase — allowing the algorithm to identify well-connected communities [8].

But many real-world graphs are huge and evolve rapidly over time. For efficiency, algorithms are needed that update results without recomputing from scratch, i.e., **dynamic algorithms**. Dynamic community detection algorithms also allow one to track the evolution of communities over time. However, research efforts have focused on detecting communities in dynamic networks using the Louvain algorithm. None of the works have extended these approaches to the Leiden algorithm.

This paper extends the **Naive-dynamic (ND)** [9], **Delta-screening (DS)** [10], and the recently proposed parallel **Dynamic Frontier (DF)** approach [11] to the Leiden algorithm, using subset renumbering, selective refinement, and load balancing of the aggregation phase.[1] Our algorithms build on top of GVE-Leiden [12], a fast multicore implementation of Static Leiden. While reported speedups are modest (DF Leiden is $3.7\times$ faster than Static Leiden for small updates), we believe these insights are crucial for developing more efficient algorithms for dynamic Leiden.

II. Related work

The **Naive-dynamic (ND)** approach [9] is a straightforward strategy for dynamic community detection involves leveraging the community memberships of vertices from the previous snapshot of the graph.

In contrast, the **Delta-screening (DS)** approach by Zarayeneh et al. [10] examines edge deletions and insertions to the original graph, and identifies a subset of vertices that are likely to be impacted by the change using the modularity objective. Subsequently, only the identified subsets are processed for community state updates, using the Louvain and Smart Local-Moving (SLM) algorithms [13]. Recently, Sahu et al. [11] introduced the parallel **Dynamic Frontier (DF)** approach for the Louvain algorithm, which incrementally identifies an approximate set of affected vertices in the graph with a low run time overhead. They also present a parallel algorithm for the Delta-screening (DS) approach.

However, to the best of our knowledge, none of the proposed approaches have been extended to the Leiden algorithm.

[1] https://bit.ly/45AOL7X

III. PRELIMINARIES

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

A. Community detection

Disjoint community detection involves finding a community membership mapping, $C : V \to \Gamma$, which maps each vertex $i \in V$ to a community ID $c \in \Gamma$, where Γ is the set of community-ids. The vertices within a community c are denoted as V_c, and the community to which a vertex i belongs is denoted as C_i. Furthermore, we define 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 $K_{i \to c} = \{w_{ij} \mid j \in J_{i \to c}\}$, the sum of edge weights within c as $\sigma_c = \sum_{(i,j) \in E \text{ and } C_i = C_j = c} w_{ij}$, and the total edge weight of c as $\Sigma_c = \sum_{(i,j) \in E \text{ and } C_i = c} w_{ij}$ [10].

B. Modularity

Modularity is a metric used to assess the quality of communities identified by community detection algorithms, which are heuristic-based [6]. It ranges from $[-0.5, 1]$ (higher is better), and is calculated as the difference between the fraction of edges within communities and the expected fraction of edges if they were distributed randomly [14]. The modularity Q of the obtained communities can be calculated using Equation 1. The *delta modularity* of moving a vertex i from community d to community c, denoted as $\Delta Q_{i:d \to c}$, is determined using Equation 2.

$$Q = \sum_{c \in \Gamma} \left[\frac{\sigma_c}{2m} - \left(\frac{\Sigma_c}{2m} \right)^2 \right] \tag{1}$$

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

C. Louvain algorithm

The Louvain method is a greedy, modularity optimization based agglomerative algorithm for detecting communities of high quality in a graph. It works in two **phases**: in the **local-moving phase**, each vertex i considers moving to a neighboring community $\{C_j \mid j \in J_i\}$ to maximize modularity increase $\Delta Q_{i:C_i \to C_j}$. In the **aggregation phase**, vertices in the same community are merged into super-vertices. These phases constitute a **pass**, and repeat until modularity gain stops. This produces a hierarchical structure (dendrogram), with the top level representing the final communities.

D. Leiden algorithm

The Leiden algorithm, proposed by Traag et al. [8], addresses the issue of internally disconnected / poorly connected communities of the Louvain method. It does this by introducing a **refinement phase** following the local-moving phase, in each pass of the algorithm. In this phase, vertices within each community undergo constrained merges into other sub-communities within their **community bounds** (communities identified during the local-moving phase), starting from a singleton sub-community — helping identify **sub-communities** within those identified during the local-moving phase. Leiden algorithm 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.

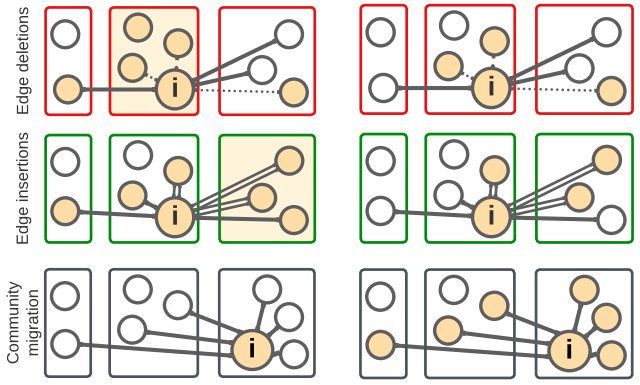
E. Dynamic approaches

A dynamic graph can be represented as a sequence of graphs, where $G^t(V^t, E^t, w^t)$ denotes the graph at time step t. The changes between the graphs $G^{t-1}(V^{t-1}, E^{t-1}, w^{t-1})$ and $G^t(V^t, E^t, w^t)$ at consecutive time steps $t-1$ and t can be described as a batch update Δ^t at time step t. This batch update consists of a set of edge deletions $\Delta^{t-} = \{(i, j) \mid i, j \in V\} = E^{t-1} \setminus E^t$ and a set of edge insertions $\Delta^{t+} = \{(i, j, w_{ij}) \mid i, j \in V; w_{ij} > 0\} = E^t \setminus E^{t-1}$ [10]. We refer to the scenario where Δ^t includes multiple edges being deleted and inserted as a *batch update*.

1) Naive-dynamic (ND) approach [9], [15]–[17]: In this approach, vertices are assigned to communities based on the previous snapshot of the graph, and all vertices are processed, regardless of the edge deletions and insertions in the batch update (hence the term *naive*).

2) Delta-screening (DS) approach [10]: In the DS approach, the batch update consisting of edge deletions $(i, j) \in \Delta^{t-}$ and insertions $(i, j, w) \in \Delta^{t+}$ are first sorted by their source vertex ID. For edge deletions within the same community, i's neighbors and j's community are marked as affected. For edge insertions across communities, the vertex j^* with the highest modularity change among all insertions linked to vertex i is selected, and i's neighbors and j^*'s community are marked as affected. Edge deletions between different communities and edge insertions within the same community are unlikely to impact community membership and are therefore ignored. Figure 1(a) shows an example.

3) Dynamic Frontier (DF) approach [11]: In this approach, for edge deletions between vertices within the same community, or edge insertions between vertices in different communities, the source vertex i is initially marked as affected. As batch updates are undirected, both endpoints i and j are effectively marked. Furthermore, when a vertex i alters its community membership during the community detection process, its neighboring vertices $j \in J_i$ are marked as affected., while i is marked as unaffected. To minimize unnecessary computation, an affected vertex i is also marked as unaffected even if its community remains unchanged. Figure 1(b) shows an example of the DF approach, upon a batch update.



(a) Delta-screening (DS) (b) Dynamic Frontier (DF)

Fig. 1: Illustration of DS and DF approaches, in the presence of edge deletions and insertions, represented with dotted lines and doubled lines, respectively. Vertices/communities identified as affected (initial) by each approach are highlighted in brown.

IV. Approach

We focus our discussion on extending the DF approach to the Leiden algorithm. The procedure to extend the ND and DS approaches to Leiden are similar.

A. No continued passes

Given a batch update, the DF Louvain algorithm [11] first processes incrementally identified affected vertices in the local-moving phase, then runs the aggregation phase to complete the first pass. Later passes process all vertices to avoid the tracking cost, since only the first pass is most expensive. The algorithm stops when the modularity change in a local-moving phase is below the iteration tolerance, τ, or the community count reduction factor after aggregation is below aggregation tolerance, τ_{agg} [11].

However, directly applying the DF approach to the Leiden algorithm produces suboptimal communities with poor modularity. This is because, for small batch updates, the algorithm may converge after just one pass, i.e., a few iterations of the local-moving phase. This is followed by the refinement phase, in order to ensure the properties of the Leiden algorithm. It, however, forces the communities identified during the local-moving phase to be divided into smaller sub-communities. The algorithm then stops before these refined communities can merge into tightly connected groups with sparse links between them — resulting in communities with low modularity. We show how to address this problem.

B. Full Refine method

Since we wish to retain the refinement phase of Leiden due to its desirable properties, one approach to address the above problem is to disallow the algorithm to converge in the first pass. Thus, even for small batch updates, this approach, which we refer to as the **Full Refine** method, would run for multiple passes until convergence occurs in a subsequent local-moving phase — similar to Static Leiden.

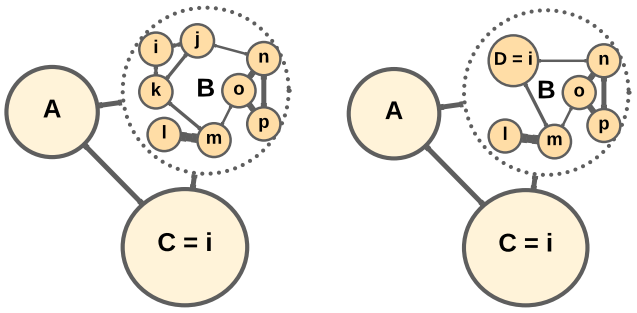
C. Subset Refine method

Nevertheless, for small batch updates, only a few communities are typically affected, and only those need refinement. To identify these communities, we track vertices that migrate between communities during the local-moving phase and mark their source and target communities as changed (i.e., needing refinement) as their sub-communities may have been altered.

However, the community IDs assigned to vertices during the local-moving phase can be inconsistent, i.e., a community $C = i$ might not contain vertex i if that vertex has migrated to another community. This inconsistency can create problems during the refinement phase, where each vertex in the communities being refined must initially belong to its own community.

To understand the issue, consider the example in Figure 2(a). Here, if we refine only community B (containing vertex i) but leave $C = i$ unchanged, vertex i may become an isolated sub-community disconnected from the rest of $C = i$. Worse, in Figure 2(b), vertices j and k could merge with i to form $D = i$, creating two disconnected communities, $C = i$ and $D = i$, with the same ID. Note that this problem occurs only when refining a subset of communities.

To address this, we must assign each community $C = i$ a new ID i', where i' corresponds to one of its constituent vertices. Additionally, we must update the total community edge weights and changed community flags to reflect the new community IDs, as these are initially tied to the old IDs. The psuedocode for this is given in Algorithm 1.



(a) Disconnected isolated vertex (b) Disconnected community forms

Fig. 2: Illustration of issues arising during the refinement phase, when only a subset of communities is refined. Here, circles represent (sub) communities, dotted circles indicate old parent communities (from the local-moving phase), and lines show both inter- and intra-community edges.

We now examine how batch updates affect communities needing refinement. Edge deletions within a community can cause it to split, or even disconnect it internally. However, if the community is isolated, local-moving phase cannot reassign its vertices. A community split may also fail to happen when the community is not isolated. Applying refinement to such communities can tackle this issue. However, as no vertex migration to/from these communities may have occurred, these communities may not be marked for refinement.

To address this, we mark communities as changed when edge deletions occur within them. Similarly, edge insertions affecting different parts of a community may cause a split, so we also mark these communities as changed — see Algorithms 2 and 3. We refer to this selective refinement approach, applied after the local-moving phase and considering both vertex migration and batch updates, as **Subset Refine**. Only marked communities are processed into sub-communities.

Note that we use selective refinement only in the first pass of Leiden, and refine all communities in subsequent passes. This is because untouched communities are already aggregated into super-vertices during the first pass. In the later passes, tracking communities across the super-vertex hierarchy would be costly, and the time saved by selective refinement in these smaller graphs would not justify the overhead.

D. Optimized Aggregation method

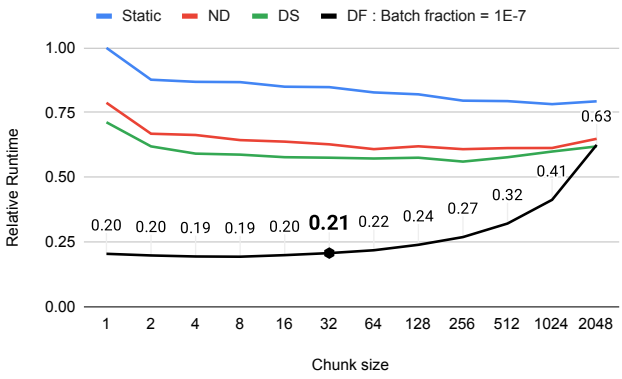
However, the above selective refinement method offers little performance gain for DF Leiden, especially with small batch updates. In these cases, the aggregation phase remains a bottleneck because refined communities are small while unrefined ones are large, creating heavy workloads for threads aggregating the latter into super-vertices.

To improve load balance, heavily loaded threads should be assigned fewer additional communities through dynamic scheduling, assigning smaller community ID ranges (chunks) to threads. However, very small chunks cause high scheduling overhead. Our experiments with chunk sizes from 1 to 2048 on large real-world graphs (Table I) with random batch updates (80% insertions, 20% deletions) and batch sizes $10^{-7}|E|$, $10^{-5}|E|$, and $10^{-3}|E|$ show that a chunk size of 32 gives the best performance for ND, DS, and DF Leiden (we refer the reader to Figure 3). We call this **Optimized Aggregation**. Figure 4 shows DF Leiden runtimes for *Full Refine*, *Subset Refine*, and *Optimized Aggregation* on large graphs with batch sizes from $10^{-7}|E|$ to $0.1|E|$.

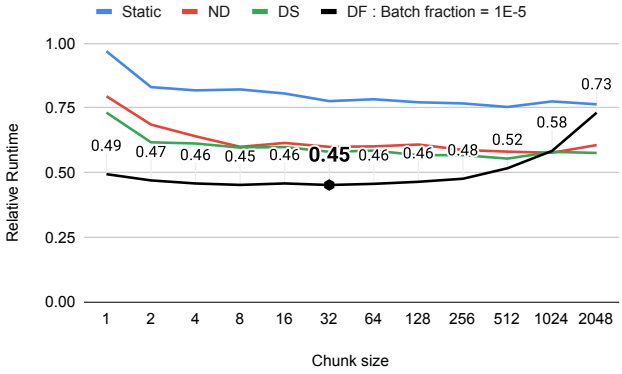
E. Our Parallel Dynamic Frontier (DF) Leiden

We refer to **Optimized Aggregation** based ND, DS, and DF Leiden simply as ND, DS, and DF Leiden. In the first pass, we process vertices marked as affected by the DS and DF approaches, initializing their communities from the previous graph snapshot. In later passes, all super-vertices are processed — as the standard Leiden algorithm. Like DF Louvain [11], we use weighted vertex degrees and total community edge weights as auxiliary information. Note that the guarantees of the original Leiden algorithm [8] extend to our methods, as selective refinement only prunes untouched communities, processing the rest identically.

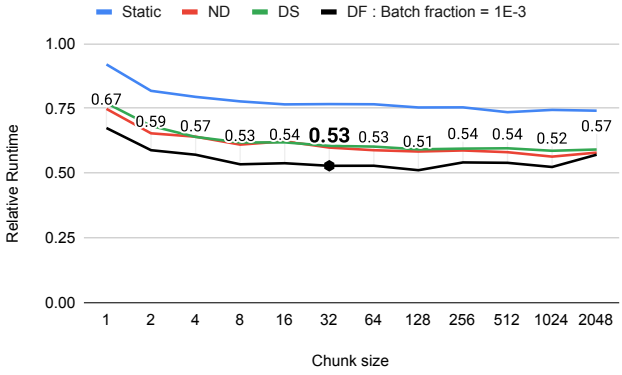
The psuedocode for DF Leiden is given in Algorithms 4, 5, 6, and 7. The psuedocode for the refinement and aggregation phases of our DF Leiden, as well as that for ND and DS Leiden is given in Algorithms 8, 9, 10, and 11.



(a) Relative runtimes on uniformly random batch updates of size $10^{-7}|E|$



(b) Relative runtimes on uniformly random batch updates of size $10^{-5}|E|$



(c) Relative runtimes on uniformly random batch updates of size $10^{-3}|E|$

Fig. 3: Relative Runtime of *Static*, *Naive-dynamic (ND)*, *Delta-screening (DS)*, and *Dynamic Frontier (DF)* Leiden, with varying dynamic schedule chunk size (OpenMP), for aggregation phase of the Leiden algorithm. These tests were conducted on large graphs, with batch updates randomly generated at sizes of $10^{-7}|E|$, $10^{-5}|E|$, and $10^{-3}|E|$. The results suggest that a chunk size of 32 is optimal (highlighted). In this figure, relative runtimes are normalized to maximum runtime, specifically that of Static Leiden with a chunk size of 1 for dynamic scheduling during the aggregation phase.

Algorithm 1 Renumber communities by ID of a vertex within.

▷ $G'(V', E')$: Input/super-vertex graph
▷ C', C'': Current, updated vertex community membership
▷ Σ', Σ'': Current, updated community total edge weight
▷ C'_Δ, C''_Δ: Current, updated changed communities flag
☐ Γ': Set of communities in C'

function $subsetRenumber(G', C', \Sigma', C'_\Delta)$
  $C'' \leftarrow \Sigma'' \leftarrow C''_\Delta \leftarrow C'_v \leftarrow \{\}$
  ▷ Obtain any vertex from each community
  for all $i \in V'$ **in parallel do**
    $c' \leftarrow C'[i]$
    if $C'_v[c] = \text{EMPTY}$ **then** $C'_v[c] \leftarrow i$
  ▷ Update community weights and changed status
  for all $c' \in \Gamma'$ **in parallel do**
    $c'' \leftarrow C'_v[c']$
    if $c'' \neq \text{EMPTY}$ **then**
      $\Sigma''[c''] \leftarrow \Sigma'[c']$
      $C''_\Delta[c''] \leftarrow C'_\Delta[c']$
  ▷ Update community memberships
  for all $i \in V'$ **in parallel do**
    $C''[i] \leftarrow C'_v[C'[i]]$
  ▷ Update in-place
  $C' \leftarrow C''$ **;** $\Sigma' \leftarrow \Sigma''$ **;** $C'_\Delta \leftarrow C''_\Delta$

Algorithm 2 Communities to refine due to batch update.

▷ $G^t(V^t, E^t)$: Current input graph
▷ $\Delta^t(\Delta^{t-}, \Delta^{t+})$: Edge deletions and insertions
▷ C^{t-1}: Previous community of each vertex
☐ C'_Δ: Changed communities flag

function $changedComms(G^t, C^{t-1}, \Delta^{t-}, \Delta^{t+})$
  $C'_\Delta \leftarrow \{\}$
  if is dynamic alg. **then**
    for all $(i, j) \in \Delta^{t-} \cup \Delta^{t+}$ **in parallel do**
      if $C^{t-1}[i] = C^{t-1}[j]$ **then** $C'_\Delta[i] \leftarrow 1$
  else $C'_\Delta \leftarrow \{1 \ \forall \ V^t\}$
  return C'_Δ

Algorithm 3 Split marked communities into isolated vertices.

☐ $G'(V', E')$: Current/super-vertex graph
☐ C': Current community of each vertex in G'
☐ K': Current weighted-degree of each vertex in G'
☐ Σ': Current community total edge weight in G'
☐ C'_Δ: Changed communities flag

function $breakChangedComms(G', C', K', \Sigma', C'_\Delta)$
  for all $i \in V'$ **in parallel do**
    if $C'_\Delta[C'[i]] = 0$ **then continue**
    $C'[i] \leftarrow i$ **;** $\Sigma'[i] \leftarrow K'[i]$

Algorithm 4 Our Parallel *Dynamic Frontier (DF)* Leiden.

▷ $G^t(V^t, E^t)$: Current/updated input graph
▷ $\Delta^t(\Delta^{t-}, \Delta^{t+})$: Edge deletions and insertions
▷ C^{t-1}, C^t: Previous, current community of each vertex
▷ K^{t-1}, K^t: Previous, current weighted-degree of vertices
▷ Σ^{t-1}, Σ^t: Prev., current total edge weight of communities
☐ δV: Flag vector indicating if each vertex is affected
☐ *isAffected*(i): Is vertex i is marked as affected?
☐ *inAffectedRange*(i): Can i be incrementally marked?
☐ *onChange*(i): What happens if i changes its community?
☐ F: Lambda functions passed to parallel Leiden (Alg. 6)

function $dfLeiden(G^t, \Delta^t, C^{t-1}, K^{t-1}, \Sigma^{t-1})$
  ▷ Mark initial affected vertices
  for all $(i, j) \in \Delta^{t-}$ **in parallel do**
    if $C^{t-1}[i] = C^{t-1}[j]$ **then** $\delta V[i] \leftarrow 1$
  for all $(i, j, w) \in \Delta^{t+}$ **in parallel do**
    if $C^{t-1}[i] \neq C^{t-1}[j]$ **then** $\delta V[i] \leftarrow 1$
  function *isAffected*(i)
    return $\delta V[i]$
  function *inAffectedRange*(i)
    return 1
  function *onChange*(i)
    for all $j \in G^t.neighbors(i)$ **do** $\delta V[j] \leftarrow 1$
  $F \leftarrow \{$*isAffected*, *inAffectedRange*, *onChange*$\}$
  ▷ Use K^{t-1}, Σ^{t-1} as auxiliary information (Alg. 5)
  $\{K^t, \Sigma^t\} \leftarrow$ *updateWeights*$(G^t, \Delta^t, C^{t-1}, K^{t-1}, \Sigma^{t-1})$
  ▷ Obtain updated communities (Alg. 6)
  $C^t \leftarrow leiden(G^t, C^{t-1}, K^t, \Sigma^t, F)$
  return $\{C^t, K^t, \Sigma^t\}$

Algorithm 5 Updating vertex/community weights in parallel.

▷ $G^t(V^t, E^t)$: Current input graph
▷ $\Delta^t(\Delta^{t-}, \Delta^{t+})$: Edge deletions and insertions
▷ C^{t-1}: Previous community of each vertex
▷ K^{t-1}: Previous weighted-degree of each vertex
▷ Σ^{t-1}: Previous total edge weight of each community
☐ K: Updated weighted-degree of each vertex
☐ Σ: Updated total edge weight of each community
☐ $work_{th}$: Work-list of current thread

function $updateWeights(G^t, \Delta^t, C^{t-1}, K^{t-1}, \Sigma^{t-1})$
  $K \leftarrow K^{t-1}$ **;** $\Sigma \leftarrow \Sigma^{t-1}$
  for all threads in parallel do
    for all $(i, j, w) \in \Delta^{t-}$ **do**
      $c \leftarrow C^{t-1}[i]$
      if $i \in work_{th}$ **then** $K[i] \leftarrow K[i] - w$
      if $c \in work_{th}$ **then** $\Sigma[c] \leftarrow \Sigma[c] - w$
    for all $(i, j, w) \in \Delta^{t+}$ **do**
      $c \leftarrow C^{t-1}[i]$
      if $i \in work_{th}$ **then** $K[i] \leftarrow K[i] + w$
      if $c \in work_{th}$ **then** $\Sigma[c] \leftarrow \Sigma[c] + w$
  return $\{K, \Sigma\}$

Algorithm 6 Our Dynamic-supporting Parallel Leiden.

▷ $G^t(V^t, E^t)$: Current input graph
▷ $\Delta^t(\Delta^{t-}, \Delta^{t+})$: Edge deletions and insertions
▷ C^{t-1}: Previous community of each vertex
▷ K^t: Current weighted-degree of each vertex
▷ Σ^t: Current total edge weight of each community
▷ F: Lambda functions passed to parallel Leiden
☐ $G'(V', E')$: Current/super-vertex graph
☐ C, C': Current community of each vertex in G^t, G'
☐ K, K': Current weighted-degree of each vertex in G^t, G'
☐ Σ, Σ': Current community total edge weight in G^t, G'
☐ C'_B: Community bound of each vertex
☐ C'_Δ: Changed communities flag
☐ τ: Iteration tolerance

function $leiden(G^t, C^{t-1}, K^t, \Sigma^t, F)$
  ▷ Mark affected vertices as unprocessed
  for all $i \in V^t$ **do**
    if $F.isAffected(i)$ **then** Mark i as unprocessed
  ▷ Initialization phase
  Vertex membership: $C \leftarrow [0..|V^t|)$
  $G' \leftarrow G^t$; $C' \leftarrow C^{t-1}$; $K' \leftarrow K^t$; $\Sigma' \leftarrow \Sigma^t$
  ▷ Mark changed communities based on Δ^t (Alg. 2)
  $C'_\Delta \leftarrow changedCommunities(G^t, C^{t-1}, \Delta^{t-}, \Delta^{t+})$
  ▷ Each pass of Leiden consists of 3 phases
  for all $l_p \in [0..\text{MAX\_PASSES})$ **do**
    ▷ Perform local-moving phase (Alg. 7)
    $l_i \leftarrow leidenMove(G', C', K', \Sigma', C'_\Delta, F)$
    ▷ Label communities by internal vertex (Alg. 1)
    $leidenSubsetRenumber(G', C', \Sigma', C'_\Delta)$
    $C'_B \leftarrow C'$ ▷ Comm. bounds for refinement phase
    ▷ Split marked communities to vertices (Alg. 3)
    $breakChangedComms(G', C', K', \Sigma', C'_\Delta)$
    ▷ Perform refinement phase (Alg. 10)
    $leidenRefine(G', C'_B, C', K', \Sigma', \tau)$
    ▷ Check if convergence has been achieved
    if not first pass **and** $l_i \leq 1$ **then break**
    $C' \leftarrow$ Renumber communities in C'
    $C \leftarrow$ Lookup dendrogram using C to C'
    ▷ Perform aggregation phase (Alg. 11)
    $G' \leftarrow leidenAggregate(G', C')$
    ▷ Prepare for next pass (isolated communities)
    $\Sigma' \leftarrow K' \leftarrow$ Weight of each vertex in G'
    Mark all vertices in G' as unprocessed
    $C' \leftarrow [0..|V'|)$ ▷ Use refine-based membership
    ▷ All communities are refined in the next pass
    $C'_\Delta \leftarrow \{1 \ \forall \ V'\}$
    ▷ Scale iteration tolerance (threshold scaling)
    $\tau \leftarrow \tau / \text{TOLERANCE\_DROP}$
  $C \leftarrow$ Lookup dendrogram using C to C'
  return C

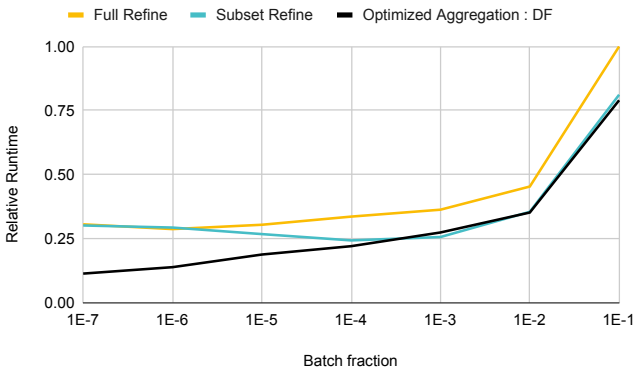


Fig. 4: Relative Runtime of DF Leiden, incorporating three successive optimizations: *Full Refine*, *Subset Refine*, and *Optimized Aggregation*. These are tested on large graphs with randomly generated batch updates of size $10^{-7}|E|$ to $0.1|E|$, consisting of 80% edge insertions and 20% deletions.

F. Implementation details

We use asynchronous Leiden with a single community membership vector, where threads process different graph parts independently — speeding up convergence, but adding variability to results. Each thread keeps its own hashtable to track delta-modularity in local-moving and refinement phases, and for edge weights between super-vertices in aggregation. Key optimizations include OpenMP *dynamic* scheduling, capping iterations at 20 per pass, dropping tolerance by $10\times$ from 0.01, vertex pruning, parallel prefix sums, preallocated CSR structures for super-vertex and community graphs, and fast collision-free per-thread hashtables [12].

Unlike GVE-Leiden [12], we always run the aggregation phase, even when few communities merge, to preserve community quality with minimal runtime overhead — across Static, ND, DS, and DF Leiden. We also use the refine-based variation of Leiden, where community labels of super-vertices are based on refinement phase, not local-moving — enabling community splits in the scenarios described in Section IV-C.

G. Time and Space complexity

The time complexity of ND, DS, and DF Leiden is the same as Static Leiden, i.e., $O(L|E^t|)$, where L is the total number of iterations performed. However, in the first pass, local-moving and refinement costs are lower and depend on the batch update's size and type. The space complexity is also the same, i.e., $O(T|V^t| + |E^t|)$, where $T|V^t|$ accounts for per-thread collision-free hashtables [12].

V. Evaluation

A. Experimental setup

1) System: We use a server with a 64-core x86 AMD EPYC-7742 CPU (2.25 GHz), with 4 MB L1 cache per core, 32 MB L2 cache per core, and a shared 256 MB L3 cache. It has 512 GB DDR4 RAM, and runs Ubuntu 20.04.

Algorithm 7 Local-moving phase of our Parallel Leiden.

▷ $G'(V', E')$: Input/super-vertex graph
▷ C': Community membership of each vertex
▷ K': Total edge weight of each vertex
▷ Σ': Total edge weight of each community
▷ C'_Δ: Changed communities flag
▷ F: Lambda functions passed to parallel Leiden
□ H_t: Collision-free per-thread hashtable
□ l_i: Number of iterations performed
□ τ: Per iteration tolerance

function $leidenMove(G', C', K', \Sigma', C'_\Delta, F)$
  for all $l_i \in [0..\text{MAX\_ITERATIONS})$ **do**
    Total delta-modularity per iteration: $\Delta Q \leftarrow 0$
    for all unprocessed $i \in V'$ **in parallel do**
      Mark i as processed (prune)
      if not $F.inAffectedRange(i)$ **then continue**
      $H_t \leftarrow scanCommunities(\{\}, G', C', i, false)$
      ▷ Use H_t, K', Σ' to choose best community
      $c^* \leftarrow$ Best community linked to i in G'
      $\delta Q^* \leftarrow$ Delta-modularity of moving i to c^*
      if $c^* = C'[i]$ **then continue**
      $\Sigma'[C'[i]] -= K'[i]$ **;** $\Sigma'[c^*] += K'[i]$ **atomic**
      $C'[i] \leftarrow c^*$ **;** $\Delta Q \leftarrow \Delta Q + \delta Q^*$
      Mark neighbors of i as unprocessed
      if is dynamic **then** $C'_\Delta[c] \leftarrow C'_\Delta[c^*] \leftarrow 1$
    if $\Delta Q \leq \tau$ **then break** ▷ Locally converged?
  return l_i

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

2) Configuration: We use 32-bit unsigned integers for vertex IDs and 32-bit floats for edge weights, switching to 64-bit floats for aggregated values like total edge weight and modularity. Affected vertices and changed communities use 8-bit integer vectors. Local-moving, refinement, and aggregation phases use OpenMP *dynamic* scheduling with a chunk size of 2048, except in the aggregation phase of ND, DS, and DF Leiden, where a chunk size of 32 is used. Our ND, DS, and DF Leiden build on GVE-Leiden [12], a state-of-the-art parallel Leiden. All implementations run on 64 threads and are compiled with GCC 9.4 and OpenMP 5.0.

3) Dataset: We experiment on 12 large real-world graphs with random batch updates, listed in Table I, obtained from the SuiteSparse Matrix Collection. These graphs have between $3.07M$ and $214M$ vertices, and $25.4M$ to $3.80B$ edges.

4) Batch generation: For each base graph in Table I, we generate random batch updates [10] consisting of an 80%–20% mix of unit-weight edge insertions and deletions [19], without vertex additions or removals. Batch sizes range from $10^{-7}|E|$ to $0.1|E|$ (100 to 10^8 edges for a billion-edge graph). For each size, we apply five random updates and report the mean.

TABLE I: List of 12 graphs from the SuiteSparse Matrix Collection [18], with directed graphs marked by $*$. Here, $|V|$ = vertices, $|E|$ = edges (after symmetrizing), $|\Gamma|$ = communities from the *Static Leiden* algorithm [12].

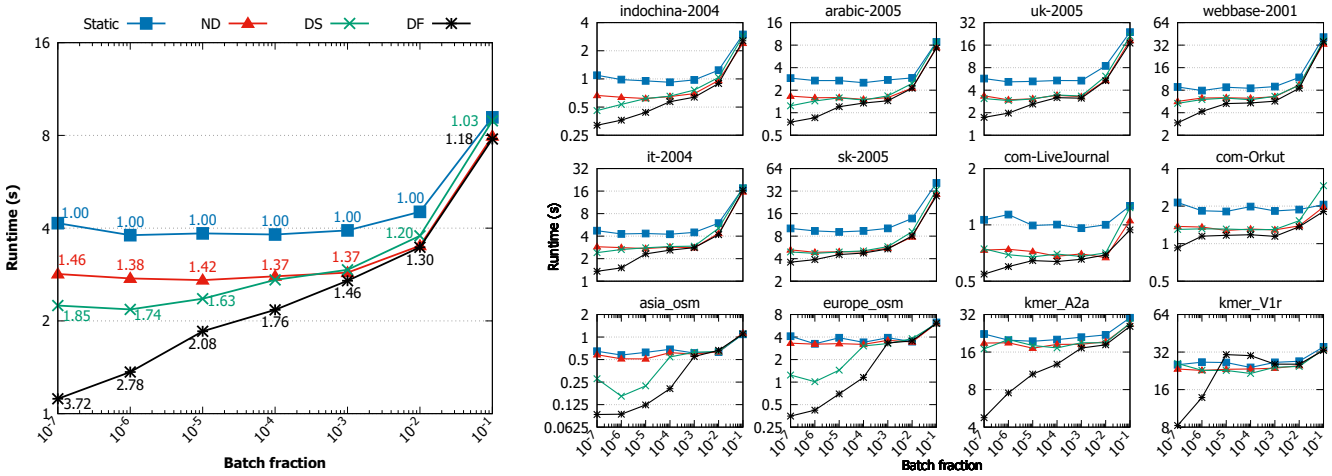
Graph	$\|V\|$	$\|E\|$	$\|\Gamma\|$
Web Graphs (LAW)			
indochina-2004*	7.41M	341M	2.68K
arabic-2005*	22.7M	1.21B	2.92K
uk-2005*	39.5M	1.73B	18.2K
webbase-2001*	118M	1.89B	2.94M
it-2004*	41.3M	2.19B	4.05K
sk-2005*	50.6M	3.80B	2.67K
Social Networks (SNAP)			
com-LiveJournal	4.00M	69.4M	3.09K
com-Orkut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

B. Performance Comparison

We now evaluate our parallel ND, DS, and DF Leiden against Static Leiden (based on GVE-Leiden [12]), on large graphs with random batch updates (see Section V-A4). Static Leiden restarts from scratch after each update, while ND, DS, and DF Leiden reuse prior communities; DS and DF Leiden process only affected vertices in the local-moving phase.

From Figure 5, we observe that ND, DS, and DF Leiden achieve mean speedups of $1.37\times$, $1.47\times$, and $1.98\times$ over Static Leiden; with higher gains for small updates: $1.46\times$, $1.74\times$, and $3.72\times$ on batch updates of size $10^{-7}|E|$. Speedups are moderate on web/social graphs; on low-degree road networks, ND Leiden slightly outperforms Static Leiden, while DS and DF Leiden perform much better by processing fewer vertices/communities. On protein k-mer graphs, DS Leiden offers low gains, but DF Leiden greatly outperforms Static Leiden for small batch updates.

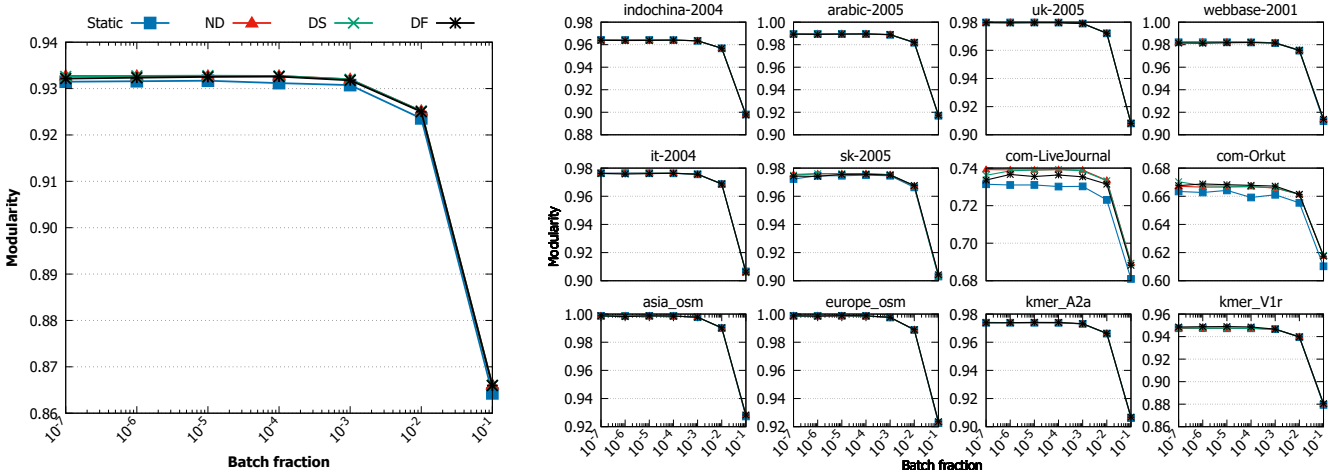
As shown in Figure 6, ND, DS, and DF Leiden achieve similar modularity to Static Leiden, but slightly outperform it on social networks. This is likely because weak community structures in such networks require more iterations for better results. Static Leiden limits iterations per pass for speed, while our methods reuse prior assignments to converge faster and with higher modularity, especially on social networks. Overall, Static Leiden's modularity is only slightly lower, with differences averaging under 0.002.



(a) Overall result

(b) Results on each graph

Fig. 5: Runtime (logarithmic scale) of our multicore implementation of *Naive-dynamic (ND)*, *Delta-screening (DS)*, and *Dynamic Frontier (DF) Leiden*, compared to *Static Leiden* (based on GVE-Leiden [12]), on large graphs with randomly generated batch updates. The size of these batch updates ranges from $10^{-7}|E|$ to $0.1|E|$ in multiples of 10, with the updates comprising 80% edge insertions and 20% edge deletions to simulate realistic dynamic graph changes. The right subfigure shows the runtime of each algorithm for individual graphs in the dataset, while the left subfigure displays overall runtimes using the geometric mean for consistent scaling across graphs. Speedup of each algorithm compared to Static Leiden is indicated on respective lines.



(a) Overall result

(b) Results on each graph

Fig. 6: Modularity comparison of our multicore implementation of *Naive-dynamic (ND)*, *Delta-screening (DS)*, and *Dynamic Frontier (DF) Leiden*, compared to *Static Leiden* (based on GVE-Leiden [12]), on large graphs with randomly generated batch updates. These batch updates vary in size from $10^{-7}|E|$ to $0.1|E|$ in powers of 10, consisting of 80% edge insertions and 20% edge deletions to mimic realistic dynamic graph updates. Here, the right subfigure shows the modularity for each algorithm for individual graphs in the dataset, while the left subfigure displays overall modularity obtained using arithmetic mean.

Also note in Figure 5 that Static Leiden's runtime grows with larger batch updates. This is due to: **(1)** increased graph size from 80% edge insertions, and **(2)** random edge updates disrupting community structure, requiring more iterations to converge. This disruption also explains the drop in modularity with larger batches (see Figure 6).

ND, DS, and DF Leiden achieve only modest speedups over Static Leiden because, unlike Louvain, Leiden must always run the refinement phase to avoid poorly connected or disconnected communities. However, stopping early leads to low modularity since large, high-quality clusters have not yet formed. Moreover, these methods can speed up only the local-moving phase of the first pass, which accounts for just $\approx 37\%$ of Static Leiden's runtime. These factors limit speedup, though DF Leiden remains the preferred dynamic method.

C. Affected vertices and Changed communities

We now examine the fraction of affected vertices and changed communities tracked by DS and DF Leiden on datasets from Table I, with batch updates from $10^{-7}|E|$ to $0.1|E|$. Tracking is done only in the first Leiden pass, as given in Sections IV-C and IV-D, and shown in Figure 7.

Figure 7 shows that DF Leiden marks fewer affected vertices and changed communities than DS Leiden, but with only a small runtime gain, since many DS-marked vertices keep their community labels and converge quickly. As expected, more affected vertices slow both algorithms (Figures 5 and 7). Higher changed communities also raise first-pass refinement costs, while future costs depend on graph's nature.

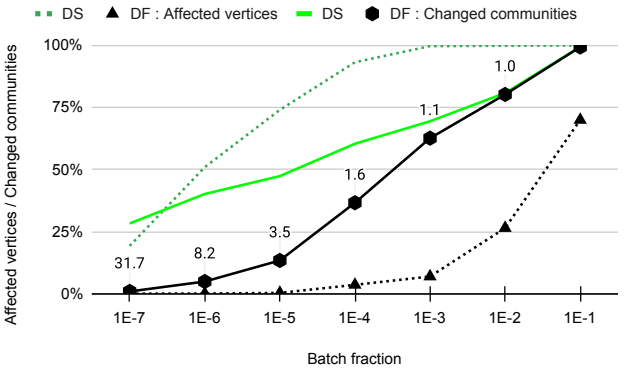


Fig. 7: Fraction of affected vertices (dotted) / changed communities (solid) for DS and DF Leiden on graphs in Table I. Labels show changed communities ratio in DS to DF Leiden.

D. Scalability

Finally, we study the strong-scaling of ND, DS, and DF Leiden, and compare it to Static Leiden. For this, we fix batch size at $10^{-3}|E|$, vary threads from 1 to 64.

As shown in Figure 8, ND, DS, and DF Leiden obtain a speedup of $10.2\times$, $9.9\times$, and $9.0\times$ respectively at 32 threads (with respect to sequential); with their speedup increasing at a mean rate of $1.59\times$, $1.58\times$, and $1.55\times$, respectively, for

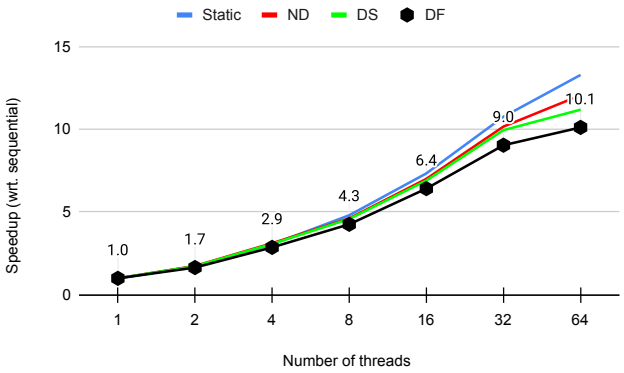


Fig. 8: Strong scaling of ND, DS, and DF Leiden, compared to Static Leiden, on batch updates of size $10^{-3}|E|$.

every doubling of threads. At 64 threads, NUMA affects the performance of our algorithms (NPS 4). In addition, increasing threads makes our algorithm less asynchronous, i.e., similar to Jacobi iterative method, contributing to lower performance.

VI. Conclusion

In this paper, we adapted three approaches for community detection in dynamic graphs to a fast multicore implementation [12] of the Leiden algorithm [8]. We expect these results to encourage further investigation of dynamic Leiden algorithm.

References

[1] A. Verhetsel, J. Beckers, and J. Cant, "Regional retail landscapes emerging from spatial network analysis," *Regional Studies*, vol. 56, no. 11, pp. 1829–1844, 2022.

[2] Y. Bai, C. Constantin, and H. Naacke, "Leiden-fusion partitioning method for effective distributed training of graph embeddings," in *ECML PKDD '24*. Springer, 2024, pp. 366–382.

[3] L. Cao and C. Zhang, "Implementation of domain-oriented microservices decomposition based on node-attributed network," in *ICSCA '22*, 2022, pp. 136–142.

[4] V. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mech.*, vol. 2008, no. 10, p. P10008, Oct 2008.

[5] S. Gregory, "Finding overlapping communities in networks by label propagation," *New J. Phys.*, vol. 12, p. 103018, 10 2010.

[6] M. Newman, "Detecting community structure in networks," *EPJ B*, vol. 38, no. 2, pp. 321–330, Mar 2004.

[7] A. Lancichinetti and S. Fortunato, "Community detection algorithms: a comparative analysis." *Phys. Rev. E*, vol. 80, no. 5 Pt 2, p. 056117, Nov 2009.

[8] V. Traag, L. Waltman, and N. Eck, "From Louvain to Leiden: guaranteeing well-connected communities." *Scientific Reports*, vol. 9, no. 1, p. 5233, Mar 2019.

[9] T. Aynaud and J. Guillaume, "Static community detection algorithms for evolving networks," in *WiOpt '10*. Avignon, France: IEEE, 2010, pp. 513–519.

[10] N. Zarayeneh and A. Kalyanaraman, "Delta-Screening: A Fast and Efficient Technique to Update Communities in Dynamic Graphs," *IEEE TNSE*, vol. 8, no. 2, pp. 1614–1629, Apr 2021.

[11] S. Sahu, K. Kothapalli, and D. S. Banerjee, "Shared-memory parallel algorithms for community detection in dynamic graphs," in *IPDPSW '24*. IEEE, 2024, pp. 250–259.

[12] ——, "Fast leiden algorithm for community detection in shared memory setting," in *ICPP '24*, 2024, pp. 11–20.

[13] L. Waltman and N. Eck, "A smart local moving algorithm for large-scale modularity-based community detection," *EPJ B*, vol. 86, no. 11, pp. 1–14, 2013.

[14] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *TKDE*, vol. 20, no. 2, pp. 172–188, 2007.

[15] W. Chong and L. Teow, "An incremental batch technique for community detection," in *FUSION '13*. Istanbul, Turkey: IEEE, 2013, pp. 750–757.

[16] J. Shang, L. Liu, F. Xie, Z. Chen, J. Miao, X. Fang, and C. Wu, "A real-time detecting algorithm for tracking community structure of dynamic networks," 2014.

[17] D. Zhuang, J. Chang, and M. Li, "Dynamo: Dynamic community detection by incrementally maximizing modularity," *IEEE TKDE*, vol. 33, no. 5, pp. 1934–1945, 2019.

[18] S. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. Davis, M. Henderson, Y. Hu, and R. Sandstrom, "The SuiteSparse matrix collection website interface," *JOSS*, vol. 4, no. 35, p. 1244, Mar 2019.

[19] K. Nijhawan, R. Saravanan, S. Sahu, and K. Kothapalli, "Evolvgraph: A tool for property-constrained generation of dynamic graphs," in *HiPCW '24*. IEEE, 2024, pp. 151–152.

APPENDIX

A. Our Parallel Naive-dynamic (ND) Leiden

Algorithm 8 details our multicore implementation of Naive dynamic (ND) Leiden. Here, vertices are assigned to communities based on the previous snapshot of the graph, with all vertices being processed regardless of edge deletions and insertions in the batch update. The algorithm accepts the current/updated graph snapshot G^t, edge deletions Δ^{t-} and insertions Δ^{t+} in the batch update, the previous community membership C^{t-1} for each vertex, the weighted degree of each vertex K^{t-1}, and the total edge weight of each community Σ^{t-1}. The output includes the updated community memberships C^t, the updated weighted degrees K^t, and the updated total edge weights of communities Σ^t.

In the algorithm, we begin by defining two lambda functions for the Leiden algorithm: *isAffected()* (lines 3 to 4) and *inAffectedRange()* (lines 5 to 6). These functions indicate that all vertices in the graph G^t should be marked as affected and that these vertices can be incrementally marked as affected, respectively. Unlike previous approaches, we then use K^{t-1} and Σ^{t-1}, along with the batch updates Δ^{t-} and Δ^{t+}, to quickly compute K^t and Σ^t, which are required for the local-moving phase of the Leiden algorithm (line 9). The lambda functions, along with the total vertex and edge weights, are then employed to run the Leiden algorithm and obtain updated community assignments C^t (line 11). Finally, C^t is returned, along K^t and Σ^t as updated auxiliary information (line 12).

Algorithm 8 Our Parallel *Naive-dynamic (ND)* Leiden.

▷ $G^t(V^t, E^t)$: Current/updated input graph
▷ $\Delta^t(\Delta^{t-}, \Delta^{t+})$: Edge deletions and insertions
▷ C^{t-1}, C^t: Previous, current community of each vertex
▷ K^{t-1}, K^t: Previous, current weighted-degree of vertices
▷ Σ^{t-1}, Σ^t: Prev., current total edge weight of communities
☐ *isAffected*(i): Is vertex i is marked as affected?
☐ *inAffectedRange*(i): Can i be incrementally marked?
☐ F: Lambda functions passed to parallel Leiden (Alg. 6)

1: **function** $ndLeiden(G^t, \Delta^t, C^{t-1}, K^{t-1}, \Sigma^{t-1})$
2: ▷ Mark affected vertices
3: **function** *isAffected*(i)
4: **return** 1
5: **function** *inAffectedRange*(i)
6: **return** 1
7: $F \leftarrow \{$*isAffected*, *inAffectedRange*$\}$
8: ▷ Use K^{t-1}, Σ^{t-1} as auxiliary information (Alg. 5)
9: $\{K^t, \Sigma^t\} \leftarrow$ *updateWeights*$(G^t, \Delta^t, C^{t-1}, K^{t-1}, \Sigma^{t-1})$
10: ▷ Obtain updated communities (Alg. 6)
11: $C^t \leftarrow leiden(G^t, C^{t-1}, K^t, \Sigma^t, F)$
12: **return** $\{C^t, K^t, \Sigma^t\}$

B. Our Parallel Delta-screening (DS) Leiden

Algorithm 9 presents the pseudocode for our multicore implementation of Delta-screening (DS) Leiden. It employs modularity-based scoring to identify an approximate region of the graph where vertices are likely to change their community membership [10]. It takes as input the current/updated graph snapshot G^t, edge deletions Δ^{t-} and insertions Δ^{t+} in the batch update, the previous community memberships of vertices C^{t-1}, the weighted degrees of vertices K^{t-1}, and the total edge weights of communities Σ^{t-1}. The algorithm outputs the updated community memberships C^t, weighted degrees K^t, and total edge weights of communities Σ^t. Prior to processing, the batch update — which includes edge deletions $(i, j, w) \in \Delta^{t-}$ and insertions $(i, j, w) \in \Delta^{t+}$ — are sorted separately

Algorithm 9 Our Parallel *Delta-screening (DS)* Leiden.

▷ $G^t(V^t, E^t)$: Current/updated input graph
▷ $\Delta^t(\Delta^{t-}, \Delta^{t+})$: Edge deletions and insertions
▷ C^{t-1}, C^t: Previous, current community of each vertex
▷ K^{t-1}, K^t: Previous, current weighted-degree of vertices
▷ Σ^{t-1}, Σ^t: Prev., current total edge weight of communities
☐ $\delta V, \delta E, \delta C$: Is vertex, neighbors, or community affected?
☐ H: Hashtable mapping a community to associated weight
☐ $isAffected(i)$: Is vertex i is marked as affected?
☐ $inAffectedRange(i)$: Can i be incrementally marked?
☐ F: Lambda functions passed to parallel Leiden (Alg. 6)

1: **function** $dsLeiden(G^t, \Delta^t, C^{t-1}, K^{t-1}, \Sigma^{t-1})$
2: $H, \delta V, \delta E, \delta C \leftarrow \{\}$
3: ▷ Mark affected vertices
4: **for all** $(i, j, w) \in \Delta^{t-}$ **in parallel do**
5: **if** $C^{t-1}[i] = C^{t-1}[j]$ **then**
6: $\delta V[i], \delta E[i], \delta C[C^{t-1}[j]] \leftarrow 1$
7: **for all** unique source vertex $i \in \Delta^{t+}$ **in parallel do**
8: $H \leftarrow \{\}$
9: **for all** $(i', j, w) \in \Delta^{t+} \mid i' = i$ **do**
10: **if** $C^{t-1}[i] \neq C^{t-1}[j]$ **then**
11: $H[C^{t-1}[j]] \leftarrow H[C^{t-1}[j]] + w$
12: $c^* \leftarrow$ Best community linked to i in H
13: $\delta V[i], \delta E[i], \delta C[c^*] \leftarrow 1$
14: **for all** $i \in V^t$ **in parallel do**
15: **if** $\delta E[i]$ **then**
16: **for all** $j \in G^t.neighbors(i)$ **do**
17: $\delta V[j] \leftarrow 1$
18: **if** $\delta C[C^{t-1}[i]]$ **then**
19: $\delta V[i] \leftarrow 1$
20: **function** *isAffected*(i)
21: **return** $\delta V[i]$
22: **function** *inAffectedRange*(i)
23: **return** $\delta V[i]$
24: $F \leftarrow \{$*isAffected*, *inAffectedRange*$\}$
25: ▷ Use K^{t-1}, Σ^{t-1} as auxiliary information (Alg. 5)
26: $\{K^t, \Sigma^t\} \leftarrow$ *updateWeights*$(G^t, \Delta^t, C^{t-1}, K^{t-1}, \Sigma^{t-1})$
27: ▷ Obtain updated communities (Alg. 6)
28: $C^t \leftarrow leiden(G^t, C^{t-1}, K^t, \Sigma^t, F)$
29: **return** $\{C^t, K^t, \Sigma^t\}$

by the source vertex ID i.

In the algorithm, we start by initializing a hashtable H that maps communities to their associated weights, and we set the affected flags δV, δE, and δC, which indicate whether a vertex, its neighbors, or its community is affected by the batch update (lines 2). We then parallelly process edge deletions Δ^{t-} and insertions Δ^{t+}. For each deletion $(i, j, w) \in \Delta^{t-}$ where vertices i and j belong to the same community, we mark the source vertex i, its neighbors, and its community as affected (lines 4-6). For each unique source vertex i in insertions $(i, j, w) \in \Delta^{t+}$ where i and j belong to different communities, we determine the community c^* with the highest delta-modularity if i moves to one of its neighboring communities, marking i, its neighbors, and the community c^* as affected (lines 7-13). We disregard deletions between different communities and insertions within the same community. Using the affected neighbor δE and community flags δC, we mark affected vertices in δV (lines 14-19). Subsequently, similar to ND Leiden, we utilize K^{t-1} and Σ^{t-1}, along with Δ^{t-} and Δ^{t+}, to quickly derive K^t and Σ^t (line 9). We define the necessary lambda functions *isAffected()* (lines 20-21) and *inAffectedRange()* (lines 22-23), and execute the Leiden algorithm, resulting in updated community assignments C^t (line 28). We then return updated community memberships C^t along K^t, Σ^t as updated auxiliary information (line 29).

C. Our Dynamic-supporting Parallel Leiden

1) Refinement phase of our Parallel Leiden: The pseudocode outlining the refinement phase of our Parallel Leiden is presented in Algorithm 7. This phase closely resembles the local-moving phase but incorporates the community membership obtained for each vertex as a *community bound*. In this phase, each vertex is required to select a community within its community bound to join, aiming to maximize modularity through iterative movements between communities, akin to the local-moving phase. At the onset of the refinement phase, the community membership of each vertex is reset so that each vertex initially forms its own community. The *leidenRefine()* function is employed, taking as input 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', changed communities flag vector $\Delta C'$, current tolerance/iteration τ, and returns iterations count l_j.

Lines 2-12 embody the central aspect of the refinement phase. During this phase, we execute what is termed the constrained merge procedure [8]. The essence of this procedure lies in enabling vertices, within each community boundary, to create sub-communities solely by permitting isolated vertices (i.e., vertices belonging to their own community) to alter their community affiliation. This process divides any internally-disconnected communities identified during the local-moving phase and prevents the emergence of new disconnected communities. Specifically, for every isolated vertex i (line 4), we explore communities connected to i within the *same community boundary* - excluding itself (line 5). The refinement phase is skipped for a vertex if its community has not been flagged as changed ($\Delta C'[c] = 0$), or if another vertex has joined its community, as indicated by the total community weight no longer matching the total edge weight of the vertex. Subsequently, we determine the optimal community $c*$ for relocating i (line 7), and assess the delta-modularity of transferring i to $c*$ (line 8). If a superior community is identified, we attempt to update the community affiliation of i provided it remains isolated (lines 10-12). Note that we do not migrate the vertex i to community c^* if the vertex representing c^* has itself moved to another community.

2) Aggregation phase of our Parallel Leiden: The pseudocode for the aggregation phase is presented in Algorithm 11, wherein communities are merged into super-vertices. Specifically, the *leidenAggregate()* function within this algorithm takes the current graph G' and the community membership

Algorithm 10 Refinement phase of our Parallel Leiden [12].

▷ $G'(V', E')$: Input/super-vertex graph
▷ C'_B: Community bound of each vertex
▷ C': Community membership of each vertex
▷ K': Total edge weight of each vertex
▷ Σ': Total edge weight of each community
▷ C'_Δ: Community changed flag
□ $G'_{C'}$: Community vertices (CSR)
□ H_t: Collision-free per-thread hashtable
□ τ: Per iteration tolerance

1: **function** $leidenRefine(G', C'_B, C', K', \Sigma', C'_\Delta, \tau)$
2: **for all** $i \in V'$ **in parallel do**
3: $c \leftarrow C'[i]$
4: **if** $C'_\Delta[c] = 0$ **or** $\Sigma'[c] \neq K'[i]$ **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** $c^* = c$ **or** $C'[c^*] \neq c^*$ **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** $pointer$

C' as inputs and produces the super-vertex graph G''.

In the algorithm, the process begins by obtaining the offsets array for the community vertices in the CSR format, denoted as $G'_{C'}.offsets$, within lines 3 to 4. This starts with counting the number of vertices in each community using *countCommunityVertices()*, followed by performing an exclusive scan on the resulting array. Next, within lines 5 to 6, we concurrently traverse all vertices, atomically placing the vertices associated with each community into the community graph CSR $G'_{C'}$. Following this, the offsets array for the super-vertex graph CSR is computed by estimating the degree of each super-vertex within lines 8 to 9. This involves calculating the total degree of each community using *communityTotalDegree()*, followed by another exclusive scan. As a result, the super-vertex graph CSR is organized with intervals for the edges and weights array of each super-vertex. Then, in lines 11 to 17, we iterate over all communities $c \in [0, |\Gamma|)$ in parallel using dynamic loop scheduling, with a chunk size of 2048 for both Static Leiden and a chunk size of 32 for ND, DS, and DF Leiden. During this phase, all communities d (and their respective edge weights w) connected to each vertex i in community c are included (via *scanCommunities()*, described in Algorithm 7) in the per-thread hashtable H_t. Once H_t contains all the connected communities and their weights, they 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''.

Algorithm 11 Aggregation phase of our Parallel Leiden [12].

▷ $G'(V', E')$: 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: **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''