International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 15, Number 1, pages 2–31, January 2025

Parallel Multicore Algorithms for Community Detection in Dynamic Graphs

Subhajit Sahu
International Institute of Information Technology Hyderabad
Hyderabad, Telangana 500032, India

Kishore Kothapalli
International Institute of Information Technology Hyderabad
Hyderabad, Telangana 500032, India

Dip Sankar Banerjee
Indian Institute of Technology Jodhpur
Karwar, Rajasthan 342030, India

Received: June 27, 2024
Revised: October 24, 2024
Accepted: November 25, 2024
Communicated by Susumu Matsumae

Abstract

Community detection is the problem of identifying natural divisions in networks. A relevant challenge in this problem is to find communities on rapidly evolving graphs. In this paper, we design efficient community detection algorithms in the batch dynamic setting. First, we present our parallel Dynamic Frontier approach. Given a batch update of edge deletions or insertions, this approach incrementally identifies an approximate set of affected vertices in the graph with minimal overhead. We apply this approach to both Louvain, a high quality, and Label Propagation Algorithm (LPA), a fast static community detection algorithm. Our approach achieves a mean speedup of $7.3\times$ and $6.7\times$, when applied to Louvain and LPA respectively, compared to our parallel and optimized implementation of Δ-screening, a recently proposed state-of-the-art approach. Finally, we show how to combine Louvain and LPA with the Dynamic Frontier approach to arrive at a hybrid algorithm. This algorithm produces high-quality communities while being $14.6\times$ faster than state-of-the-art, and identifying communities with the same quality score.[1]

Keywords: Dynamic graphs, Community detection, Parallel algorithms, Dynamic Frontier approach, Dynamic Louvain algorithm, Dynamic Label Propagation Algorithm (LPA), Dynamic Hybrid Louvain-LPA

1 Introduction

Graphs provide a powerful way to represent data and the relationships between them. In recent years, the use of graphs to model data and their connections has grown significantly. These graphs are often vast, driven by applications like machine learning and social networks.

Communities are groups of vertices that more strongly connected together to other vertices within their groups, than those outside. Finding communities in graphs, have extensive applications in recommendation

[1]This work is partially supported by a grant from the Department of Science and Technology (DST), India, under the National Supercomputing Mission (NSM) R&D in Exascale initiative vide Ref. No: DST/NSM/R&D_Exascale/2021/16; and from the Science & Engineering Research Board (SERB) of the Department of Science and Technology (DST), India, vide Project No: CRG/2023/005225.

systems, targeted advertising, drug discovery, protein annotation, and topic discovery [13]. Communities are *intrinsic* when defined solely by network topology and are *disjoint* when each vertex is assigned to only one community. Community detection algorithms assess the quality of identified communities using the modularity metric introduced by Newman et al. [28]. However, maximizing modularity to detect communities is an NP-hard problem [12]. One must therefore resort to heuristics, which obtain a solution that is close to the optimal one, in a reasonable amount of time.

Two popular heuristic-based algorithms for intrinsic and disjoint community detection are the Louvain method [4] and the Label Propagation Algorithm (LPA) [30]. Several recent studies show how to implement the *Louvain* method and *LPA* on modern architectures such as multi-core CPUs [10], GPUs [6], CPU-GPU hybrid platforms [3], distributed platforms [11], and the like.

However, with the data deluge and ever-changing application requirements, newer challenges are emerging. Many real-world graphs evolve with the insertion/deletion of edges/vertices. For efficiency reasons, one needs algorithms that update the results without re-computing from scratch — known as *dynamic algorithms*. Nonetheless, parallelizing dynamic graph algorithms is challenging due to the complexities of handling concurrency, optimizing data access, reducing resource contention, and load imbalance. Further, in the parallel setting, processing a batch of updates is often an effective method as doing so offers scope for exploiting parallelism and minimizes computational effort compared to processing individual updates. Given these theoretical and practical efficiency considerations, designing parallel batch dynamic graph algorithms is naturally more challenging. Examples of parallel dynamic graph algorithms include those for graph coloring [2], shortest paths [18], and centrality scores [32].

Dynamic community detection algorithms aim to obtain communities on evolving graphs while minimizing computation time. One does this usually by choosing a suitable algorithm, reusing old community labels of vertices, and processing a subset of the graph likely to be affected by changes. Ideally, the algorithm should identify a subset of the graph to be processed with a low overhead [31]. If the identified subset is too small, we may end up with inaccurate communities; if it is too large, we incur a significant computation time.

A critical examination of the extant literature on dynamic community detection algorithms indicates a few shortcomings. Some of these algorithms [8, 24] do not outperform static algorithms even for modest-sized batch updates. Aynaud et al. [1] and Chong et al. [7] adapt the existing community labels and run an algorithm, such as Louvain, on the entire graph. Often, this is unwarranted since not every vertex would need to change its community on the insertion/deletion of a few edges. Riedy et al. [33] and Cordeiro et al. [8] do not consider the cascading impact of changes in community labels, where the community label of a vertex changes because of a change in the community label of its neighbor. Zarayeneh et al. [42] and Riedy et al. [33] identify a subset of vertices whose community labels are likely to change on the insertion/deletion of a few edges. However, as this set of vertices identified is large, the algorithm of Zaranayeh et al. [42] incurs a significant computation time. In addition, most existing studies evaluate their algorithms on small graphs (with less than a million edges) [15, 22, 26, 37, 41], leading to potentially misleading conclusions. Moreover, many of the reported algorithms [15, 17, 22, 26, 37, 41–43] lack parallelism.

The above discussion, summarized in Table 1, motivates us to design efficient parallel algorithms that update the community structures of an evolving graph.

Property	[1, 7, 8, 22, 24]	[41–43]	[15]	[33]	**This paper**
Fully dynamic	✓	✓	✓	✓	✓
Batch update	✓	✓	✓	✓	✓
Process subset	×	✓	✓	✓	✓
Cascading updates	×	×	✓	×	✓
Parallel algorithm	×	×	×	✓	✓
Hybrid algorithm	×	×	×	×	✓

Table 1: Comparison of community detection papers.

1.1 Our Contributions

This paper addresses the design of efficient parallel algorithms in the *batch dynamic* setting, where multiple edge updates are processed simultaneously.

- We start by proposing our parallel *Dynamic Frontier* approach (Section 4.1), which we also refer to as Alg. P-DF. Given a batch update consisting of edge deletions or edge insertions, Alg. P-DF incrementally identifies an approximate set of affected vertices in the graph with a low run time overhead.
- Next, we show how to combine Alg. P-DF with two parallel algorithms: Parallel Louvain (Alg. Static_L) and Parallel LPA (Alg. Static_LPA), in Sections 4.3 and 4.4 respectively. We refer to these algorithms as Alg. P-DF_L and Alg. P-DF_LPA respectively. In addition to accepting the previous community membership of each vertex, our algorithms accept auxiliary information to improve scalability.
- We also show how to use Alg. P-DF under a combination of both the Louvain and LPA to arrive at a *hybrid* algorithm (Alg. P-DF_H), in Section 4.5.
- In Sections 5.1 and 5.2, we present the correctness arguments of Alg. P-DF_L and Alg. P-DF_LPA.
- The implementation details of Alg. Static_L and Alg. Static_LPA are discussed in Sections 6.1 and 6.2, respectively. This includes plots indicating how our static algorithms outperform well known implementations of the respective static algorithm.
- In Section 7, we compare our dynamic algorithms with our *custom parallel implementation* of the Δ-screening approach running on a 64-core AMD EPYC server. Table 2 shows the speedup obtained by our algorithms on a collection of eight graphs from four different classes. Our experimental results use our optimized parallel implementation of the Louvain and LPA. In addition, we show that our algorithms achieve good community stability, and have good scaling performance.[2]

Algorithm	Dynamic Frontier		
	+ *Louvain*	+ LPA	+ Hybrid
Speedup	7.3×	6.7×	14.6×
Modularity	0.90	0.78	0.90

Table 2: Average speedup compared to parallel Δ-screening algorithm [42], and the average modularity score achieved by our algorithms on a batch updates ranging from 10^{-7} to 0.01 times the number of edges in the original graph.

2 Related work

Louvain method is a popular and efficient algorithm to identify communities with a high modularity. As a result, it is widely favored by researchers [20, 23]. Existing works on Static Louvain algorithm propose a number of algorithmic and programming optimizations [11, 14, 25, 25, 34].

While the Louvain method obtains high-modularity communities, we find it to be $2.3 - 14\times$ slower than LPA (which obtains communities of lower modularity by $3.0 - 30\%$). LPA is faster than the Louvain algorithm, as it does not require repeated optimization steps and is easier to parallelize.

A growing number of research efforts have focused on detecting communities in *dynamic networks*. A core idea among most approaches is to use the community membership of each vertex from the previous snapshot of the graph instead of initializing each vertex into singleton communities [1, 7, 8, 33, 42]. Aynaud et al. [1] run the Louvain algorithm after assigning the community membership of each vertex as its previous community membership. Chong et al. [7] reset the community membership of vertices linked to an updated edge, in addition to the steps performed by Aynaud et al., and process all vertices with the Louvain algorithm. However, finding a subset of vertices that need to be processed can help minimize computation time.

[2] For reproducibility, our source code is at `https://github.com/merferry/communities-cpu--artifact`

Meng et al. [24] introduce a Dynamic Louvain algorithm aiming for temporally smoothed community structures in evolving graphs. They utilize an approximate delta-modularity optimization to mitigate abrupt changes in community structure but fail to outperform the Static Louvain algorithm, achieving lower modularity scores. This discrepancy is attributed to prioritizing temporal smoothing over modularity maximization. Cordeiro et al. [8] propose a similar dynamic algorithm focusing on tracking communities over time. Their method employs local modularity optimization for communities affected by edge and vertex changes, yielding modularity scores comparable to the Static Louvain algorithm but at the cost of slower performance, even for small batch updates.

The works of Kanezashi and Suzumura [17] and that of Nath and Roy [26] handle only edge insertions and do not handle deletion of edges. Xie et al. [41] present a stabilized process for community detection in dynamic graphs, based on LabelRank. However, they do not report runtime of their algorithm. We implemented LabelRank, and observed it to perform slower than LPA. Han et al. [15] propose ALPA, another dynamic approach, which first performs a warm-up *LPA* on a subset of the network, followed by expansion as a frontier of nodes that change labels. However, their algorithm is sequential. Liu et al. [22] propose DLPAE based on an improved version of their static algorithm. However, their evaluation is limited to small graphs, and they do not report runtimes of their algorithm. Sun et al. [37] propose DCDID, a dynamic community detection algorithm in the batch dynamic setting. However, their approach is sequential, and they test their algorithm only on small graphs.

Riedy et al. [33] adapt their original static agglomerative community detection algorithm to the dynamic setting by identifying vertices likely to change communities due to edge additions/deletions. However, they overlook cascading changes in community memberships, and do not employ *Louvain* or *LPA* algorithms — making direct comparison irrelevant.

Zarayeneh et al. [42] introduce *Delta-screening*, a method for updating communities in dynamic graphs. It 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. Despite possessing desirable properties for dynamic community detection, our observations reveal that Δ-screening tends to flag a significant number of vertices as affected, thus requiring a large amount of work to identify the new communities.

Most existing works mentioned above test their algorithms on small graphs (with less than a million edges) [15, 22, 26, 37, 41]. In addition, the dynamic algorithms mentioned above are *sequential* [1, 7, 8, 15, 17, 22, 24, 26, 37, 41, 42]. Thus, there is a need for efficient parallel algorithms for community detection on dynamic graphs. Further, none of the works recommend reusing the previous *weighted degree* of each vertex, and the *total community weight* (for local-moving phase of the Louvain algorithm) as auxiliary information to the dynamic algorithm. Recomputing it from scratch is expensive and becomes a bottleneck for Dynamic Louvain algorithm.

3 Preliminaries

Let $G(V, E, w)$ be an undirected graph, with V as the set of vertices, E as the set of edges, and $w_{ij} = w_{ji}$ a positive weight associated with each edge in the graph. If the graph is unweighted, we assume each edge to be associated with unit weight ($w_{ij} = 1$). 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 number of vertices and edges in the graph as $N = |V|$ and $M = |E|$ respectively, 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 is the process of arriving at 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. We denote the vertices of a community $c \in \Gamma$ as V_c. We denote the community that a vertex i belongs to as C_i. Further, we denote the neighbors of vertex i belonging to a community c as $J_{i \to c} = \{j \mid j \in J_i \; and \; C_j = c\}$, the sum of those edge weights as $K_{i \to c} = \sum_{j \in J_{i \to c}} w_{ij}$, the sum of edge weights within a community c as $\sigma_c = \sum_{(i,j) \in E \; and \; C_i = C_j = c} w_{ij}$, and the total edge weight of c as $\Sigma_c = \sum_{(i,j) \in E \text{ and } C_i = c} w_{ij}$ [42].

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

Like many authors, we use *modularity* Q to evaluate the quality of communities obtained, as shown in Eq. 1. It is equal to the fraction of edges that fall within the given "modules", minus the expected fraction if edges were uniformly distributed/assigned at random. It lies in the range $[-0.5, 1]$ and a higher value is better [5]. Optimizing this function theoretically leads to the best possible grouping [27, 38]. The *delta modularity* of moving a vertex i from the community d to the community c, denoted as $\Delta Q_{i:d \to c}$, can be calculated using Eq. 2. Here, $Q_{i:d \to i}$ indicates the change in modularity when vertex i is moved from community d to an isolated community, and $Q_{i:i \to c}$ represents the change in modularity when i is transferred from an isolated community into community c. Table 3 shows the commonly used notations in this paper.

$$\begin{aligned} \Delta Q_{i:d \to c} &= \Delta Q_{i:d \to i} + \Delta Q_{i:i \to c} \\ &= \left[\frac{\sigma_d - 2K_{i \to d}}{2m} - \left(\frac{\Sigma_d - K_i}{2m} \right)^2 \right] + \left[0 - \left(\frac{K_i}{2m} \right)^2 \right] - \left[\frac{\sigma_d}{2m} - \left(\frac{\Sigma_d}{2m} \right)^2 \right] \\ &+ \left[\frac{\sigma_c + 2K_{i \to c}}{2m} - \left(\frac{\Sigma_c + K_i}{2m} \right)^2 \right] - \left[\frac{\sigma_c}{2m} - \left(\frac{\Sigma_c}{2m} \right)^2 \right] - \left[0 - \left(\frac{K_i}{2m} \right)^2 \right] \\ &= \frac{1}{m}(K_{i \to c} - K_{i \to d}) - \frac{K_i}{2m^2}(K_i + \Sigma_c - \Sigma_d) \end{aligned} \tag{2}$$

Table 3: List of symbols, and their explanations.

Symbol	**Meaning**
$G^t(V^t, E^t)$	Current input graph
Δ^{t-}, Δ^{t+}	Edge deletions and insertions (batch update)
C^{t-1}	Previous community of each vertex
K^{t-1}	Previous weighted-degree of vertices
Σ^{t-1}	Previous total edge weight of communities
G'	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 total edge weight of each community in G^t, G'
τ, τ_{agg}	Iteration, aggregation tolerance
$\delta E, \delta C$	Are neighbors, or community affected?
H	Hashtable mapping a community to associated weight

3.2 Algorithms for Static Graphs

3.2.1 Louvain algorithm [4]

Louvain is a greedy, modularity optimization based agglomerative algorithm that finds high-modularity communities in a graph with a time complexity of $O(LM)$ and a space complexity of $O(N + M)$, where L is the total number of iterations performed across all passes [20]. It consists of two phases: the *local-moving phase*, where each vertex i greedily decides to move to the community of one of its neighbors $j \in J_i$ that gives the highest increase in modularity $\Delta Q_{i:C_i \to C_j}$ (using Eq. 2), and the *aggregation phase* which collapses all vertices in a community into a single super-vertex. These two phases make up one pass and repeat until there is no further increase in modularity. As a result, we have a hierarchy of community memberships for each vertex as a dendrogram. The top-level hierarchy is the final result. We use an efficient parallel implementation of Louvain, detailed in Section 6.1, which we refer as Alg. Static_L

3.2.2 Label Propagation Algorithm (LPA) [30]

LPA is a popular diffusion-based method for finding communities that is simpler, faster, and more scalable (due to its lower memory footprint) than Louvain. In LPA, every vertex i is initialized with a unique label, or community id, C_i. Each vertex adopts the label with the most interconnecting weight at every step, as shown in Eq. 3. In the equation, $K_{i \to c}$ denotes the total interconnecting weight of each label, and $\underset{c \in \Gamma}{\arg\max}$ denotes the label with the most interconnecting weight. Through this iterative process, densely connected groups of vertices form a consensus on a unique label that corresponds to a community. For a given tolerance parameter τ, the algorithm converges when at least $1 - \tau$ fraction of vertices do not change their community membership. LPA has a time complexity of $O(LM)$ and a space complexity of $O(N + M)$, where L is the number of iterations performed. We present an efficient parallel implementation of LPA in Section 6.2, referred to here as Alg. $\text{Static}_{\text{LPA}}$.

$$C_i = \underset{c \in \Gamma}{\arg\max}\, K_{i \to c} \tag{3}$$

3.3 Dynamic Graphs

A dynamic graph is a sequence of graphs, where $G^t(V^t, E^t, w^t)$ denotes the graph at time step t with $t \geq 0$. The graph G^0 is the *base graph*. We consider only changes to the edges of the graph over time. We use Δ^t to denote the changes to the edges of 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. The set Δ^t consists of a set of edge deletions $\Delta^{t-} = E^{t-1} \setminus E^t$ and a set of edge insertions $\Delta^{t+} = E^t \setminus E^{t-1}$. Thus, $\Delta^t = \Delta^{t-} \cup \Delta^{t+}$. We refer to the setting where Δ^t consists of multiple edges deleted and inserted as a batch update.

For simplicity, we consider a batch of edges where all edges in the batch are being inserted into G^t – in which case $\Delta^t = E^t - E^{t-1}$, or the case where all edges in the batch are being deleted from G^t where $\Delta^t = E^{t-1} - E^t$. In our experiments, we let G^0 be a non-empty graph and run a static algorithm to identify the community labels of each vertex in G^0.

3.4 Δ-screening approach [42] for Dynamic Graphs

Δ-*screening* uses modularity to determine an approximate graph region where vertices are likely to change their community membership. Here, Zarayeneh et al. first separately sort the batch update consisting of edge deletions $(i, j) \in \Delta^{t-}$ and insertions $(i, j, w) \in \Delta^{t+}$ by their source vertex-id. For edge deletions within the same community, they mark i's neighbors and j's community as affected. For edge insertions across communities, they pick vertex $j*$ with the highest change in modularity among all insertions linked to vertex i and mark i's neighbors and $j*$'s community as affected. Edge deletions between different communities and edge insertions between the same community are *unlikely* to affect the community membership of any vertex and hence ignored.

The affected vertices identified by Δ-*screening* are processed in the first pass of Louvain algorithm (for the remaining passes, all vertices are processed), and the community membership of each vertex, C^t, is initialized at the start of the algorithm to that obtained in previous snapshot of the graph, C^{t-1}, where $t \geq 1$.

The Δ-*screening* technique, as proposed in [42], is not a parallel algorithm. We redesign it as a multicore parallel algorithm. To this end, we go through sorted edge deletions and insertions in parallel, apply Δ-*screening* as mentioned above, and mark vertices, neighbors of a vertex, and the community of a vertex using three separate flag vectors. Finally, we use the neighbors and community flag vectors to mark affected vertices. For this, we use per-thread collision-free hash tables [34]. Every hash table comprises a vector of keys, a vector of values (of size $|V|$), and a key count. The value corresponding to each key is stored or accumulated at the index indicated by the key. To prevent false cache sharing, we allocate each key count separately on the heap, as they are updated independently. Each hashtable is allocated separately, in order to avoid false sharing between CPU caches. We further optimize it by taking as input the previous weighted degree of each vertex K_i and total edge weight of each community Σ_c, and incrementally update them based on the batch update instead of recomputing from scratch which is costly. We refer to this parallel version of

Algorithm 1 Our *Parallel* Δ*-screening* approach (Alg. P-DS).

```
▷ G^t(V^t, E^t): Current input graph                                ▷ Inputs ↓
▷ Δ^{t-}, Δ^{t+}: Edge deletions and insertions (batch update)
▷ C^{t-1}: Previous community of each vertex
□ δE, δC: Are neighbors, or community affected?                     □ Variables ↓
□ H: Hashtable mapping a community to associated weight

 1: ▷ Step 1: Marking of affected vertices
 2: function P-DS(G^t, Δ^{t-}, Δ^{t+}, C^{t-1})
 3:     H, δE, δC ← {}
 4:     for all [i, j] ∈ Δ^{t-} in parallel do
 5:         if C^{t-1}[i] = C^{t-1}[j] then
 6:             δE[i], δC[C^{t-1}[j]] ← true
 7:             Mark i as affected
 8:     for all unique source vertex i ∈ Δ^{t+} in parallel do
 9:         H ← {}
10:         for all (i', j, w) ∈ Δ^{t+} | i' = i do
11:             if C^{t-1}[i] ≠ C^{t-1}[j] then
12:                 H[C^{t-1}[j]] ← H[C^{t-1}[j]] + w
13:         [c*, w*] ← chooseCommunity(H)
14:         δE[i], δC[c*] ← true
15:         Mark i as affected
16:     for all i ∈ V^t in parallel do
17:         if δE[i] then
18:             for all j ∈ G^t.neighbors(i) do
19:                 Mark i as affected
20:         if δC[C^{t-1}[i]] then
21:             Mark i as affected

22: ▷ Step 2: Processing of affected vertices
23: function P-DS-PROCESS(G^t)
24:     while iterations are not complete do
25:         for all i ∈ V^t do
26:             if i is not affected then continue
27:             Mark i as not affected (prune)
28:             Pick best community for i
29:         repeat until communities have converged

30: ▷ Step 3: Algorithm-specific post-processing (optional)
```

Δ*-screening* as Alg. P-DS. Its pseudocode is shown in Algorithm 1. We apply Alg. P-DS to Louvain and LPA, and refer to them as Alg. P-DS$_{\text{L}}$ and Alg. P-DS$_{\text{LPA}}$, respectively.

4 Approach

Given a batch update on the original graph, it is likely that only a small subset of vertices in the graph would change their community membership. Selection of the appropriate set of affected vertices to be processed — that are likely to change their community — in addition to the overhead of finding them, plays a significant role in the overall accuracy and efficiency of a dynamic batch parallel algorithm. Too small a subset may result in poor-quality communities, while a too-large subset will increase computation time. However, Alg. P-DS generally overestimates the set of affected vertices and has a high overhead. Our proposed *Dynamic*

Frontier approach (which we from here on refer to as Alg. P-DF) addresses these issues.

4.1 Our Dynamic Frontier approach (Alg. P-DF)

We now explain our parallel *Dynamic Frontier* approach (Alg. P-DF), given in Algorithm 2. Consider a batch update consisting of either edge deletions $(i, j) \in \Delta^{t-}$ or edge insertions $(i, j, w) \in \Delta^{t+}$. We consider that batch updates are undirected, i.e., if the edge (i, j, w) is in Δ^{t+}, so is the edge (j, i, w). At the start, we initialize the membership of each vertex, C^t, to that obtained in the previous snapshot of the graph C^{t-1}.

Algorithm 2 *Parallel Dynamic Frontier* approach (Alg. P-DF).

```
▷ G^t(V^t, E^t): Current input graph                                ▷ Inputs ↓
▷ Δ^{t−}, Δ^{t+}: Edge deletions and insertions (batch update)
▷ C^{t−1}: Previous community of each vertex

 1: ▷ Step 1: Initial marking of affected vertices
 2: function P-DF-INITIAL(G^t, Δ^{t−}, Δ^{t+}, C^{t−1})
 3:     for all (i, j) ∈ Δ^{t−} in parallel do
 4:         Mark i as affected if C^{t−1}[i] = C^{t−1}[j]
 5:     for all (i, j, w) ∈ Δ^{t+} in parallel do
 6:         Mark i as affected if C^{t−1}[i] ≠ C^{t−1}[j]

 7: ▷ Step 2: Incremental marking of affected vertices
 8: function P-DF-INCREMENTAL(G^t)
 9:     while iterations are not complete do
10:         for all i ∈ V^t do
11:             if i is not affected then continue
12:             Mark i as not affected (prune)
13:             Pick best community for i
14:             if community of i changes then
15:                 Mark neighbors of i as affected
16:         repeat until communities have converged

17: ▷ Step 3: Algorithm-specific post-processing (optional)
```

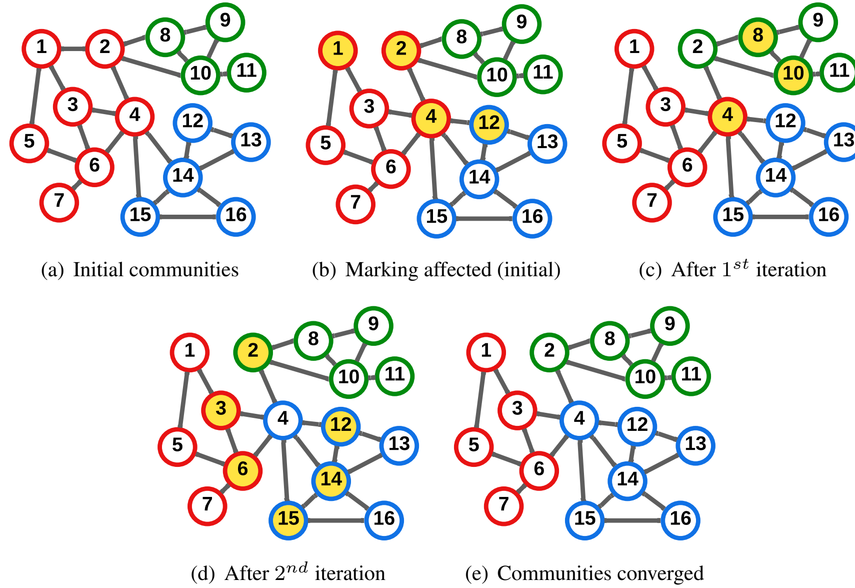
Alg. P-DF has two main steps, as show in Algorithm 2. *Step 1* marks an initial set of vertices as affected. If Δ^t is a batch of edge insertions Δ^{t+}, for each edge $(i, j, w) \in \Delta^{t+}$, we mark i and j as affected, provided i and j have different community labels — ignoring edge insertions within the same community. Similarly, suppose Δ^t is a batch of edge deletions Δ^{t-}, for each edge $(i, j) \in \Delta^{t-}$. In that case, we mark i and j as affected, provided i and j belong to the same community, ignoring edges deletions across distinct communities. Note that the edges we ignore are unlikely to impact the community structure of the graph, as Zarayeneh et al. [42] also observe. Alg. P-DF is thus an approximate algorithm for dynamic community detection, similar to Alg. P-DS.

In *Step 2*, the community membership of each vertex, obtained while running a community detection algorithm, is used by Alg. P-DF to update the set of affected vertices, as follows, incrementally. If the community label of an affected vertex i changes, then the neighbors of i are marked as affected. Once the community detection algorithm has processed a vertex, we mark it as unaffected, to minimize unnecessary computation. We call this the vertex pruning optimization. Subsequently, the community detection algorithm continues to execute to identify the community labels of the affected vertices. This process continues until the algorithm converges. Finally, in *Step 3*, depending on the community detection algorithm used, any necessary post-processing steps are performed as needed.

Application to the first pass of Louvain algorithm We apply Alg. P-DF to the first pass of *Louvain* algorithm, as with Alg. P-DS. In subsequent passes, if the aggregation tolerance condition is not met (Line 14 in Algorithm 3), all super-vertices are marked as affected and processed according to Louvain. This takes less than 14% of total time, so we do not use Alg. P-DF to find affected super-vertices. The tolerance condition only fails in the case of large batch updates.

4.2 An Example of Dynamic Frontier approach (Alg. P-DF)

Figure 1 shows an example of Alg. P-DF.



(a) Initial communities (b) Marking affected (initial) (c) After 1st iteration

(d) After 2nd iteration (e) Communities converged

Figure 1: In this example of the Dynamic Frontier approach (Alg. P-DF), vertex community membership is represented by border colors (red, green, or blue), with the algorithm progressing from left to right. A batch update arrives, affecting vertices 1, 2, 4, and 12. In the first iteration, vertex 2 switches from red to green, impacting neighbors 4, 8, and 10. In the second iteration, vertex 4 changes from red to blue, affecting neighbors 2, 3, 6, 12, 14, and 15. Afterward, there are no more community changes.

Initial communities The original graph, shown in Figure 1(a) comprises a total of 16 vertices divided into three communities distinguished by the border colors of *red*, *green*, and *blue*. We consider a batch of edge insertions consisting of adding two edges to the original graph.

Batch update and Marking affected (initial) Subsequently, Figure 1(b) shows a batch update applied to the original graph involving the deletion of one edge between vertices 1 and 2, and the insertion of one edge between vertices 4 and 12. To process this batch update, we perform the initial step of Alg. P-DF, marking endpoints 1, 2, 4, and 12 as affected. At this point, we are ready to execute the first iteration of a community detection algorithm.

After first iteration During the first iteration (see Figure 1(c)), suppose that the community membership of vertex 2 changes from *red* to *green* because it exhibits stronger connections with vertices in the *green* community. In response to this change, the Alg. P-DF incrementally marks the neighbors of 2 as affected, specifically vertices 4, 8 and 10. Vertex 2 is no longer marked as affected due to the pruning optimization.

After second iteration During the second iteration (see Figure 1(d)), suppose that vertex 4 is now more strongly connected to the *blue* community, resulting in a change of its community membership from *red* to

blue. As before, we mark the neighbors of vertex 4 as affected, namely vertices 2, 3, 6, 12, 14, and 15. However, vertex 4 is no longer marked as affected due to vertex pruning.

Communities converged In the subsequent iteration (see Figure 1(e)), suppose that no other vertices have a strong enough reason to change their community membership. At this point, the post-processing step is invoked and we obtain the updated community labels — i.e., when employing *Louvain*, the aggregation phase commences consolidating communities into super-vertices to prepare for the subsequent pass of the algorithm. However, when employing the *LPA*, this marks the conclusion of the algorithm.

4.3 Our *Dynamic Frontier* based *Louvain* (Alg. P-DF$_\mathrm{L}$)

We now show how to apply the *Dynamic Frontier* approach (Alg. P-DF) to *Louvain* in Algorithm 3, which we call Alg. P-DF$_\mathrm{L}$. We take as input the previous snapshot of the graph G^{t-1}, the batch update consisting of edge deletions Δ^{t-} and insertions Δ^{t+}, the previous community membership of each vertex C^{t-1}, the previous weighted degree of each vertex K^{t-1}, and the previous total edge weight of each community Σ^{t-1} in Line 1. By using K^{t-1} and Σ^{t-1} of the previous graph G^{t-1} as auxiliary information to the dynamic algorithm, we are able to quickly obtain the updated K^t and Σ^t. To the best of our knowledge, none of the existing dynamic algorithms for community detection make use of such auxiliary information.

Initial marking phase (Line 3): First, based on Alg. P-DF, we mark the initial set of vertices as affected.

Initialization phase (Lines 5-8): We initialize the community membership C of each vertex, obtain the updated vertex weights K and community weights Σ, and get the graph G', community membership C', vertex weights K', and community weights Σ' at the current pass.

Local-moving and aggregation phases (Lines 10-20): For each pass, we perform the *local-moving* phase of *Louvain* in Line 11. If the community labels converge after one iteration, we terminate the algorithm in Line 12. In Line 14, we check if only a small fraction of communities merged. We calculate the ratio $|\Gamma|/|\Gamma_{old}|$ of current to original number of communities. If it is below an aggregation tolerance, τ_{agg} (optimal value in [34]), we avoid the expensive *aggregation* phase, as it does not provide a noticeable benefit.

In Line 15, we renumber the community-ids. This renumbering helps generate the aggregated graph G' in the Compressed Sparse Row (CSR) format. In Line 16, we update the community membership of each vertex C based on the community membership of each super-vertex, in order to obtain the top-level hierarchy of the dendrogram as the final result. In Line 17, we proceed with aggregation, storing the result as graph G'.

Once the graph has been aggregated, we obtain the super-vertex weights K' in Line 18. In the aggregated graph, each vertex belongs to its own singleton community. Hence, the super-community weights Σ' are the same as K'. Next, in Line 19, we mark all super-vertices as affected, and initialize the community membership of each super-vertex. In Line 20, we perform the threshold scaling optimization [25], i.e., we reduce the tolerance τ using a factor `TOLERANCE_DROP`. This reduces local-moving phase iterations, enhancing performance with minimal impact on community modularity. Once all necessary passes are complete, we perform the dendrogram flattening in Line 21. Finally, in Line 22, we return the membership of each vertex C, and the weighted degree of each vertex K and community Σ in the updated graph as auxiliary information.

Explanation of LOUVAINMOVE *(Lines 23-35)*: We now discuss the *local-moving* phase of Alg. P-DF$_\mathrm{L}$. For each iteration (Lines 24-33), and for each affected vertex i in the graph G', we mark i as not affected, for the next iteration, as vertex pruning step of Alg. P-DF (Line 27), and use per-thread collision-free hashtables to obtain the best community c^* linked to each vertex, as well as the associated delta-modularity (highest) δQ^* using Eq. 2 (Lines 28-29). If the best community c^* is different from the original community membership $C'[i]$ of vertex i (Line 30), we update the community membership of the vertex C' and atomically update the total edge weights linked to each community Σ' in Lines 31-32. If c^* is no longer the best choice, vertex i will be re-processed in the next iteration. At the end of each iteration, if the total delta-modularity across all vertices ΔQ is less than the specified tolerance τ, we terminate the local-moving phase (Line 34) and returns the number of iterations performed. The algorithm stops when convergence is reached, or when the number of iterations reaches $MAX_ITERATIONS$.[3]

[3]In all of our experiments, we notice that the algorithm stops by convergence rather than by reaching $MAX_ITERATIONS$.

Algorithm 3 *Dynamic Frontier Louvain* (Alg. P-DF$_\text{L}$).

▷ $G^t(V^t, E^t)$: Current input graph ▷ Inputs ↓
▷ Δ^{t-}, Δ^{t+}: Edge deletions and insertions (batch update)
▷ C^{t-1}: Previous community of each vertex
▷ K^{t-1}: Previous weighted-degree of vertices
▷ Σ^{t-1}: Previous total edge weight of communities
□ G': Current/super-vertex graph □ Variables ↓
□ 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 total edge weight of each community in G^t, G'
□ τ, τ_{agg}: Iteration, aggregation tolerance

1: **function** P-DF$_\text{L}$($G^t, \Delta^{t-}, \Delta^{t+}, C^{t-1}, K^{t-1}, \Sigma^{t-1}$)
2: ▷ Initial marking phase
3: P-DF-INITIAL($G^t, \Delta^{t-}, \Delta^{t+}, C^{t-1}$) ▷ See Alg. 2
4: ▷ Initialization phase
5: Vertex membership: $C \leftarrow [0..|V^t|)$
6: $K \leftarrow vertexWeights(K^{t-1}, \Delta^{t-}, \Delta^{t+})$
7: $\Sigma \leftarrow communityWeights(\Sigma^{t-1}, \Delta^{t-}, \Delta^{t+}, C^{t-1})$
8: $G' \leftarrow G^t$ **;** $C' \leftarrow C^{t-1}$ **;** $K' \leftarrow K$ **;** $\Sigma' \leftarrow \Sigma$
9: ▷ Local-moving and aggregation phases
10: **for all** $l_p \in$ [0..MAX_PASSES) **do**
11: $numIters \leftarrow louvainMove(G', C', K', \Sigma')$
12: **if** $numIters \leq 1$ **then break** ▷ Globally converged?
13: $|\Gamma|, |\Gamma_{old}| \leftarrow$ Number of communities in C, C'
14: **if** $|\Gamma|/|\Gamma_{old}| > \tau_{agg}$ **then break** ▷ Low shrink?
15: $C' \leftarrow$ Renumber communities in C'
16: $C \leftarrow$ Lookup dendrogram using C to C'
17: $G' \leftarrow$ Aggregate communities in G' using C'
18: $\Sigma' \leftarrow K' \leftarrow vertexWeights(G')$
19: Mark all vertices in G' as affected **;** $C' \leftarrow [0..|V'|)$
20: $\tau \leftarrow \tau/$TOLERANCE_DROP ▷ Threshold scaling
21: $C \leftarrow$ Lookup dendrogram using C to C'
22: **return** $C^{t-1} \leftarrow C, K^{t-1} \leftarrow K, \Sigma^{t-1} \leftarrow \Sigma$

23: **function** LOUVAINMOVE(G', C', K', Σ')
24: **for all** $l_s \in$ [0..MAX_ITERATIONS) **do**
25: Delta modularity: $\Delta Q \leftarrow 0$
26: **for all** affected $i \in V'$ **in parallel do**
27: Mark i as not affected (prune)
28: $c^* \leftarrow$ Best community linked to i in G'
29: $\delta Q^* \leftarrow$ Delta-modularity of moving i to c^*
30: **if** $c^* = C'[i]$ **then continue**
31: $\Sigma'[C'[i]] -= K'[i]$ **;** $\Sigma'[c^*] += K'[i]$ **atomic**
32: $C'[i] \leftarrow c^*$ **;** $\Delta Q \leftarrow \Delta Q + \delta Q^*$
33: Mark neighbors of i as affected
34: **if** $\Delta Q \leq \tau$ **then break** ▷ Locally converged?
35: **return** l_s

Application for Alg. P-DF to first pass of Louvain in Alg. P-DF$_L$: Note that we apply Alg. P-DF only to the first pass of *Louvain*, as with Alg. P-DS. In subsequent passes, if the aggregation tolerance τ_{agg} condition is not met, all super-vertices are marked as affected and processed. This takes less than 14% of total time, so we do not use Alg. P-DF to find affected super-vertices. The τ_{agg} condition only fails with large updates.

4.4 Our *Dynamic Frontier* based LPA (Alg. P-DF$_{\text{LPA}}$)

We now show how to apply Alg. P-DF to *LPA* in Algorithm 4, which we call Alg. P-DF$_{\text{LPA}}$. Here, We take as input the previous snapshot of the graph G^{t-1}, the batch update consisting of edge deletions Δ^{t-} and insertions Δ^{t+}, and the previous community membership of each vertex C^{t-1} in Line 1.

Algorithm 4 *Dynamic Frontier LPA* (Alg. P-DF$_{\text{LPA}}$).

```
▷ G^t(V^t, E^t): Current input graph                                    ▷ Inputs ↓
▷ Δ^{t-}, Δ^{t+}: Edge deletions and insertions (batch update)
▷ C^{t-1}: Previous community of each vertex
□ G′: Current graph                                                     □ Variables ↓
□ C′: Current community of each vertex in G′
□ τ: Iteration tolerance

 1: function P-DF_LPA(G^t, Δ^{t-}, Δ^{t+}, C^{t-1})
 2:     P-DF-INITIAL(G^t, Δ^{t-}, Δ^{t+}, C^{t-1})                      ▷ See Alg. 2
 3:     Vertex membership: C′ ← C^{t-1} ; G′ ← G^t
 4:     for all l_s ∈ [0..MAX_ITERATIONS) do
 5:         ΔN ← lpaMove(G′, C′)
 6:         if ΔN/N ≤ τ then break                                      ▷ Converged?
 7:     return C^{t-1} ← C′

 8: function LPAMOVE(G′, C′)
 9:     Changed vertices: ΔN ← 0
10:     for all affected i ∈ V′ in parallel do
11:         Mark i as not affected (prune)
12:         c* ← Most weighted label to i in G′
13:         if c* = C′[i] then continue
14:         C′[i] ← c* ; ΔN ← ΔN + 1
15:         Mark neighbors of i as affected
16:     return ΔN
```

First, in Line 2, based on Alg. P-DF, we mark the initial set of vertices as affected. In Line 3, we initialize the community membership of each vertex C'. For each iteration (Lines 4-6), we perform the *label-propagation* step of *LPA* in Line 5. If only a small fraction of vertices changed their community membership, we recognize that the communities have converged and hence end the algorithm (Line 6).

Explanation of LPAMOVE *(Lines 8-16)*: In the *label diffusion* step of *LPA*, for each affected vertex i in the graph G', we mark i as not affected, for the next iteration, as vertex pruning step of Alg. P-DF (Line 11), and use a per-thread collision-free hash table to identify the label, c^*, of the maximum weight in parallel using Eq. 3 (Line 12). If this label is different from the original label $C'[i]$ of vertex i (Line 13), we update the label associated with vertex i in Line 14. In addition, based on Alg. P-DF, we mark neighbors of vertex i as affected (Line 15).

4.5 Our Dynamic Frontier Hybrid Louvain-LPA (Alg. P-DF$_{\text{H}}$)

Louvain is known for its high-modularity community detection but at the cost of being slow. On the other hand, *LPA* is fast, but the communities it detects are of moderate modularity [16]. We combine the strengths of both algorithms by creating a Hybrid dynamic algorithm, which we call Alg. P-DF$_{\text{H}}$. To this end, we use

Louvain as the base/static method and *LPA* as the dynamic method (i.e., Alg. P-DF$_\text{H}$), as shown in Algorithm 5. This provides us with superior performance, while obtaining communities with high modularity score.

Algorithm 5 *Dynamic Frontier Hybrid Louvain-LPA* (Alg. P-DF$_\text{H}$).

▷ $G^t(V^t, E^t)$: Current input graph ▷ Inputs ↓
▷ Δ^{t-}, Δ^{t+}: Edge deletions and insertions (batch update)
▷ C^{t-1}: Previous community of each vertex

1: **function** P-DF$_\text{H}$($G^t, \Delta^{t-}, \Delta^{t+}, C^{t-1}$)
2:     **if** $t = 0$ **then return** Alg. Static$_\text{L}(G^t)$
3:     **return** Alg. P-DF$_\text{LPA}(G^t, \Delta^{t-}, \Delta^{t+}, C^{t-1})$

4.6 Time and Space complexity

To analyze the time complexity of our algorithms, we use N_B to denote the number of vertices marked as affected (which is dependent on the size and nature of batch update) by the dynamic algorithm on a batch B of edge updates, use M_B to denote the number of edges with one endpoint in N_B, and L to denote the total number of iterations performed. Then the time complexity of Algorithms 3-5 is $O(LM_B)$. In the worst case, the time complexity of our algorithms would be the same as that of the respective static algorithms, i.e., $O(LM)$. The space complexity of our algorithms is the same as that of the static algorithms, i.e., $O(N+M)$.

5 Correctness

5.1 Correctness of Dynamic Frontier based Louvain (Alg. P-DF$_\text{L}$)

Given a batch update consisting of edge deletions Δ^{t-} and insertions Δ^{t+}, we now show that Alg. P-DF$_\text{L}$ marks the essential vertices, which have an incentive to change their community membership, as affected. For any given vertex i in the original graph (before the batch update), the delta-modularity of moving it from its current community d to a new community c is given by Equation 4 — as detailed earlier in Section 3.1. We now consider the direct effect of each individual edge deletion (i, j) or insertion (i, j, w) in the batch update, on the delta-modularity of a vertex, as well as the indirect cascading effect of migration of a vertex to another community.

$$\Delta Q_{i:d\to c} = \frac{1}{m}(K_{i\to c} - K_{i\to d}) - \frac{K_i}{2m^2}(K_i + \Sigma_c - \Sigma_d) \tag{4}$$

5.1.1 On edge deletion

Lemma 5.1. *Given an edge deletion (i, j) between vertices i and j belonging to the same community d, vertex i (and j) should be marked as affected.*

Consider the case of edge deletion (i, j) of weight w between vertices i and j belonging to the same community $C_i = C_j = d$ (see Figure 2, where $j = i_1$ or i_2). Let i'' be a vertex belonging i's community $C_{i''} = d$, and let k'' be a vertex belonging to another community $C_{k''} = b$. As shown below in Case **(1)**, the delta-modularity of vertex i moving from its original community d to another community b after edge deletion (i, j), $\Delta Q^{new}_{i:d\to b}$, has a significant positive factor w/m (indicated with square brackets). Note that $\Delta Q_{i:d\to b}$ represents the delta-modularity of vertex i moving from community d to community b before the edge deletion. There is thus a chance that vertex i would change its community membership, and we should mark it as affected. The same argument applies for vertex j, as the edge is undirected. On the other hand, for the Cases **(2)**-**(3)**, there is only a small positive change in delta-modularity for vertex k''. Thus, there is little incentive for vertex k'' to change its membership, and no incentive for a change in membership of vertex i''.

Note that it is possible that the community d would split due to the edge deletion. However, this is unlikely, given that one would need a large number of edge deletions between vertices belonging to the same community for the community to split. One can take care of such rare events by running Alg. Static$_\text{L}$ every 1000 batch updates, which also helps us ensure high-quality communities. The same applies to Alg. P-DS$_\text{L}$.

$$
\begin{aligned}
\Delta Q^{new}_{i:d\to b} &= \frac{K_{i\to b} - (K_{i\to d} - w)}{m} - \frac{K_i - w}{2m^2}\left((K_i - w) + \Sigma_b - (\Sigma_d - 2w)\right) \\
&= \frac{K_{i\to b} - K_{i\to d}}{m} - \frac{K_i}{2m^2}\left(K_i + \Sigma_b - \Sigma_d\right) + \frac{w}{m} + \frac{w}{2m^2}(\Sigma_b - \Sigma_d + w) \\
&= \Delta Q_{i:d\to b} + [\frac{w}{m}] + \frac{w}{2m^2}(\Sigma_b - \Sigma_d + w)
\end{aligned}
$$

Case 1. $\Delta Q^{new}_{i:d\to b} = \Delta Q_{i:d\to b} + [\frac{w}{m}] + \frac{w}{2m^2}(\Sigma_b - \Sigma_d + w)$ (see steps above)

Case 2. $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} - \frac{wK_{i''}}{m^2}$

Case 3. $\Delta Q^{new}_{k'':b\to d} = \Delta Q_{k'':b\to d} + \frac{wK_{k''}}{m^2}$

Now, consider the case of edge deletion (i, j) between vertices i and j belonging to different communities, i.e., $C_i = d$, $C_j = e$ (see Figure 2, where $j = j_1$). Let i'' be a vertex belonging to i's community $C_{i''} = d$, j'' be a vertex belonging to j's community $C_{j''} = e$, and k'' be a vertex belonging another community $C_{k''} = b$. As shown in Cases **(4)**-**(8)**, due to the absence of any significant positive change in delta-modularity, there is little to no incentive for vertices i, j, k'', i'', and j'' to change their community membership.

Case 4. $\Delta Q^{new}_{i:d\to e} = \Delta Q_{i:d\to e} - \frac{w}{m} + \frac{w}{2m^2}(2K_i + \Sigma_e - \Sigma_d - w)$

Case 5. $\Delta Q^{new}_{i:d\to b} = \Delta Q_{i:d\to b} + \frac{w}{2m^2}(K_i + \Sigma_b - \Sigma_d)$

Case 6. $\Delta Q^{new}_{i'':d\to e} = \Delta Q_{i'':d\to e}$

Case 7. $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} - \frac{wK_{i''}}{2m^2}$

Case 8. $\Delta Q^{new}_{k'':b\to d/e} = \Delta Q_{k'':b\to d/e} + \frac{wK_{k''}}{m^2}$ ◇

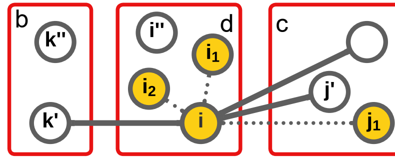


Figure 2: Processing edge deletions in the *Dynamic Frontier (DF)* approach. Here, pre-existing edges are shown by solid lines, while deletions are marked by dashed lines. Vertex i is the source of edge deletions. Vertices i_1, i_2, and j_1 are the destination vertices for deleted edges. Vertices j' and k' are neighbors of i, while i'' and k'' are non-neighbors. Vertices marked as affected, initially, are indicated with a yellow highlight. Communities are indicated with labels b, e, and d.

5.1.2 On edge insertion

Lemma 5.2. *Given an edge insertion (i, j, w) between vertices i and j belonging to different communities d and e, vertex i (and j) should be marked as affected.*

Let us consider the case of edge insertion (i, j, w) between vertices i and j belonging to different communities $C_i = d$ and $C_j = e$ respectively (see Figure 3, where $j = j_1$ or j_2). Let i'' be a vertex belonging i's community $C_{i''} = d$, j'' be a vertex belonging to j's community $C_{j''} = e$, and k'' be a vertex belonging to another community $C_{k''} = b$. As shown below in Case **(9)**, we have a significant positive factor w/m (and a small negative factor) which increases the delta-modularity of vertex i moving to j's community after the

insertion of the edge (i, j). There is, therefore, incentive for vertex i to change its community membership. Accordingly, we mark i as affected. Again, the same argument applies for vertex j, as the edge is undirected. Further, we observe from other Cases (**(10)**-**(13)**) there is only a small change in delta-modularity. Thus, there is hardly any to no incentive for a change in community membership of vertices i'', j'', and k''.

Case 9. $\Delta Q^{new}_{i:d\to e} = \Delta Q_{i:d\to e} + [\frac{w}{m}] - \frac{w}{2m^2}(2K_i + \Sigma_e - \Sigma_d + w)$

Case 10. $\Delta Q^{new}_{i:d\to b} = \Delta Q_{i:d\to b} - \frac{w}{2m^2}(K_i + \Sigma_b - \Sigma_d)$

Case 11. $\Delta Q^{new}_{i'':d\to e} = \Delta Q_{i'':d\to e}$

Case 12. $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} + \frac{wK_{i''}}{2m^2}$

Case 13. $\Delta Q^{new}_{k'':b\to d/e} = \Delta Q_{k'':b\to d/e} - \frac{wK_{k''}}{2m^2}$

Now, consider the case of edge insertion (i, j, w) between vertices i and j belonging to the same community $C_i = C_j = d$ (see Figure 3, where $j = i_1$). From Cases **(14)**-**(16)**, we note that it is little to no incentive for vertices i'', k'', i, and j to change their community membership. Note that it is possible for the insertion of edges within the same community to cause it to split into two or more strongly connected communities, but it is very unlikely.

Case 14. $\Delta Q^{new}_{i:d\to b} = \Delta Q_{i:d\to b} - \frac{w}{m} - \frac{w}{2m^2}(\Sigma_b - \Sigma_d - w)$

Case 15. $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} + \frac{wK_{i''}}{m^2}$

Case 16. $\Delta Q^{new}_{k'':b\to d} = \Delta Q_{k'':b\to d} - \frac{wK_{k''}}{m^2}$ ◇

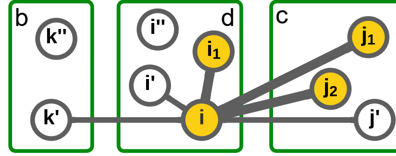


Figure 3: Processing edge insertions in the *Dynamic Frontier (DF)* approach. Here, pre-existing edges are represented by solid lines, and i represents a source vertex of edge insertions in the batch update. Edge insertions in the batch update with i as the source vertex are denoted by thickened lines. Vertices i_1, j_1, and j_2 represent the destination vertices of edge insertions. Vertices i', j', and k' signify neighboring vertices of vertex i. Finally, vertices i'' and k'' represent non-neighbor vertices (to vertex i). Vertices marked as affected, initially, are indicated with a yellow highlight. Communities are indicated with labels b, c, and d.

5.1.3 On vertex migration to another community

Lemma 5.3. *When a vertex i changes its community membership, and vertex j is its neighbor, j should be marked as affected.*

We considered the direct effects of deletion and insertion of edges above. Now we consider its indirect effects by studying the impact of change in community membership of one vertex on the other vertices. Consider the case where a vertex i changes its community membership from its previous community d to a new community e (see Figure 4). Let i' be a neighbor of i and i'' be a non-neighbor of i belonging to i's previous community $C_{i'} = C_{i''} = d$, j' be a neighbor of i and j'' be a non-neighbor of i belonging to i's new community $C_{j'} = C_{j''} = e$, k' be a neighbor of i and k'' be a non-neighbor of i belonging to another community $C_{k'} = C_{k''} = b$. From Cases **(17)**-**(22)**, we note that neighbors i' and k' have an incentive to change their community membership (and thus necessitate marking), but not j'. However, to keep the algorithm simple, we simply mark all the neighbors of vertex i as affected.

Case 17. $\Delta Q^{new}_{i':d\to e} = \Delta Q_{i':d\to e} + [\frac{2w_{ii'}}{m}] - \frac{K_i K_{i'}}{m^2}$

Case 18. $\Delta Q^{new}_{i':d\to b} = \Delta Q_{i':d\to b} + [\frac{w_{ii'}}{m}] - \frac{K_i K_{i'}}{2m^2}$

Case 19. $\Delta Q^{new}_{j':e\to d} = \Delta Q_{j':e\to d} - \frac{2w_{ij'}}{m} + \frac{K_i K_{j'}}{m^2}$

Case 20. $\Delta Q^{new}_{j':e\to b} = \Delta Q_{j':e\to b} - \frac{w_{ij'}}{m} + \frac{K_i K_{j'}}{2m^2}$

Case 21. $\Delta Q^{new}_{k':b\to d} = \Delta Q_{k':b\to d} - \frac{w_{ik'}}{m} + \frac{K_i K_{k'}}{2m^2}$

Case 22. $\Delta Q^{new}_{k':b\to e} = \Delta Q_{k':b\to e} + [\frac{w_{ik'}}{m}] - \frac{K_i K_{k'}}{2m^2}$

Further, from Cases **(23)-(28)**, we note that there is hardly any incentive for a change in community membership of vertices i'', j'', and k''. This is due to the change in delta-modularity being insignificant. There could still be an indirect cascading impact, where a common neighbor between vertices i and j would change its community, which could eventually cause vertex j to change its community as well [42]. However, this case is automatically taken care of as we perform marking of affected vertices during the community detection process.

Case 23. $\Delta Q^{new}_{i'':d\to e} = \Delta Q_{i'':d\to e} + \frac{K_i K_{i''}}{m^2}$

Case 24. $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} - \frac{K_i K_{i''}}{2m^2}$

Case 25. $\Delta Q^{new}_{j'':e\to d} = \Delta Q_{j'':e\to d} + \frac{K_i K_{j''}}{m^2}$

Case 26. $\Delta Q^{new}_{j'':e\to b} = \Delta Q_{j'':e\to b} + \frac{K_i K_{j''}}{2m^2}$

Case 27. $\Delta Q^{new}_{k'':b\to d} = \Delta Q_{k'':b\to d} + \frac{K_i K_{k''}}{2m^2}$

Case 28. $\Delta Q^{new}_{k'':b\to e} = \Delta Q_{k'':b\to e} - \frac{K_i K_{k''}}{2m^2}$ $\diamond$

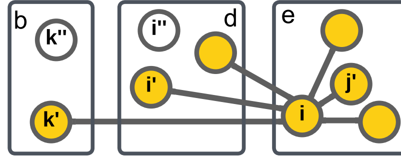


Figure 4: Processing community migration of vertices from one community to another, in the *Dynamic Frontier (DF)* approach. Here, pre-existing edges are represented by solid lines, and i represents the migrating vertex. Vertices i', j', and k' signify neighboring vertices of vertex i. Finally, vertices i'' and k'' represent non-neighbor vertices (to vertex i). Vertices marked as affected, in the current iteration of the algorithm, are indicated with a yellow highlight. Communities are indicated with labels b, e, and d.

5.1.4 Overall

Finally, based on Lemmas 5.1, 5.2, and 5.3, we can state the following for Alg. P-DF$_L$.

Theorem 5.4. *Given a batch update, Alg. P-DF$_L$ marks vertices having an incentive to change their community membership as affected.* □

We note that with Alg. P-DF$_L$, outlier vertices may not be marked as affected even if they have the potential to change community without any direct link to vertices in the frontier. Such outliers may be weakly connected to multiple communities, and if the current community becomes weakly (or less strongly) connected, they may leave and join some other community. It may also be noted that Alg. P-DS$_L$ is also an approximate approach and can miss certain outliers. In practice, however, we see little to no impact of this approximation of the affected subset of the graph on the final quality, i.e., modularity, of the communities obtained, as shown in Section 7.

5.2 Correctness of Dynamic Frontier based LPA (Alg. P-DF$_{\text{LPA}}$)

Given a batch update consisting of edge deletions Δ^{t-} and insertions Δ^{t+}, we now show that Alg. P-DF$_{\text{LPA}}$ marks all vertices as affected that might change their community membership. With *LPA*, the label C_i of a vertex i (before the batch update) is determined using Equation 5, as detailed in Section 3.1. We now consider the direct effect of each individual edge deletion (i, j) or insertion (i, j, w) in the batch update, on the label of a vertex — along with the indirect cascading effect of the change of label of a vertex, on the label associated with other vertices.

$$C_i = \underset{c \,\in\, \Gamma}{\arg\max}\, K_{i \to c} \tag{5}$$

5.2.1 On edge deletion

Lemma 5.5. *Given an edge deletion (i, j) between vertices i and j having the same label, vertex i (and j) should be marked as affected.*

Consider the case of edge deletion (i, j) of weight w, between vertices i and j having the same label $C_i = C_j = d$. The new label of i would be $C_i^{new} = \underset{c \,\in\, \Gamma}{\arg\max}\, K_{i \to c}^{new}$, where $K_{i \to d}^{new} = K_{i \to d} - w$. Note that, $K_{i \to c}^{new}$ denotes the total interconnecting weight of each label $c \in \Gamma$ to vertex i after the edge deletion (i, j), while $K_{i \to c}$ represents the total interconnecting weight of label c to vertex i before the edge deletion. Here, we have a reduced total weight associated with the previous best label d. Thus, i's label can change, and we mark it as affected. The same argument applies to vertex j as the edges are undirected.

Now consider the case of edge deletion (i, j) between vertices i and j having different labels $C_i = d$ and $C_j = e$ respectively. The new label for vertex i would be $C_i^{new} = \underset{c \,\in\, \Gamma}{\arg\max}\, K_{i \to c}^{new}$, where $K_{i \to d}^{new} = K_{i \to d}$. As we do not have any reduction in total weight associated with the previous best label d, the label of vertex i cannot change. Again, the same argument applies from vertex j. ◇

5.2.2 On edge insertion

Lemma 5.6. *Given an edge insertion (i, j, w) between vertices i and j having different labels, vertex i (and j) should be marked as affected.*

Consider the case of edge insertion (i, j, w) between vertices i and j having different labels $C_i = d$ and $C_j = e$. The new label for vertex i would be $C_i^{new} = \underset{c \,\in\, \Gamma}{\arg\max}\, K_{i \to c}^{new}$, where $K_{i \to d}^{new} = K_{i \to d}$ and $K_{i \to e}^{new} = K_{i \to e} + w$. Here, the label e may be the new maximum label for vertex i. We thus mark vertex i as affected. Again, the same argument applies for j due to the edges being undirected.

Now consider the case of edge insertion (i, j, w) between vertices i and j having the same label $C_i = C_j = d$. The new label for vertex i would be $C_i^{new} = \underset{c \,\in\, \Gamma}{\arg\max}\, K_{i \to c}^{new}$, where $K_{i \to d}^{new} = K_{i \to d} + w$. Now, here we actually have an increase in the total weight associated with the previous best label d. Thus, the label of vertex i cannot change. Again, the same argument applies to j. ◇

5.2.3 On vertex migration to another community

Lemma 5.7. *When a vertex i changes its label, and vertex j is its neighbor, the neighbor vertex j should be marked as affected.*

We now consider the indirect effects of deletion and insertion of edges by observing the impact of change in the label of one vertex on the labels of other vertices. Consider the case where a vertex i with label $C_i = d$

changes its label to $C_i^{new} = e$. Let i' be a neighbor of i with i's previous label $C_{i'} = d$, j' be a neighbor of i with i's new label $C_{j'} = e$, and k' be a neighbor of i with another label $C_{k'} = b$.

From Cases **(29)-(31)**, we note that neighbors i' and k' have a possibility to change their community membership (as thus necessitate marking), but not j'. However, to keep the algorithm simple, we simply mark all the neighbors of vertex i as affected. Finally, consider the case where vertices i and i'' are not neighbors, and vertex i changes its label. Note that by the definition of *LPA*, this cannot affect the label of vertex i''. However, there could still be an indirect impact, where a common neighbor between vertices i and i'' would change its label, which could eventually cause vertex i'' to change its label. Note that this case is automatically taken care of, as we mark affected vertices during the community detection process.

Case 29. $C_{i'}^{new} = \underset{c \in \Gamma}{\arg\max}\, K_{i' \to c}^{new}$, where $K_{i' \to d}^{new} = K_{i' \to d} - w_{ii'}$ and $K_{i' \to e}^{new} = K_{i' \to e} + w_{ii'}$

Case 30. $C_{j'}^{new} = \underset{c \in \Gamma}{\arg\max}\, K_{j' \to c}^{new}$, where $K_{j' \to e}^{new} = K_{j' \to e} + w_{ij'}$

Case 31. $C_{k'}^{new} = \underset{c \in \Gamma}{\arg\max}\, K_{k' \to c}^{new}$, where $K_{k' \to d}^{new} = K_{k' \to d} - w_{ik'}$ and $K_{k' \to e}^{new} = K_{k' \to e} + w_{ik'}$ ◇

5.2.4 Overall

Finally, based on Lemmas 5.5, 5.6, and 5.7, we can state the following for Alg. P-DF$_{\text{LPA}}$.

Theorem 5.8. *Given a batch update, Alg. P-DF$_{\text{LPA}}$ marks vertices that could change labels as affected.* □

6 Implementation details

6.1 Our Parallel Louvain implementation

We use an *asynchronous* implementation of the *Louvain* method (Algorithm 3), where threads work independently on different parts of the graph. Such asynchronicity allows for faster convergence but can also lead to more variability in the final result [4, 14]. We allocate a separate hashtable per thread to keep track of the delta-modularity of moving to each community linked to a vertex in the local-moving phase and to keep track of the total edge weight from one super-vertex to the other super-vertices in the aggregation phase.

Our optimizations include using OpenMP's `dynamic` loop schedule, limiting the number of iterations per pass to 20, using a tolerance drop rate of 10, setting an initial tolerance of 0.01, using an aggregation tolerance of 0.8, employing vertex pruning, making use of parallel prefix sum and preallocated Compressed Sparse Row (CSR) data structures for finding community vertices and for storing the super-vertex graph during the aggregation phase and using fast collision-free per-thread hashtables, well separated in memory.

6.2 Our Parallel LPA implementation

Like *Louvain*, we use an *asynchronous* parallel implementation of *LPA* (Algorithm 4). Further, we allocate a separate hashtable per thread that keeps track of the total weight of each unique label linked to a vertex. We observe that parallel *LPA* obtains communities of higher quality than its sequential version, possibly due to randomization in the processing order of vertices.

For *LPA*, our optimizations include using OpenMP's `dynamic` loop schedule, setting an initial tolerance of 0.05, enabling vertex pruning, employing the strict version of *LPA*, and using fast collision-free per-thread hashtables which are well separated in their memory addresses.

7 Evaluation

7.1 Experimental Setup

7.1.1 System

We use a server that has a 64-core x86-based AMD EPYC-7742 processor running at 2.25 GHz. Each core has an L1 cache of 4 MB, an L2 cache of 32 MB, and a shared L3 cache of 256 MB. The machine has 512 GB of DDR4 memory and runs on Ubuntu 20.04. We use GCC 9.4 and OpenMP 5.0 [29], and all programs are compiled with the -O3 flag enabled.

7.1.2 Configuration

We use 32-bit unsigned integer for vertex ids, 32-bit floating point for edge weights, but use 64-bit floating point for hashtable values, total edge weight, modularity calculation, and all places where performing an aggregation/sum of floating point values. Unless mentioned otherwise, we execute all parallel implementations with a default of 64 threads (to match the number of cores available on the system).

7.1.3 Dataset

Table 4 shows the graphs we use in our experiments. All of them are obtained from the SuiteSparse Matrix Collection [19]. The number of vertices in the graphs varies from 3.07 million to 214 million, and the number of edges varies from 25.4 million to 3.80 billion. These graphs are big enough in size and do not fit in the system cache(s). This makes the results of our experiments on these graphs interesting and realistic. We ensure that all edges are undirected and weighted with a default weight of 1.

Existing dynamic graphs in the SNAP repository [21] are small in size and do not help us study the proposed algorithms at scale. Further, the datasets from SNAP with ground-truth communities have a non-disjoint community structure where each vertex may be part of multiple communities. Our work is aimed at disjoint communities and hence we could not use the ground-truth that SNAP provides.

Table 4: List of 8 graphs obtained from the *SuiteSparse Matrix Collection* [19] (directed graphs are marked with $*$). Here, $|V|$ is the number of vertices, $|E|$ is the number of edges (after making the graph undirected), $|\Gamma|$ is the number of communities obtained using *Static Louvain*. Suffixes *B*, *M*, and *K* refer to a billion, a million, and a thousand respectively.

Graph	$\|V\|$	$\|E\|$	$\|\Gamma\|$
Web Graphs (LAW)			
it-2004*	41.3M	2.19B	5.28K
sk-2005*	50.6M	3.80B	3.47K
Social Networks (SNAP)			
com-LiveJournal	4.00M	69.4M	2.54K
com-Orkut	3.07M	234M	29
Road Networks (DIMACS10)			
asia_osm	12.0M	25.4M	2.38K
europe_osm	50.9M	108M	3.05K
Protein k-mer Graphs (GenBank)			
kmer_A2a	171M	361M	21.2K
kmer_V1r	214M	465M	6.17K

7.1.4 Batch generation

We take a base graph from the dataset and generate a random batch update [42] consisting purely of edge deletions or insertions for simplicity [7], each with an edge weight of 1. All batch updates are undirected, i.e.,

for every edge insertion (i, j, w) in the batch update, the edge (j, i, w) is also a part of the batch update. For simplicity, we generate these edges such that the selection of each vertex (as endpoint) is equally probable, and we do not add any new vertices to the graph. Testing with mixed batch updates is part of our future work.

7.1.5 Adjusting batch size

For all dynamic graph-based experiments, we modify the batch size as a fraction of the total number of edges in the original (undirected) graph from 10^{-7} to 0.1 (i.e., from $10^{-7}|E|$ to $0.1|E|$). For a billion-edge graph, this amounts to a batch size of 100 to 100 million edges. Keep in mind that dynamic graph algorithms are helpful for small batch sizes in interactive applications. For large batches, it is usually more efficient to run the static algorithm. We employ multiple random batch updates for each batch size and report the average across these runs in our experiments.

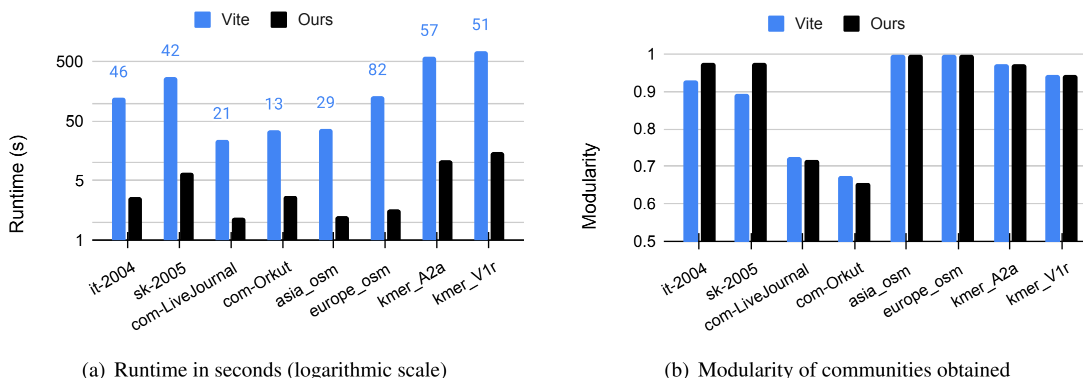
7.1.6 Determining optimality of result

Community detection is an NP-hard problem and existing polynomial algorithms are *heuristic*. We study correctness in terms of *modularity score* of communities identified (higher is better), similar to previous works in the area [9, 11, 39, 42]. As Figure 7 shows, modularity of communities detected by our proposed dynamic algorithms is close to the modularity of communities detected by corresponding static algorithms.

7.2 Performance of Alg. Static$_{\text{L}}$

Figures 5(a) shows the runtime taken by our parallel implementation of Louvain, in contrast to Vite [11],[4] both running on the system given in Section 7.1.1, while Figure 5(b) shows the modularity of obtained communities. Our parallel *Louvain*, Alg. Static$_{\text{L}}$, has a runtime of 6.2 seconds on the undirected `sk-2005` graph containing 1.9 billion edges, and is on average $42\times$ faster than Vite, an MPI+OpenMP implementation of Louvain method for undirected graph clustering [11]. We observe that graphs with lower average degree (*road networks* and *protein k-mer graphs*) and graphs with poor community structure (such as `com-LiveJournal` and `com-Orkut`) have a larger time$/|E|$ factor.

We note that 48% (most) of the runtime is spent in the local-moving phase, while the aggregation phase accounts for only 29% of the run time. Further, 68% (most) of the runtime is spent in the first pass of the algorithm, which is the most expensive pass (later passes work on super-vertex graphs) [40].



(a) Runtime in seconds (logarithmic scale)

(b) Modularity of communities obtained

Figure 5: Runtime in seconds and modularity of communities obtained, by Vite (Louvain) and Alg. Static$_{\text{L}}$

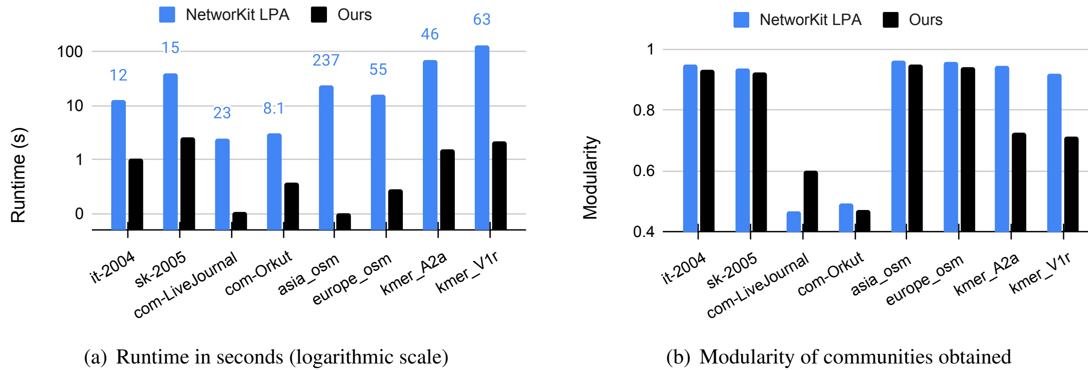
7.3 Performance of Alg. Static$_{\text{LPA}}$

Figures 6(a) shows the runtime taken by our parallel implementation of LPA, in contrast to NetworKit LPA [35],[5] both running on the system given in Section 7.1.1, while Figure 6(b) shows the modularity of obtained

[4] The source code Vite is obtained from `https://github.com/ECP-ExaGraph/vite`

[5] The source code for NetworKit is obtained from `https://github.com/networkit/networkit`

communities. Our parallel *LPA*, Alg. Static$_{\text{LPA}}$, has a run time of 2.7 seconds on the undirected *sk-2005* graph containing 1.9 billion edges, and is on average $57\times$ faster than NetworKit LPA, a popular parallel implementation of LPA. We observe that graphs with a lower average degree, such as road networks and protein k-mer graphs, have a larger time$/|E|$ factor.



(a) Runtime in seconds (logarithmic scale)

(b) Modularity of communities obtained

Figure 6: Runtime in seconds and modularity of communities obtained, by NetworKit LPA and Alg. Static$_{\text{LPA}}$

7.4 Performance of Alg. P-DF$_{\text{L}}$ (Algorithm 3)

7.4.1 Overall Performance

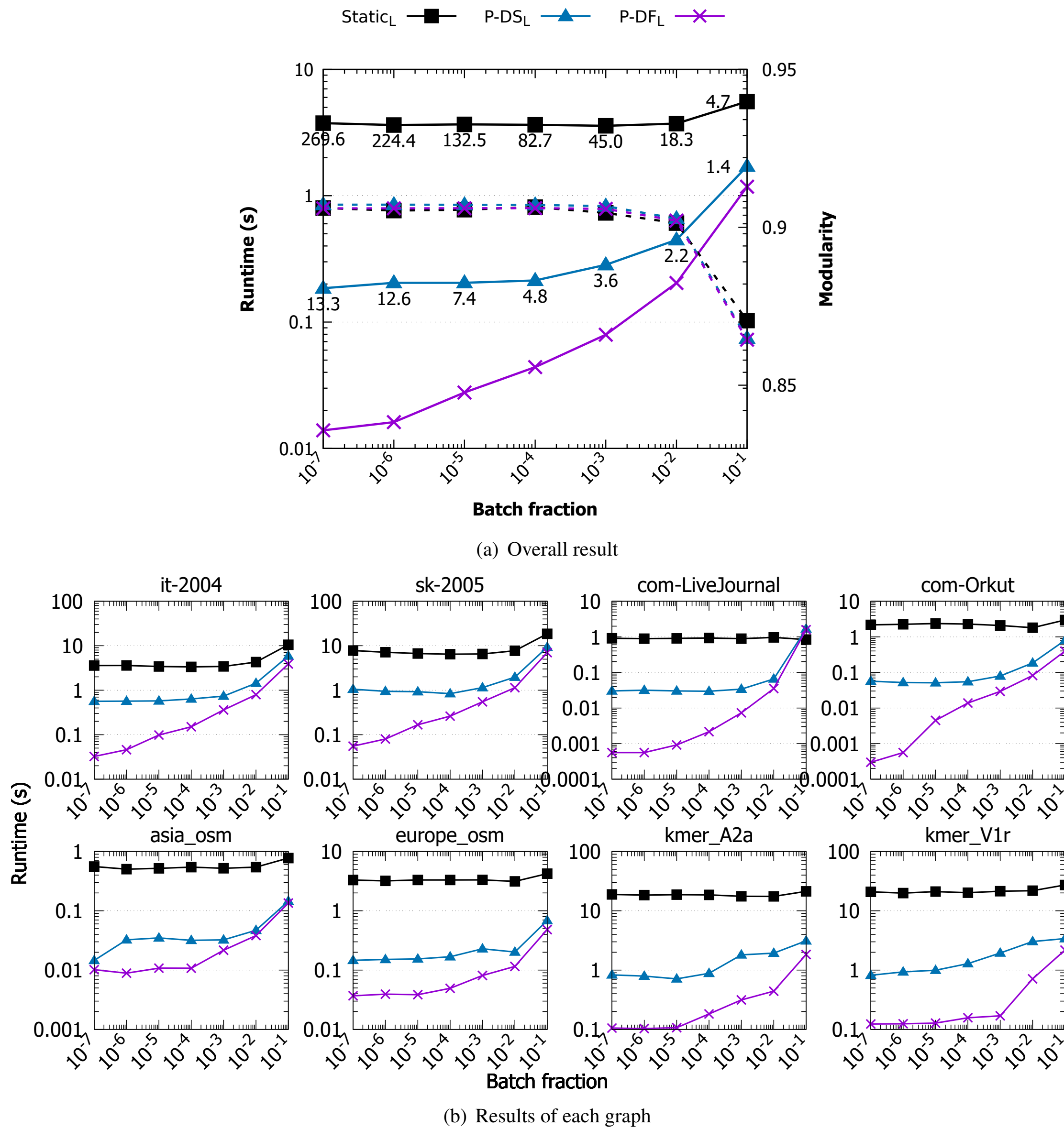
We first study the performance of Alg. P-DF$_{\text{L}}$ on batch updates of size varying from $10^{-7}|E|$ to $0.1|E|$, and compare it with parallel Alg. Static$_{\text{L}}$ and Alg. P-DS$_{\text{L}}$ (Alg. P-DS applied to *Louvain*).

Figure 7(a) shows the average results of the experiment. The average execution time is calculated using the geometric mean ensuring consistent scaling across graphs, while average modularity score is calculated using arithmetic mean due to less variation across graphs and batch sizes. We observe the following from Figure 7(a). The modularity of communities obtained by both Alg. P-DF$_{\text{L}}$ and Alg. P-DS$_{\text{L}}$ is nearly identical to that obtained by Alg. Static$_{\text{L}}$. Alg. P-DF$_{\text{L}}$ converges the fastest with an average speedup of $128.8\times$ over Alg. Static$_{\text{L}}$, and $7.3\times$ over Alg. P-DS$_{\text{L}}$. As the batch size increases, the number of vertices marked as affected by Alg. P-DF$_{\text{L}}$ increases. This increases the time taken by Alg. P-DF$_{\text{L}}$ as the batch size increases. Further, as Figure 7(b) shows, dynamic approaches significantly outperform Alg. Static$_{\text{L}}$ on social networks, road networks, and protein k-mer graphs (which do not have a dense community structure or have a low $|E|/|V|$ ratio).

We note that the difference in modularity score of communities identified by Alg. P-DF$_{\text{L}}$ and Alg. Static$_{\text{L}}$ across graphs and batch sizes is less than 0.004. Therefore, the average modularity score is shown only in Figure 7(a) with the modularity score anchored to the Y2-axis. At a batch size of $0.1|E|$, the base graph has a 10% increase/decrease in the number of edges which arbitrarily disrupt the original community structure. This results in the static algorithm needing more iterations to converge.

The slowdown of Alg. Static$_{\text{L}}$ can be attributed to uniform batches of edge deletions and insertions, which arbitrarily disrupt the original community structure. This results in the static algorithm needing more iterations to converge.

From our experiments, we also note that the pre-processing step of copying the auxiliary information of vertex and community weights from the current graph to the new graph takes under 1% of the total run time on average. The actual computation of identifying the new community labels of affected vertices consumes more than 90% of the total run time, and post-processing corresponding to community aggregation and recomputing vertex and community weights for the next batch update takes under 5% of the total run time on average across instances and batch sizes.



(a) Overall result

(b) Results of each graph

Figure 7: Time taken (solid lines), and modularity of communities obtained (dashed lines) along the Y2 axis, by Alg. Static$_L$, Alg. P-DS$_L$, and Alg. P-DF$_L$ on batch updates of increasing size from $10^{-7}|E|$ to $0.1|E|$. Both axes are logarithmic. The numbers on the lines corresponding to Alg. Static$_L$ and Alg. P-DS$_L$ indicate the speedup of Alg. P-DF$_L$ over the two algorithms.

7.4.2 Affected vertices and Performance

We now study the difference in the number of vertices marked as affected by Alg. P-DF$_{\mathrm{L}}$ and Alg. P-DS$_{\mathrm{L}}$. In Figure 8, we show the fraction of vertices marked as affected by Alg. P-DF$_{\mathrm{L}}$ and Alg. P-DS$_{\mathrm{L}}$ on average over the instances in Table 4. The numbers on the line corresponding to Alg. P-DS$_{\mathrm{L}}$ in Figure 8 shows the ratio of the number of vertices marked affected by Alg. P-DS$_{\mathrm{L}}$ to that of Alg. P-DF$_{\mathrm{L}}$.

We notice from Figure 8 that Alg. P-DF$_{\mathrm{L}}$ marks a significantly smaller fraction of vertices as affected compared to Alg. P-DS$_{\mathrm{L}}$. The run time of these algorithms, however, do not differ in this ratio as observed from Figure 7(a). This is because of the following reasons. Alg. P-DS$_{\mathrm{L}}$ marks entire communities of vertices as affected but many such affected vertices may not really undergo a change in their community label. So, a large fraction of such affected vertices converge in only one iteration. In addition, Alg. P-DF needs to expand — so the work is not proportional to affected vertices.

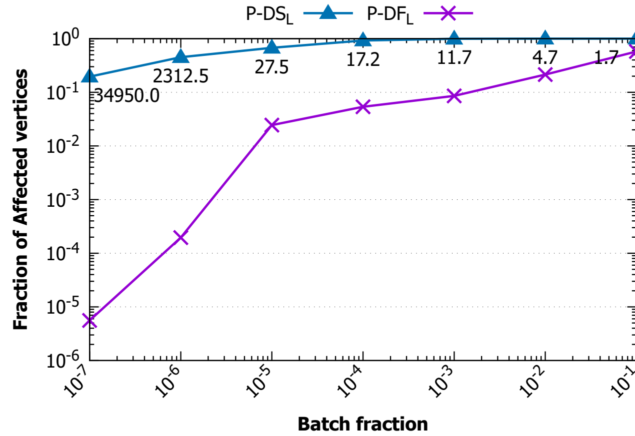


Figure 8: Fraction of vertices marked as affected (average) by Alg. P-DS$_{\mathrm{L}}$ and Alg. P-DF$_{\mathrm{L}}$, as mentioned in Section 7.4, on graphs in Table 4. The numbers on the line corresponding to Alg. P-DS$_{\mathrm{L}}$ indicate the ratio of the fraction of vertices identified as affected by Alg. P-DS$_{\mathrm{L}}$ to that of Alg. P-DF$_{\mathrm{L}}$.

7.4.3 Stability of Alg. P-DF$_{\mathrm{L}}$

Intuitively, if the graphs G^t and $G^{t'}$ are identical for some t and t', we expect Alg. P-DF$_{\mathrm{L}}$ to produce the same communities for G^t and $G^{t'}$. We refer to this property of a dynamic algorithm as its stability, measured as the percentage of vertices that agree on the community label across two identical graphs. Vertices within weak communities structures tend to be unstable, as they connect to multiple communities with similar strength.

To measure the stability of Alg. P-DS$_{\mathrm{L}}$ and Alg. P-DF$_{\mathrm{L}}$, we proceed as follows. Let G be an initial graph. We generate random batch updates of size $10^{-7}|E|$ to $0.1|E|$ consisting of edge deletions to obtain the graph G^1. We then apply each of the above algorithms on G^1 to identify the new communities. Subsequently, we create another batch of updates that consists of inserting the edges deleted in the prior time step. This graph, G^2, is essentially the original graph G. We obtain the community labels of the vertices in the graph G^2 by appealing to the dynamic algorithms. Finally, we compare the community label of each vertex in the graphs G and G^2. We measure the resulting match in community membership of vertices with Alg. P-DS$_{\mathrm{L}}$ and Alg. P-DF$_{\mathrm{L}}$ on batch updates of size $10^{-7}|E|$ to $0.1|E|$.

Our results indicate that Alg. P-DS$_{\mathrm{L}}$ has minimum of 99.68% match with the original community memberships across all batch sizes, while Alg. P-DF$_{\mathrm{L}}$ has a minimum of 99.70% match. This indicates that both the algorithms are stable.

7.5 Performance of Alg. P-DF$_{\text{LPA}}$ (Algorithm 4)

7.5.1 Overall Performance

In this experiment, we study the performance of Alg. P-DF$_{\text{LPA}}$ with random batch updates of size ranging from $10^{-7}|E|$ to $0.1|E|$, and compare it to parallel Alg. Static$_{\text{LPA}}$ and Alg. P-DS$_{\text{LPA}}$.

Figure 9(a) shows the average result of the experiment, obtained with geometric mean of the respective runtimes. While all approaches obtain communities of equivalent modularity, Alg. P-DF$_{\text{LPA}}$ converges on average $48.8\times$ faster than Alg. Static$_{\text{LPA}}$, and $6.7\times$ faster than Alg. P-DS$_{\text{LPA}}$ from a batch size of $10^{-7}|E|$ up to $0.01|E|$. As shown in Figure 9(b), it has good performance on web graphs and social networks (graphs with high $|E|/|V|$ ratio). Graphs with a low $|E|/|V|$ ratio are likely to have more affected vertices for a given batch size, more hashtable resets, low cache use, and more community updates. Note, however, that the modularity of communities obtained with *LPA* is not on par with *Louvain*. The slowdown of the static algorithm can be attributed to the uniform batches of insertions/deletions, which arbitrarily disrupt the original community structure — necessitating more iterations for the static algorithm to converge.

7.5.2 Stability of Alg. P-DF$_{\text{LPA}}$

Just as in Section 7.4.3, we study the stability of Alg. P-DS$_{\text{LPA}}$ and Alg. P-DF$_{\text{LPA}}$ on random batch updates of size $10^{-7}|E|$ to $0.1|E|$. From the results, we observe that Alg. P-DS$_{\text{LPA}}$ has minimum of 95.53% match with the original community memberships across all batch sizes, while Alg. P-DF$_{\text{LPA}}$ has a minimum of 95.75% match. This indicates that Alg. P-DF$_{\text{LPA}}$ is stable.

7.6 Performance of Alg. P-DF$_{\text{H}}$ (Algorithm 5)

7.6.1 Overall Performance

We now study the performance of Alg. P-DF$_{\text{H}}$ on batch updates of size $10^{-7}|E|$ to $0.1|E|$, and compare it with Alg. P-DF$_{\text{L}}$ and Alg. P-DF$_{\text{LPA}}$.

The average modularity of communities obtained by Alg. P-DF$_{\text{H}}$ is nearly identical to that obtained by Alg. P-DF$_{\text{L}}$, as shown in Figure 10(a) while obtaining a mean speedup of $2.0\times$ across batch sizes of $10^{-7}|E|$ to $0.01|E|$. Alg. P-DF$_{\text{H}}$ is thus an efficient and high-modularity dynamic community detection approach, especially on road networks and protein k-mer graphs, as shown in Figure 10(b), where it significantly outperforms Alg. P-DF$_{\text{L}}$ for smaller batch updates. Alg. P-DF$_{\text{LPA}}$ has about the same performance but obtains communities of lower modularity (see blue dotted line in Figure 10(a)).

7.7 Scalability of Alg. P-DF$_{\text{L}}$, Alg. P-DF$_{\text{LPA}}$, and Alg. P-DF$_{\text{H}}$

Finally, we study the strong-scaling behavior of Alg. P-DF$_{\text{H}}$ and compare it with Alg. P-DF$_{\text{L}}$ and Alg. P-DF$_{\text{LPA}}$. To do this, we fix the batch size at $10^{-3}|E|$, vary the number of threads in use from 1 to 64, and measure the speedup of each algorithm to its sequential version.

As shown in Figure 11(a), Alg. P-DF$_{\text{L}}$, Alg. P-DF$_{\text{LPA}}$, and Alg. P-DF$_{\text{H}}$ obtain a speedup of $7.0\times$, $16.1\times$, and $16.1\times$ respectively at 64 threads; with their speedup increasing at a mean rate of $1.38\times$, $1.59\times$, and $1.59\times$ respectively for every doubling of threads. The y-axis of Figure 11(a) shows the ratio of the time taken by the respective algorithms using one thread to the time taken using a given number of threads. Also note from Figure 11(b) that Alg. P-DF$_{\text{H}}$ offers good speedup on web graphs and social networks (consider the scale of the y-axis), but does not scale well on road networks and protein k-mer graphs (with low $|E|/|V|$ ratio). We can also observe from Figure 11(b) that for some of the graphs (*it-2004*, *asia_osm*, *europe_osm*, *kmer_A2a*, *kmer_V1r*) the speedup achieved drops when scaling beyond 32 threads. This could be attributed to the lack of enough work for all the 64 threads in these instances.

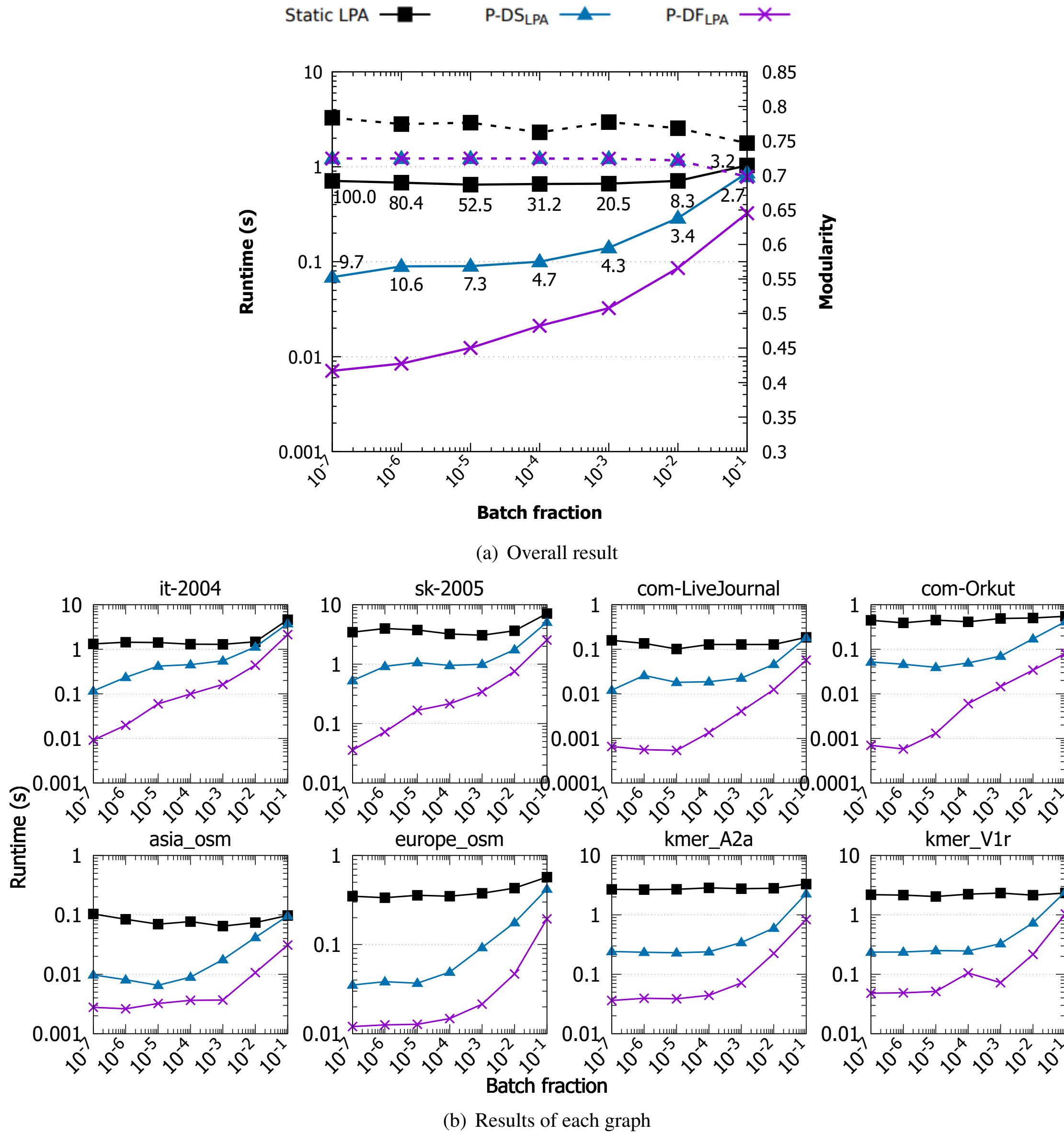
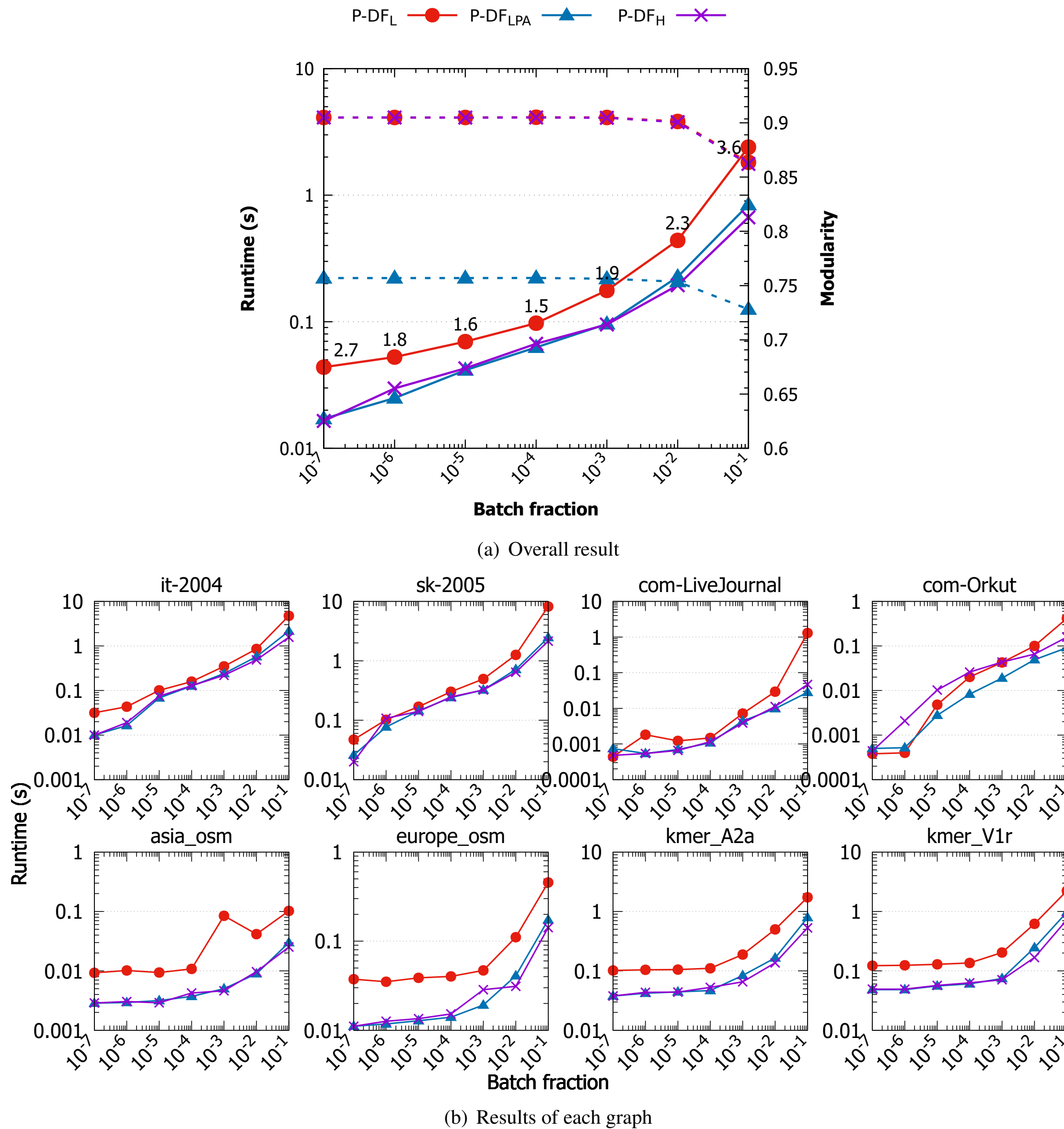


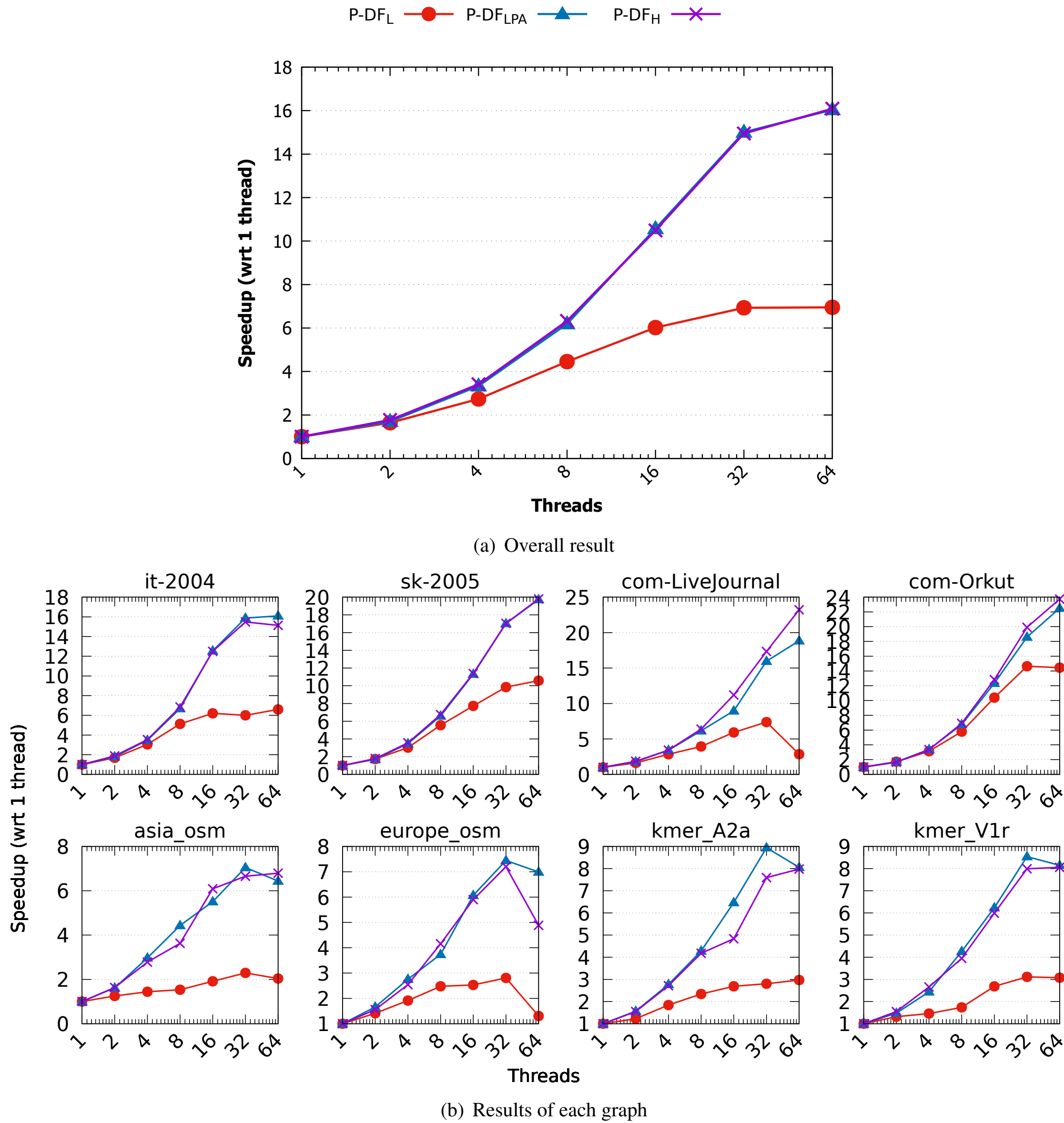
Figure 9: Time taken (solid lines), and modularity of communities obtained (dashed lines) along the Y2 axis, by Alg. $\text{Static}_{\text{LPA}}$, Alg. P-DS_{LPA}, and Alg. P-DF_{LPA} on batch updates of increasing size from $10^{-7}|E|$ to $0.1|E|$. Note that both axes are logarithmic. The labels corresponding to Alg. $\text{Static}_{\text{LPA}}$ and Alg. P-DS_{LPA} indicate the speedup of Alg. P-DF_{LPA} over the two algorithms.



(a) Overall result

(b) Results of each graph

Figure 10: Time taken (solid lines), and modularity of communities obtained (dashed lines) along the Y2 axis, by Alg. P-DF$_{\text{L}}$, Alg. P-DF$_{\text{LPA}}$, and Alg. P-DF$_{\text{H}}$ on batch updates of increasing size from $10^{-7}|E|$ to $0.1|E|$. Note that both axes are logarithmic. The number on the line corresponding to Alg. P-DF$_{\text{L}}$ indicates the speedup of Alg. P-DF$_{\text{H}}$ over Alg. P-DF$_{\text{L}}$.



(a) Overall result

(b) Results of each graph

Figure 11: Strong scalability result of Alg. P-DF$_{\mathrm{L}}$, Alg. P-DF$_{\mathrm{LPA}}$, and Alg. P-DF$_{\mathrm{H}}$ (Algorithms 3, 4, and 5) on batch updates of size $10^{-3}|E|$. The number of threads is doubled from 1 to 64 (logarithmic scale).

7.8 Further Discussion

7.8.1 Processing a mixed batch of insertions and deletions

The batch updates that we generate contain a set of edges that are to be inserted or deleted. This practice is in line with other existing works [36, 42]. Our algorithms do not need any changes to handle a batch update consisting of a mix of insert and delete operations. Keeping them separate has minimal impact on the performance of the algorithm.

7.8.2 Weighted Graphs

In our experiments, we set the weight on every edge to be 1. This corresponds to an unweighted setting. We note that our algorithms and programs continue to work in the case of weighted graphs also with minor changes. We focused our testing on unweighted graphs because publicly available datasets of large real-world weighted graphs are scarce. Further, since these minor changes equally impact our algorithms and the Δ-screening based algorithms, we expect the performance to be along similar lines.

7.8.3 Comparison with respect to Riedy and Bader

Riedy and Bader [33] propose a batch parallel dynamic algorithm for community detection. They compare the run time of their dynamic algorithm to that of a static recomputation. On the graphs `caidaRouterLevel`, `coPapersDBLP`, and `eu-2005`, and at respective batch sizes of $0.08|E|$, $0.03|E|$ and $0.06|E|$, they report a speedup of $40.1\times$, $10.8\times$, and $327\times$ over their corresponding static algorithm. On these three graphs and respective batch sizes, Alg. P-DF$_{\text{H}}$ achieves a speedup of $41.7\times$, $29.5\times$, and $14.5\times$, respectively, compared to a full static recomputation. This might compare unfavorably. However, the algorithm of Riedy and Bader does not identify cascading changes to communities. As their source code is not available, we could not do a more direct comparison.

7.8.4 Maintaining modularity across batches

In our experiments with continuous batch updates of edge insertions of size $10^{-3}|E|$, we find that the modularity of communities obtained using Alg. P-DS$_{\text{L}}$and Alg. P-DF$_{\text{L}}$ starts to drop (compared to Alg. Static$_{\text{L}}$) by 48% and 5.7% respectively after around 1300 batches of updates. The same happens for Alg. P-DF$_{\text{H}}$ by 10% after around 600 updates. Therefore, we recommend the reader to run Alg. Static$_{\text{L}}$ once for $1000/500$ updates to maintain the modularity of communities across multiple batch updates and correct the error introduced.

7.9 Reproducibility

All our results are reproducible. The source code for the experiments reported in this paper along with necessary scripts for obtaining the datasets and compiling the software is available at `https://github.com/merferry/communities-cpu--artifact`.

8 Conclusion

In conclusion, this paper focused on designing high-speed community detection algorithms for batch dynamic settings. We introduced the *Dynamic Frontier* approach (Alg. P-DF), which efficiently handles edge deletions and insertions by processing only the affected vertices with minimal overhead. Through parallel implementations of the *Louvain* and *LPA* algorithms, we demonstrated up to $7.3\times$ and $6.7\times$ performance improvements, respectively, compared to Δ-*screening*. Our hybrid algorithm also achieved high-quality results, running $14.6\times$ faster than the state-of-the-art. In the future, we plan to explore applying our algorithms to GPUs, along with applications of Alg. P-DF in other social network algorithms.

References

[1] T. Aynaud and J. Guillaume. Static community detection algorithms for evolving networks. In *8th IEEE WiOpt*, pages 513–519, 2010.

[2] S. Bhattacharya, D. Chakrabarty, M. Henzinger, and D. Nanongkai. Dynamic algorithms for graph coloring. In *Proc. of 29th ACM-SIAM SODA*, pages 1–20, 2018.

[3] A. Bhowmick, S. Vadhiyar, and P. V. Varun. Scalable multi-node multi-gpu louvain community detection algorithm for heterogeneous architectures. *CCPE*, 34(17):1–18, 2022.

[4] V. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *J. Stat. Mech.*, 2008(10):P10008:1–12, 2008.

[5] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner. On modularity clustering. *IEEE TKDE*, 20(2):172–188, 2007.

[6] C. Cheong, H. Huynh, D. Lo, and R. Goh. Hierarchical Parallel Algorithm for Modularity-Based Community Detection Using GPUs. In *Proc. of 19th Euro-Par*, pages 775–787. Springer-Verlag, 2013.

[7] W. Chong and L. Teow. An incremental batch technique for community detection. In *Proc. of 16th IEEE FUSION*, pages 750–757, 2013.

[8] M. Cordeiro, R. Sarmento, and J. Gama. Dynamic community detection in evolving networks using locality modularity optimization. *Social Network Analysis and Mining*, 6(1):1–20, 2016.

[9] Mário Cordeiro, Rui Portocarrero Sarmento, and Joao Gama. Dynamic community detection in evolving networks using locality modularity optimization. *Soc. Netw. Anal. Min.*, 6:1–20, 2016.

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

[11] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarria-Miranda, A. Khan, and A. Gebremedhin. Distributed Louvain algorithm for graph community detection. In *IEEE IPDPS*, pages 885–895, 2018.

[12] A. Ghoshal, N. Das, S. Bhattacharjee, and G. Chakraborty. A fast parallel genetic algorithm based approach for community detection in large networks. In *Proc. Intl. Conf. Comm. Sys. & Net. (COMSNETS)*, pages 95–101, 2019.

[13] S. Gregory. Finding overlapping communities in networks by label propagation. *New J. Physics*, 12:103018:1–26, 10 2010.

[14] M. Halappanavar, H. Lu, A. Kalyanaraman, and A. Tumeo. Scalable static and dynamic community detection using Grappolo. In *IEEE HPEC*, pages 1–6, Sep 2017.

[15] J. Han, W. Li, L. Zhao, Z. Su, Y. Zou, and W. Deng. Community detection in dynamic networks via adaptive label propagation. *PloS one*, 12(11):e0188655:1–16, 2017.

[16] B. Hu, W. Li, X. Huo, Y. Liang, M. Gao, and P. Pei. Improving louvain algorithm for community detection. In *Intl. Conf. AI and Engg. Appl.*, pages 110–115, 2016.

[17] H. Kanezashi and T. Suzumura. An incremental local-first community detection method for dynamic graphs. In *Proc. Intl. Conf. on Big Data*, pages 3318–3325. IEEE, 2016.

[18] A. Khanda, S. Srinivasan, S. Bhowmick, B. Norris, and S. Das. A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks. *IEEE TPDS*, 33(4):929–940, 2021.

[19] S. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. Davis, M. Henderson, Y. Hu, and R. Sandstrom. The SuiteSparse matrix collection website interface. *JOSS*, 4(35):1244, Mar 2019.

[20] A. Lancichinetti and S. Fortunato. Community detection algorithms: a comparative analysis. *Physical Review. E*, 80(5):1–11, 2009.

[21] J. Leskovec and A. Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. 06 2014.

[22] K. Liu, J. Huang, H. Sun, M. Wan, Y. Qi, and H. Li. Label propagation based evolutionary clustering for detecting overlapping and non-overlapping communities in dynamic networks. *Knowledge-Based Systems*, 89:487–496, 2015.

[23] H. Lu, M. Halappanavar, and A. Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.

[24] X. Meng, Y. Tong, X. Liu, S. Zhao, X. Yang, and S. Tan. A novel dynamic community detection algorithm based on modularity optimization. In *Proc. Intl. Conf. Software Engg. and Service science*, pages 72–75, 2016.

[25] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo. Community detection on the GPU. In *IEEE IPDPS*, pages 625–634, May 2017.

[26] K. Nath and S. Roy. Detecting intrinsic communities in evolving networks. *Social Network Analysis and Mining*, 9:1–15, 2019.

[27] M. Newman. Detecting community structure in networks. *The European Physical Journal B - Condensed Matter*, 38(2):321–330, Mar 2004.

[28] M. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3):036104:1–19, 2006.

[29] OpenMP Architecture Review Board. OpenMP application program interface version 5.0, 2018.

[30] U. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106:1–10, Sep 2007.

[31] G. Ramalingam. *Bounded Incremental Computation*, volume 1089. Springer-Verlag, LNCS, 1996.

[32] S. Regunta, S. Tondomker, K. Shukla, and K. Kothapalli. Efficient parallel algorithms for dynamic closeness-and betweenness centrality. In *Proc. ACM ICS*, pages e6650:1–12, 2020.

[33] J. Riedy and D. A. Bader. Multithreaded community monitoring for massive streaming graph data. In *IEEE IPDPSW*, pages 1646–1655, 2013.

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

[35] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016.

[36] H. Sun, J. Huang, X. Zhang, J. Liu, D. Wang, H. Liu, J. Zou, and Q. Song. Incorder: Incremental density-based community detection in dynamic networks. *Knowledge-Based Systems*, 72:1–12, 2014.

[37] Z. Sun, J. Sheng, B. Wang, A. Ullah, and F. R. Khawaja. Identifying communities in dynamic networks using information dynamics. *Entropy*, 22(4):425, 2020.

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

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

[40] C. Wickramaarachchi, M. Frincu, P. Small, and V. Prasanna. Fast parallel algorithm for unfolding of communities in large graphs. In *IEEE HPEC*, pages 1–6, 2014.

[41] J. Xie, M. Chen, and B. Szymanski. LabelrankT: Incremental community detection in dynamic networks via label propagation. In *SIGMOD/PODS'13*, pages 25–32. ACM, 2013.

[42] N. Zarayeneh and A. Kalyanaraman. Delta-Screening: A Fast and Efficient Technique to Update Communities in Dynamic Graphs. *IEEE TNSE*, 8(2):1614–1629, 2021.

[43] D. Zhuang, J. Chang, and M. Li. Dynamo: Dynamic community detection by incrementally maximizing modularity. *IEEE TKDE*, 33(5):1934–1945, 2019.