Shared-Memory Parallel Algorithms for Community Detection in Dynamic Graphs

SUBHAJIT SAHU and KISHORE KOTHAPALLI, IIIT Hyderabad, India
DIP SANKAR BANERJEE, IIT Jodhpur, India

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 high-speed 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 is with minimal overhead. We apply this approach to both Louvain, a high quality, and Label Propagation Algorithm (LPA), a high speed static community detection algorithm. Our approach achieves a mean speedup of 2.4× and 5.1× 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 providing a speedup of 5.0× on top of Dynamic Frontier based Louvain.

Additional Key Words and Phrases: Dynamic graphs, Community detection, Parallel algorithms, Dynamic Frontier approach, Dynamic Louvain algorithm, Dynamic Label Propagation Algorithm (LPA), Dynamic Hybrid Louvain-LPA

1 INTRODUCTION

Graphs present a powerful abstraction for representing data and their internal relationships. Finding communities in graphs, where the vertices within a community are more strongly connected than those outside, is an NP-hard problem [10]. These communities find extensive applications in recommendation systems, targeted advertising, drug discovery, protein annotation, and topic discovery [11].

Two popular heuristic-based algorithms for intrinsic and disjoint community detection are the *Louvain* method [4] and the *Label Propagation Algorithm (LPA)* [22]. Communities are intrinsic when identified based on network topology alone and are disjoint when each vertex belongs to only one community. Community detection algorithms use the modularity metric proposed by Newman et al. [21] to measure the quality of the communities identified.

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. Such algorithms are known as *dynamic algorithms*.

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 efficiency considerations, designing parallel batch dynamic graph algorithms is naturally more challenging. Examples of parallel dynamic algorithms include those for graph coloring [2], shortest paths [15], and centrality scores [24].

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. However, a critical examination of the extant literature indicates a few shortcomings.

	[1, 7, 8, 19]	[28, 31, 32]	[25]	**This Paper**
Fully dynamic	✓	✓	✓	✓
Batch update	✓	✓	✓	✓
Process subset	×	✓	✓	✓
Cascading updates	×	×	×	✓
Parallel algorithm	×	×	✓	✓
Hybrid algorithm	×	×	×	✓

Table 1. Comparison of community detection papers.

Some of these algorithms, [8, 19], do not outperform the static algorithms even for modest-sized batch updates. Many reported algorithms [28, 31, 32], are not designed in the parallel setting. Algorithms from [1, 7] adapt the existing community labels and run an algorithm such as the Louvain algorithm 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. Some of the existing algorithms [8, 25] do not consider the cascading impact of changes in community labels. Cascading changes refer to the possibility that the community label of a vertex changes because of a change in the community label of its neighbor. Finally, algorithms such as from [25, 31] 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. [31] incurs a significant computation time. Table 1 summarizes the above discussion.

A perusal of Table 1 motivates us to design efficient parallel algorithms that update the community structures of an evolving graph. Ideally, the algorithm should identify a subset of the graph to be process with a low overhead [23]. 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.

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 (**cf.** Section 3.1). Given a batch update consisting of edge deletions or edge insertions, the *Dynamic Frontier* approach incrementally identifies an approximate set of affected vertices in the graph with a low run time overhead.

We show how to combine our *Dynamic Frontier* approach with two parallel algorithms: the parallel *Louvain* and the parallel LPA (**cf.** Sections 3.3 and 3.4). In addition to accepting the previous community membership of each vertex, our algorithms accept auxiliary information to improve scalability. In addition, we also show how to use the Dynamic Frontier approach under a combination of both the *Louvain* and LPA to arrive at a *hybrid* algorithm (**cf.** Section 3.5).

We compare these three algorithms with a 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.

Additionally, our experimental results demonstrate that our algorithms achieve good community stability. Community stability is helpful because it will simplify tracking communities over time. Finally, we show that our algorithm has good scaling performance. [1]

[1] For reproducibility, our source code is at https://bit.ly/ipdps24-375

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 a parallel Δ-screening algorithm [31], 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.

1.2 Related work

The Louvain algorithm is a popular and efficient algorithm to identify communities with a high modularity. As a result, it is widely favored among researchers [17]. Algorithmic improvements to the *Louvain* method include early pruning of non-promising candidates [12], threshold scaling [12, 20], threshold cycling [9], and early termination [9].

A variety of techniques have been studied to parallelize the *Louvain* algorithm on multicore CPUs, GPUs [6], hybrid CPU-GPUs, multi GPUs [3, 6], and distributed systems [3]. These include parallelizing the costly first iteration [27], ordering vertices via graph coloring [12], using vector-based hashtables for data caching [12], using adaptive parallel thread assignment [20], using sort-reduce instead of hashing [6], using simple partitions based on vertex ids [6, 9], and identifying and moving ghost/doubtful vertices [3].

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, 31]. 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.

While *Louvain* obtains high-quality communities, we find it to be $2.3 - 14\times$ slower than *LPA* (which obtains communities of lower quality by $3.0 - 30\%$). LPA is faster than the Louvain algorithm, as it does not require repeated optimization steps and is easier to parallelize. Improvements upon the *LPA* include using stable max-label choosing [29], identifying central nodes and combining communities [30], and using frontiers with alternating push-pull to reduce edges visited and improve solution quality [18].

For *LPA*, a stabilized process based on LabelRank algorithm has been proposed [28]. Adaptive Label Propagation Algorithm (ALPA) is another dynamic approach, which first performs a warm-up *LPA* on a subset of the network, followed by Local Label Propagation (LLP), which expands as a frontier of nodes that change labels [13].

2 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 $m = \sum_{i,j \in V} w_{ij}/2$.

2.1 Community detection

Disjoint community detection is the process of arriving at a community membership mapping, $C : V \rightarrow \Gamma$, 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 $J_{i\rightarrow c} = \{j \mid j \in J_i \; and \; C_j = c\}$, the sum of those edge weights as $K_{i\rightarrow c} = \{w_{ij} \mid j \in J_{i\rightarrow c}\}$, 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 $\Sigma_c = \sum_{(i,j)\in E} \text{ and } {}_{C_i=c} \, w_{ij}$ [31].

We use *modularity* Q to evaluate the quality of communities obtained, as shown in Eq. 1. It represents the difference between the fraction of edges within communities and the expected fraction of edges in a uniform assignment of edges to communities. It lies in the range $[-0.5, 1]$ and a higher value is better [5]. The *delta modularity* of moving a vertex i from the community d to the community c, denoted as $\Delta Q_{i:d\rightarrow c}$, can be calculated using Eq. 2.

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

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

2.2 Algorithms for Static Graphs

2.2.1 Louvain algorithm [4]. *Louvain* is a greedy, modularity optimization based agglomerative algorithm that finds high-quality communities in a graph with a time complexity of $O(LM)$ and a space complexity of $O(N + M)$, where L is the number of iterations performed across all passes [17]. 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 $\Delta Q_{i:C_i\rightarrow C_j}$ (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 refer to the static version of this algorithm as Static_{L}.

2.2.2 Label Propagation Algorithm (LPA) [22]. *LPA* is a popular diffusion-based method for finding communities that is simpler, faster, and more scalable (due to its lower memory footprint) compared to *Louvain*. In *LPA*, every vertex i is initialized with a unique label (community id) C_i. Each vertex adopts the label with the most interconnecting weight at every step, as shown in Equation 3. It has a time complexity of $O(LM)$ and a space complexity of $O(N + M)$, where L is the number of iterations performed. We refer to the static version of this algorithm as $\text{Static}_{\text{LPA}}$.

$$C_i = \underset{c \,\in\, \Gamma}{\arg\max} \sum_{j\in J_i \,\mid\, C_j=c} w_{ij} \tag{3}$$

2.3 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 initial graph or 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 consider that the set of edges in Δ^t either insert an edge (i, j, w) or delete the edge (i, j), and not both. We refer to the setting where Δ^t consists of multiple edges deleted and inserted as a batch update. 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. Subsequent updates modify G^0.

2.4 Δ-screening approach [31] for Dynamic Graphs

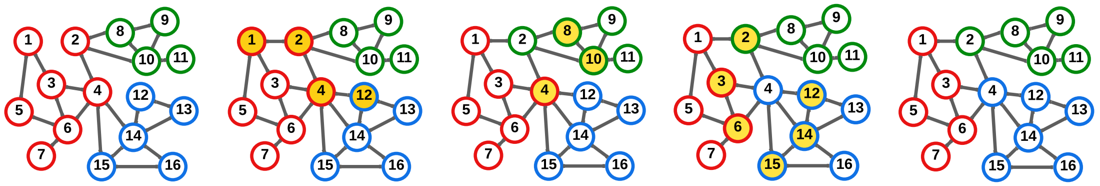
Δ-*screening* uses modularity to determine an approximate graph region where vertices are likely to change their community membership. Here, Zarayeneh et al. first separately sort the batch update consisting of edge deletions $(i, j) \in \Delta^{t-}$ and insertions $(i, j, w) \in \Delta^{t+}$ by their 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* apply to the first pass of *Louvain* algorithm, and the community membership of each vertex is initialized at the start of the algorithm to that obtained in previous snapshot of the graph. We note that Δ-*screening* is an approximate technique and is *not* guaranteed to explore all vertices that have the potential to change their community membership.

The Δ-*screening* technique, as proposed in [31], is not a parallel algorithm. We redesign it as a multicore parallel algorithm. To this end, we scan 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. We further optimize it by taking as input the previous total edge weight of each vertex and community and incrementally update them based on the batch update instead of recomputing from scratch which is costly. We refer to this parallel version of Δ-*screening* as P-DS. Its pseudocode is shown in Algorithm 1.

3 APPROACH

3.1 Our Dynamic Frontier approach (P-DF)



(a) Initial communities (b) Marking affected (initial) (c) After 1^{st} iteration (d) After 2^{nd} iteration (e) Communities converged

Fig. 1. In this example of the Dynamic Frontier approach (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 8 and 10. In the second iteration, vertex 4 changes from red to blue, affecting neighbors 3, 6, 12, 14, and 15. Afterward, there are no more community changes.

We now explain our parallel *Dynamic Frontier* approach, which we also refer to as P-DF. 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

Algorithm 1 Parallel Δ-*screening* approach.

```
□ $\delta E, \delta C$: Are neighbors, or community affected?
□ $H$: Hashtable mapping a community to associated weight

  function P-DS($G^t, \Delta^{t-}, \Delta^{t+}, C^{t-1}$)
      $H, \delta E, \delta C \leftarrow \{\}$
      for all $[i, j] \in \Delta^{t-}$ in parallel do
          if $C^{t-1}[i] = C^{t-1}[j]$ then
              $\delta E[i], \delta C[C^{t-1}[j]] \leftarrow true$
              Mark $i$ as affected
      for all unique source vertex $i \in \Delta^{t+}$ in parallel do
          $H \leftarrow \{\}$
          for all $(i', j, w) \in \Delta^{t+} \mid i' = i$ do
              if $C^{t-1}[i] \neq C^{t-1}[j]$ then
                  $H[C^{t-1}[j]] \leftarrow H[C^{t-1}[j]] + w$
          $[c^*, w^*] \leftarrow chooseCommunity(H)$
          $\delta E[i], \delta C[c^*] \leftarrow true$
          Mark $i$ as affected
      for all $i \in V^t$ in parallel do
          if $\delta E[i]$ then
              for all $j \in G^t.neighbors(i)$ do
                  Mark $i$ as affected
          if $\delta C[C^{t-1}[i]]$ then
              Mark $i$ as affected
```

undirected, i.e., if the edge (i, j, w) is in Δ^{t+}, so is the edge (j, i, w). At the start of the algorithm, we initialize the community membership of each vertex to that obtained in the previous snapshot of the graph.

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. [31] also observe. P-DF is thus an approximate algorithm for dynamic community detection, similar to P-DS.

We design P-DF to work with any algorithm to update the community labels of a specified set of affected vertices. Accordingly, in *Step 2*, the community membership of each vertex, obtained while running such an algorithm, is used by 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 to minimize unnecessary computation, we mark it as unaffected. 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 has converged. Finally, in *Step 3*, depending on the algorithm used, any necessary post-processing steps are performed as needed.

3.2 An Example of Dynamic Frontier approach (P-DF)

Figure 1 shows an example of P-DF. The original graph, shown in Fig. 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.

Subsequently, Figure 1(b) shows a batch update applied to the original graph involving the insertion of two edges: one between vertices 1 and 2, and another between vertices 4 and 12. To process this batch update, we perform the initial step of 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.

During the first iteration (see Figure 1(c)), let us consider the case 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 P-DF incrementally marks the neighbors of 2 as affected, specifically vertices 8 and 10. Vertex 2 is no longer marked as affected due to the pruning optimization.

During the second iteration (see Figure 1(d)), 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 12, 14, and 15. However, vertex 4 is no longer marked as affected due to vertex pruning.

In the subsequent iteration (see Figure 1(e)), 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 of the new graph.

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

```
▷ Step 1: Initial marking of affected vertices
function P-DF-initial($G^t, \Delta^{t-}, \Delta^{t+}, C^{t-1}$)
    for all $(i, j) \in \Delta^{t-}$ in parallel do
        Mark $i$ as affected if $C^{t-1}[i] = C^{t-1}[j]$
    for all $(i, j, w) \in \Delta^{t+}$ in parallel do
        Mark $i$ as affected if $C^{t-1}[i] \neq C^{t-1}[j]$

▷ Step 2: Incremental marking of affected vertices
function P-DF-incremental($G^t$)
    while iterations are not complete do
        for all $i \in V^t$ do
            if $i$ is not affected then continue
            Mark $i$ as not affected (prune)
            Pick best community for $i$
            if community of $i$ changes then
                Mark neighbors of $i$ as affected
        repeat until communities have converged

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

3.3 Our *Dynamic Frontier* based *Louvain* (P-DF$_L$)

We now show how to apply *Dynamic Frontier* approach (P-DF) to *Louvain* in Algorithm 3, which we call P-DF$_L$. We take as input the previous snapshot of the graph G^{t-1}, the batch update consisting of edge deletions Δ^{t-} and insertions

Algorithm 3 *Dynamic Frontier Louvain* (P-DF$_L$).

```
1:  function P-DF_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:         $l_i \leftarrow louvainMove(G', C', K', \Sigma')$
12:         if $l_i \leq 1$ 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($G', C', K', \Sigma'$)
24:     for all $l_i \in [0..MAX\_ITERATIONS)$ 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 $l_i$
```

Δ^{t+}, the previous community membership of each vertex C^{t-1}, the previous total edge weight of each vertex K^{t-1}, and the previous total edge weight of each community Σ^{t-1} in Line 1. By using the vertex and community weights of the previous graph G^{t-1} as auxiliary information to the dynamic algorithm, we are able to quickly obtain the updated vertex and community weights. 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 P-DF, we mark the initial set of vertices as affected.

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

```
function P-DF_LPA($G^t, \Delta^{t-}, \Delta^{t+}, C^{t-1}$)
    P-DF-initial($G^t, \Delta^{t-}, \Delta^{t+}, C^{t-1}$)            ▷ See Alg. 2
    Vertex membership: $C' \leftarrow C^{t-1}$ ; $G' \leftarrow G^t$
    for all $l_i \in [0..\text{MAX\_ITERATIONS})$ do
        $\Delta N \leftarrow lpaMove(G', C')$
        if $\Delta N / N \leq \tau$ then break                        ▷ Converged?
    return $C^{t-1} \leftarrow C'$

function lpaMove($G', C'$)
    Changed vertices: $\Delta N \leftarrow 0$
    for all affected $i \in V'$ in parallel do
        Mark $i$ as not affected (prune)
        $c^* \leftarrow$ Most weighted label to $i$ in $G'$
        if $c^* = C'[i]$ then continue
        $C'[i] \leftarrow c^*$ ; $\Delta N \leftarrow \Delta N + 1$
        Mark neighbors of $i$ as affected
    return $\Delta N$
```

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

```
function P-DF_H($G^t, \Delta^{t-}, \Delta^{t+}, C^{t-1}$)
    if $t = 0$ then return $\text{Static}_{\text{L}}(G^t)$
    return P-DF_LPA($G^t, \Delta^{t-}, \Delta^{t+}, C^{t-1}$)
```

Initialization phase (Lines 5-8): Then, we initialize the community membership of each vertex C, 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's below an aggregation tolerance (τ_{agg}, optimal value in Sec. 5.1), we stop to avoid the computationally 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 3. 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 [20], 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 quality. Once all necessary passes are complete, we perform the dendrogram flattening in Line 21. Finally, in Line 22, we return the community membership of each vertex C, and the total edge weight 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 P-DF$_\text{L}$. For each iteration (Lines 24-33), and for each affected vertex i in the graph G', we mark i as not affected as vertex pruning step of 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^* in parallel using Equation 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 and atomically update the total edge weights linked to each community in Lines 31-32. 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.

Application for P-DF to first pass of Louvain in P-DF$_L$: Note that we apply P-DF only to the first pass of *Louvain*, as with P-DS. In subsequent passes, if the aggregation tolerance 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 P-DF to find affected super-vertices. The tolerance condition only fails with large updates.

Correctness of P-DF$_L$: In Sec. 4.1, we show the correctness of Algorithm 3.

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

We now show how to apply *Dynamic Frontier* approach (P-DF) to *LPA* in Algorithm 4, which we call 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.

First, in Line 2, based on 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 as vertex pruning step of 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 Equation 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 P-DF, we mark neighbors of vertex i as affected in Line 15.

Correctness of P-DF$_{LPA}$: We show the correctness of Algorithm 4 in Sec. 4.2.

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

Louvain is known for its high-quality community detection but at the cost of being slow. On the other hand, *LPA* is fast, but the communities it detects are of moderate quality [14]. We combine the strengths of both algorithms by creating a Hybrid dynamic algorithm, which we call P-DF$_\text{H}$. To this end, we use *Louvain* as the base/static method and *LPA* as the dynamic method (i.e., P-DF$_\text{H}$). This provides us with superior performance, while obtaining communities with high modularity score. See Algorithm 5 for details.

3.6 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., $O(N + M)$.

4 CORRECTNESS

We now provide arguments for the correctness of P-DF$_{\text{L}}$ and P-DF$_{\text{LPA}}$. To help with this, we refer the reader to Fig. 2. Here, pre-existing edges are represented by solid lines, and i represents a source vertex of edge deletions/insertions in the batch update. Edge deletions in the batch update with i as the source vertex are shown in the top row (denoted by dashed lines), edge insertions are shown in the middle row (also denoted by dashed lines), and community migration of vertex i is is shown in the bottom row. Vertices i_n and j_n represent the destination vertices (of edge deletions or insertions). Vertices i', j', and k' signify neighboring vertices of vertex i. Finally, vertices i'', j'', and k'' represent non-neighbor vertices (to vertex i). Yellow highlighting is used to indicate vertices marked as affected, initially or in the current iteration of the community detection algorithm. We understand this figure is dense, but we tried to capture several details for correctness arguments in Sections 4.1 and 4.2.

4.1 Correctness of Dynamic Frontier based Louvain (P-DF$_{\text{L}}$, Algorithm 3)

Given a batch update consisting of edge deletions Δ^{t-} and insertions Δ^{t+}, we now show that 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. 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 the a vertex, as well as the indirect cascading effect of migration of a vertex (to another community) on other vertices.

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

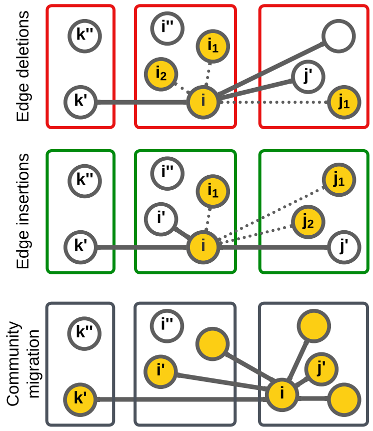


Fig. 2. *Dynamic Frontier* approach in detail.

4.1.1 On edge deletion.

Lemma 4.1. *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 Fig. 2, where $j = i_1$). 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 has a significant positive factor w/m. 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 community membership, and no incentive for a change in community 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 Static$_{\text{L}}$ every 1000 batch updates, which also helps us ensure high-quality communities. The same applies to P-DS$_{\text{L}}$.

(1) $\Delta Q^{new}_{i:d\to b} = \Delta Q_{i:d\to b} + [\frac{w}{m}] + \frac{w}{2m^2}(\Sigma_c - \Sigma_d + w)$
(2) $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} - \frac{wK_{i''}}{m^2}$
(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., $C_i = d$, $C_j = c$ (see Fig. 2, where $j = j_2$ or j_3). Let i'' be a vertex belonging to i's community $C_{i''} = d$, j'' be a vertex belonging to j's community $C_{j''} = c$, 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.

(4) $\Delta Q^{new}_{i:d\to c} = \Delta Q_{i:d\to c} - \frac{w}{m} + \frac{w}{2m^2}(2K_i + \Sigma_c - \Sigma_d - w)$
(5) $\Delta Q^{new}_{i:d\to b} = \Delta Q_{i:d\to b} + \frac{w}{2m^2}(K_i + \Sigma_b - \Sigma_d)$
(6) $\Delta Q^{new}_{i'':d\to c} = \Delta Q_{i'':d\to c}$
(7) $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} - \frac{wK_{i''}}{2m^2}$
(8) $\Delta Q^{new}_{k'':b\to d/c} = \Delta Q_{k'':b\to d/c} + \frac{wK_{k''}}{m^2}$ ◇

4.1.2 On edge insertion.

Lemma 4.2. *Given an edge insertion (i, j, w) between vertices i and j belonging to different communities d and c, 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 $C_j = c$ respectively (see Fig. 2, where $j = j_3$). Let i'' be a vertex belonging i's community $C_{i''} = d$, j'' be a vertex belonging to j's community $C_{j''} = c$, 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''.

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

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

(11) $\Delta Q^{new}_{i'':d\to c} = \Delta Q_{i'':d\to c}$

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

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

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 Fig. 2, where $j = i_1$ or i_2). 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 more strongly connected communities, but it is very unlikely.

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

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

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

4.1.3 On vertex migration to another community.

Lemma 4.3. *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 c (see Fig. 2). 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 $C_{j'} = C_{j''} = c$, 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 (as thus necessitate marking), but not j'. However, to keep the algorithm simple, we simply mark all the neighbors of vertex i as affected.

(17) $\Delta Q^{new}_{i':d\to c} = \Delta Q_{i':d\to c} + [\frac{2w_{ii'}}{m}] - \frac{K_iK_{i'}}{m^2}$

(18) $\Delta Q^{new}_{i':d\to b} = \Delta Q_{i':d\to b} + [\frac{w_{ii'}}{m}] - \frac{K_iK_{i'}}{2m^2}$

(19) $\Delta Q^{new}_{j':c\to d} = \Delta Q_{j':c\to d} - \frac{2w_{ij'}}{m} + \frac{K_iK_{j'}}{m^2}$

(20) $\Delta Q^{new}_{j':c\to b} = \Delta Q_{j':c\to b} - \frac{w_{ij'}}{m} + \frac{K_iK_{j'}}{2m^2}$

(21) $\Delta Q^{new}_{k':b\to d} = \Delta Q_{k':b\to d} - \frac{w_{ik'}}{m} + \frac{K_iK_{k'}}{2m^2}$

(22) $\Delta Q^{new}_{k':b\to c} = \Delta Q_{k':b\to c} + [\frac{w_{ik'}}{m}] - \frac{K_iK_{k'}}{2m^2}$

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 [31]. However, this case is automatically taken care of as we perform marking of affected vertices during the community detection process.

(23) $\Delta Q^{new}_{i'':d\to c} = \Delta Q_{i'':d\to c} + \frac{K_i K_{i''}}{m^2}$
(24) $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} - \frac{K_i K_{i''}}{2m^2}$
(25) $\Delta Q^{new}_{j'':c\to d} = \Delta Q_{j'':c\to d} + \frac{K_i K_{j''}}{m^2}$
(26) $\Delta Q^{new}_{j'':c\to b} = \Delta Q_{j'':c\to b} + \frac{K_i K_{j''}}{2m^2}$
(27) $\Delta Q^{new}_{k'':b\to d} = \Delta Q_{k'':b\to d} + \frac{K_i K_{k''}}{2m^2}$
(28) $\Delta Q^{new}_{k'':b\to c} = \Delta Q_{k'':b\to c} - \frac{K_i K_{k''}}{2m^2}$ ◇

4.1.4 Overall. Finally, based on Lemmas 4.1, 4.2, and 4.3, we can state the following for P-DF$_{\text{L}}$.

THEOREM 4.4. *Given a batch update, P-DF$_{\text{L}}$ marks vertices having an incentive to change their community membership as affected.* □

We note that with P-DF$_{\text{L}}$, 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 P-DS$_{\text{L}}$ 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 (modularity) of the communities obtained, as shown in Section 6.

4.2 Correctness of Dynamic Frontier based LPA (P-DF$_{\text{LPA}}$, Algorithm 4)

Given a batch update consisting of edge deletions Δ^{t-} and insertions Δ^{t+}, we now show that 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 is determined as given in Equation 5. 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 a vertex, along with the indirect cascading effect of the change of label of a vertex on the label associated with other vertices.

$$C_i = \underset{c \,\in\, \Gamma}{\arg\max} \sum_{j \in J_i \;|\; C_j = c} w_{ij} \tag{5}$$

4.2.1 On edge deletion.

LEMMA 4.5. *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 $C^{new}_i = \arg\max\{[d, K_{i\to d} - w], ...\}$. 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 $C_j = c$ respectively. The new label for vertex i would be $C_i^{new} = \arg\max\{(d, K_{i\to d}), ...\}$. 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. ◇

4.2.2 *On edge insertion.*

Lemma 4.6. *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 $C_j = c$. The new label for vertex i would be $C_i^{new} = \arg\max\{(d, K_{i\to d}), (c, K_{i\to c} + w)\}$. Here, c 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 $C_i^{new} = \arg\max\{(d, K_{i\to d}) + w), ...\}$. 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. ◇

4.2.3 *On vertex migration to another community.*

Lemma 4.7. *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 $C_i^{new} = c$. 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 $C_{j'} = d$, 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 perform marking of affected vertices during the community detection process.

(29) $C_{i'}^{new} = \arg\max\{(d, K_{i'\to d} - w), (c, K_{i'\to c} + w)\}$
(30) $C_{j'}^{new} = \arg\max\{(c, K_{j'\to c} + w)\}$
(31) $C_{k'}^{new} = \arg\max\{(d, K_{i'\to d} - w), (c, K_{i'\to c} + w)\}$ ◇

4.2.4 *Overall.*

Finally, based on Lemmas 4.5, 4.6, and 4.7, we can state the following for P-DF$_{\text{LPA}}$.

Theorem 4.8. *Upon a given batch update, P-DF$_{LPA}$ marks any vertices that could change their labels as affected.* □

5 IMPLEMENTATION DETAILS

5.1 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, 12]. 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 of the algorithm.

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 their memory addresses.

5.1.1 Results with optimized implementation. Our parallel *Louvain* has a runtime of 6.2 seconds on the undirected `sk-2005` graph containing 1.9 billion edges, and is 93× faster than Vite, an MPI+OpenMP implementation of Louvain method for undirected graph clustering [9]. 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) [27].

5.2 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.

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.

5.2.1 Results with optimized implementation. Our parallel *LPA* has a run time of 2.7 seconds on the undirected `sk-2005` graph containing 1.9 billion edges. We observe that graphs with a lower average degree (road networks and protein k-mer graphs) have a larger time/$|E|$ factor.

6 EVALUATION

6.1 Experimental Setup

6.1.1 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 and all programs are compiled with the -O3 flag enabled. Unless mentioned otherwise, we use 64 threads to match the number of cores available on the system.

Table 3. List of 8 graphs obtained from the *SuiteSparse Matrix Collection* [16] (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

6.1.2 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://bit.ly/ipdps24-375.

6.1.3 Dataset. Table 3 shows the graphs we use in our experiments. All of them are obtained from the SuiteSparse Matrix Collection [16]. The number of vertices in the graphs varies from 3.07 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.

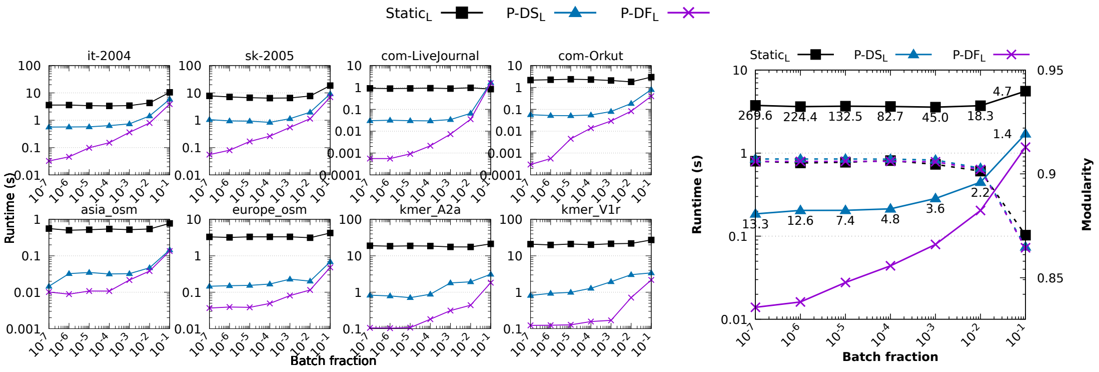
6.1.4 Batch generation. We take a base graph from the dataset and generate a random batch update [31] consisting purely of edge deletions or insertions for simplicity [7], each with an edge weight of 1. All batch updates are undirected, i.e., for every edge insertion (i, j, w) in the batch update, the 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 batches having various distributions of mixed edge deletions and insertions is part of our future work.

6.1.5 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. We employ multiple random batch updates for each batch size and report the average across these runs in our experiments.

6.2 Performance of P-DF$_L$

6.2.1 Overall Performance. We first study the performance of P-DF$_L$ on batch updates of size varying from $10^{-7}|E|$ to $0.1|E|$, and compare it with Static$_L$ and P-DS$_L$ (P-DS applied to *Louvain*).

Fig. 3(b) shows the average results of the experiment. We observe the following from Fig. 3(b). The modularity of communities obtained by both P-DF$_L$ and P-DS$_L$ is nearly identical to that obtained by Static$_L$. P-DF$_L$ converges the



(a) Results of each graph

(b) Overall result

Fig. 3. Time taken (solid lines), and modularity of communities obtained (dashed lines) along the Y2 axis, with Static$_{\text{L}}$, P-DS$_{\text{L}}$, and P-DF$_{\text{L}}$ (Algorithm 3) on batch updates of increasing size from $10^{-7}|E|$ to $0.1|E|$. Note that both axes are logarithmic. The numbers on the lines corresponding to Static$_{\text{L}}$ and P-DS$_{\text{L}}$ indicate the speedup of P-DF$_{\text{L}}$ over the two algorithms, respectively.

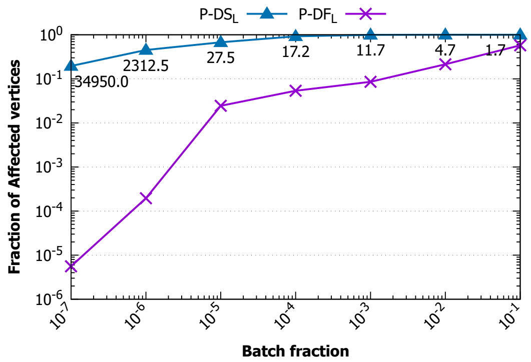
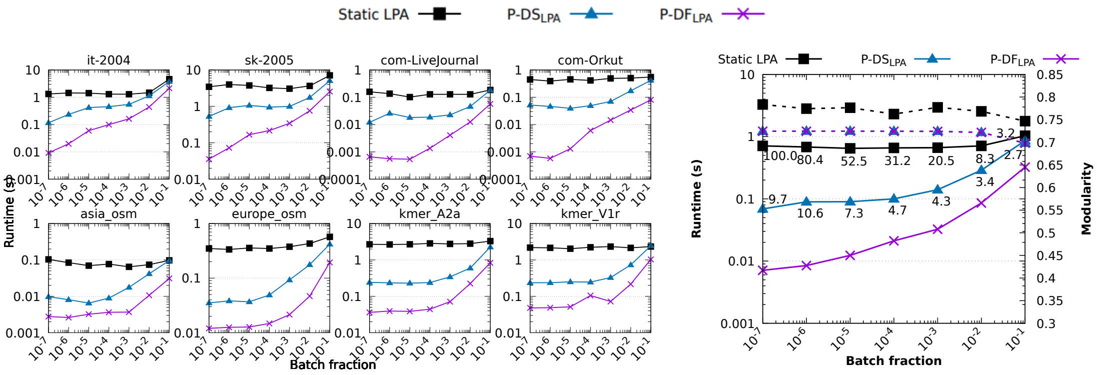


Fig. 4. Fraction of vertices marked as affected (average) with P-DS$_{\text{L}}$ and P-DF$_{\text{L}}$, as mentioned in Section 6.2, on graphs in Table 3. The numbers on the line corresponding to P-DS$_{\text{L}}$ indicate the ratio of the fraction of vertices identified as affected by P-DS$_{\text{L}}$ to that of P-DF$_{\text{L}}$.

fastest with an average speedup of 128.8× over Static$_{\text{L}}$, and 7.3× over P-DS$_{\text{L}}$. As the batch size increases, the number of vertices marked as affected by P-DF$_{\text{L}}$ increases. This increases the time taken by P-DF$_{\text{L}}$ as the batch size increases. Further, as Fig. 3(a) shows, dynamic approaches significantly outperform Static$_{\text{L}}$ on *social networks*, *road networks*, and *k-mer protein 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 P-DF$_{\text{L}}$ and Static$_{\text{L}}$ across graphs and batch sizes is less than 0.004. Therefore, the average modularity score is shown only in Fig. 3(b) with the modularity score anchored to the Y2-axis. Fig. 3(b) also shows the average execution time is shown through the primary Y-axis along with average speedup numbers with respect to Static$_{\text{L}}$ and P-DS$_{\text{L}}$. Finally, 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.



(a) Results of each graph

(b) Overall result

Fig. 5. Time taken (solid lines), and modularity of communities obtained (dashed lines) along the Y2 axis, with Static$_{\text{LPA}}$, P-DS$_{\text{LPA}}$, and P-DF$_{\text{LPA}}$ (Algorithm 4) on batch updates of increasing size from $10^{-7}|E|$ to $0.1|E|$. Note that both axes are logarithmic. The numbers on the lines corresponding to Static$_{\text{LPA}}$ and P-DS$_{\text{LPA}}$ indicate the speedup of P-DF$_{\text{LPA}}$ over the two algorithms, respectively.

6.2.2 *Affected vertices and Performance.* We now study the difference in the number of vertices marked as affected by P-DF$_{\text{L}}$ and P-DS$_{\text{L}}$. In Fig. 4, we show the fraction of vertices marked as affected by P-DF$_{\text{L}}$ and P-DS$_{\text{L}}$ on average over the instances in Table 3. The numbers on the line corresponding to P-DS$_{\text{L}}$ in Fig. 4 shows the ratio of the number of vertices marked affected by P-DS$_{\text{L}}$ to that of P-DF$_{\text{L}}$.

We notice from Fig. 4 that P-DF$_{\text{L}}$ marks a significantly smaller fraction of vertices as affected compared to P-DS$_{\text{L}}$. The run time of these algorithms, however, do not differ in this ratio as observed from Fig. 3(b). This is because of the following reasons. P-DS$_{\text{L}}$ 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.

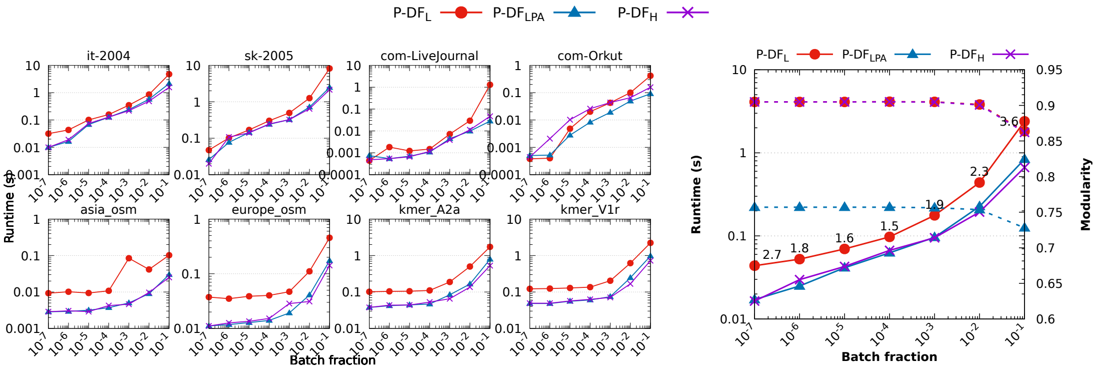
6.2.3 *Stability of P-DF$_{L}$.* Intuitively, if the graphs G^t and $G^{t'}$ are identical for some t and t', we expect P-DF$_{\text{L}}$ to produce the same communities for G^t and $G^{t'}$. We refer to this property of a dynamic algorithm as its stability, measured as the percentage of vertices that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

To measure the stability of P-DF$_{\text{L}}$ and P-DS$_{\text{L}}$, we proceed as follows. Let G be an initial graph. We generate a batch update 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 inserts 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 observe that P-DS$_{\text{L}}$ has a minimum of 99.68% match with the original community memberships across all batch sizes. P-DF$_{\text{L}}$ has a minimum of 99.70% match, thus indicating that both algorithms are stable.

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

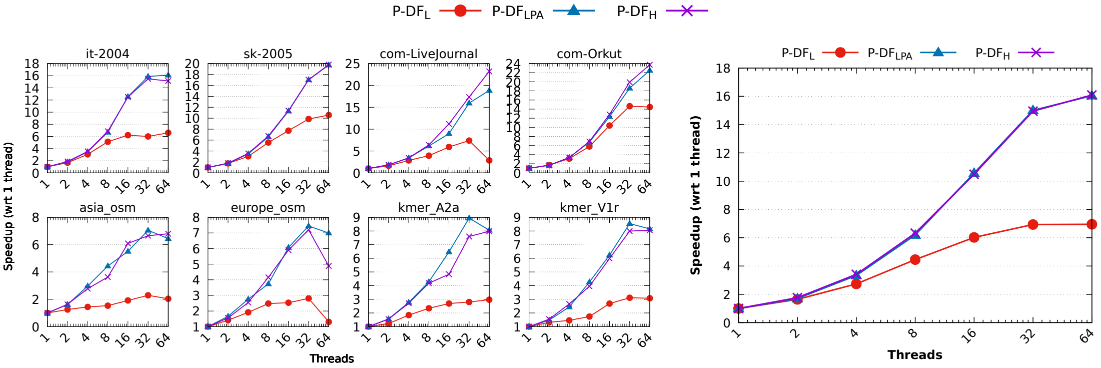
6.3.1 *Overall Performance.* In this experiment, we study the performance of P-DF$_{\text{LPA}}$ with random batch updates of size ranging from $10^{-7}|E|$ to $0.1|E|$, and compare it to Static$_{\text{LPA}}$ and P-DS$_{\text{LPA}}$ (P-DS applied to *LPA*).



(a) Results of each graph

(b) Overall result

Fig. 6. Time taken (solid lines), and modularity of communities obtained (dashed lines) along the Y2 axis, with P-DF$_L$, P-DF$_{LPA}$, and P-DF$_H$ (Algorithms 3, 4, and 5) 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 P-DF$_L$ indicates the speedup of P-DF$_H$ over P-DF$_L$.



(a) Results of each graph

(b) Overall result

Fig. 7. Strong scalability result of P-DF$_L$, P-DF$_{LPA}$, and P-DF$_H$ (Algorithms 3, 4, and 5) on batch updates of size $10^{-3}|E|$. The number of threads is doubled from 1 to 64. Note that the Y-axis is linear, while the X-axis is logarithmic and captures doubling of threads naturally.

Fig. 5(b) shows the average result of the experiment. While all approaches obtain communities of equivalent modularity, P-DF$_{LPA}$ converges on average 48.8× faster than Static$_{LPA}$, and 6.7× faster than P-DS$_{LPA}$ from a batch size of $10^{-7}|E|$ up to $0.01|E|$. As shown in Fig. 5(a), it has good performance on *web graphs* and *social networks* (graphs with high $|E|/|V|$ ratio). Note, however, that the quality of communities obtained with *LPA* is not on par with *Louvain*.

6.3.2 Stability of P-DF$_{LPA}$. We study the stability of P-DF$_{LPA}$ using the notion of stability described in Section 6.2.3. We observe that P-DF$_{LPA}$ identifies the same community label for a minimum of 95.75% of the vertices.

6.4 Performance of P-DF$_H$ (Algorithm 5)

6.4.1 Overall Performance. We now study the performance of P-DF$_H$ on batch updates of size $10^{-7}|E|$ to $0.1|E|$, and compare it with P-DF$_L$ and P-DF$_{LPA}$ (Algorithms 3, 4).

The average modularity of communities obtained by P-DF$_{\text{H}}$ is nearly identical to that obtained by P-DF$_{\text{L}}$, as shown in Fig. 6(b) while obtaining a mean speedup of 5.0× across batch sizes of $10^{-7}|E|$ to $0.01|E|$. P-DF$_{\text{H}}$ is thus an efficient and high-quality dynamic community detection approach, especially on web graphs, as shown in Fig. 6(a), where it significantly outperforms P-DF$_{\text{L}}$ for smaller batch updates. Note that P-DF$_{\text{LPA}}$ has about the same performance but obtains communities of lower quality.

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

Finally, we study the strong-scaling behavior of P-DF$_{\text{H}}$ and compare it with P-DF$_{\text{L}}$ and 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 Fig. 7(b), P-DF$_{\text{L}}$, P-DF$_{\text{LPA}}$, and P-DF$_{\text{H}}$ obtain a speedup of 7.0×, 16.1×, and 16.1× respectively at 64 threads; with their speedup increasing at a mean rate of 1.38×, 1.59×, and 1.59× respectively for every doubling of threads. The Y-axis of Fig. 7(b) 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 Fig. 7(a) that P-DF$_{\text{H}}$ offers good speedup on web graphs and social networks, but does not scale well on *road networks* and *k-mer* graphs (with low $|E|/|V|$ ratio). We can also observe from Fig. 7(b) that for some of the graphs (it-2004, com-Orkut, com-LiveJournal, 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. This lack of enough work leads to a decrease in speedup that can be achieved with respect to one thread.

6.6 Further Discussion

6.6.1 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 [26, 31]. Our algorithms do not need any changes to handle a batch update consisting of a mix of insert and delete operations.

6.6.2 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. Further, since these minor changes equally impact our algorithms and the Δ-screening based algorithms, we expect the performance to be along similar lines.

6.6.3 Comparison with respect to Riedy and Bader [25]. Riedy and Bader 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 the batch sizes of $0.08|E|$, $0.03|E|$ and $0.06|E|$, they report a speedup of 40.1×, 10.8×, and 327× over their corresponding static algorithm. On these three graphs and respective batch sizes, P-DF$_{\text{H}}$ achieves a speedup of 41.7×, 29.5×, and 14.5×, 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.

7 CONCLUSION

This paper addressed the design of high-speed community detection algorithms in the batch dynamic setting. We presented our algorithms through the *Dynamic Frontier* approach. We showed that this approach, when applied to our hybrid algorithm, produces high-quality results while being 14.6× faster than state-of-the-art, and identifying communities with the same quality score.

REFERENCES

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

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

[3] A. Bhowmick, S. Vadhiyar, and PV. Varun. 2022. Scalable multi-node multi-GPU Louvain community detection algorithm for heterogeneous architectures. *CCPE* 34, 17 (2022), 1–18.

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

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

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

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

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

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

[10] A. Ghoshal, N. Das, S. Bhattacharjee, and G. Chakraborty. 2019. A fast parallel genetic algorithm based approach for community detection in large networks. In *11th International Conference on Communication Systems & Networks (COMSNETS)*. IEEE, Bangalore, India, 95–101.

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

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

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

[14] Bin Hu, Wenmin Li, Xuesong Huo, Ye Liang, Minghui Gao, and Pei Pei. 2016. Improving Louvain algorithm for community detection. In *2016 International Conference on Artificial Intelligence and Engineering Applications*. Atlantis Press, 110–115.

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

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

[17] A. Lancichinetti and S. Fortunato. 2009. Community detection algorithms: a comparative analysis. *Physical Review. E* 80(5) (2009), 056117:1–11.

[18] X. Liu, M. Halappanavar, K. Barker, A. Lumsdaine, and A. Gebremedhin. 2020. Direction-optimizing Label Propagation and its application to community detection. In *Proc. of ACM CF*. 192–201.

[19] X. Meng, Y. Tong, X. Liu, S. Zhao, X. Yang, and S. Tan. 2016. A novel dynamic community detection algorithm based on modularity optimization. In *7th IEEE international conference on software engineering and service science (ICSESS)*. IEEE, Beijing,China, 72–75.

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

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

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

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

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

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

[26] Heli Sun, Jianbin Huang, Xin Zhang, Jiao Liu, Dong Wang, Huailiang Liu, Jianhua Zou, and Qinbao Song. 2014. IncOrder: Incremental density-based community detection in dynamic networks. *Knowledge-Based Systems* 72 (2014), 1–12.

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

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

[29] Y. Xing, F. Meng, Y. Zhou, M. Zhu, M. Shi, and G. Sun. 2014. A node influence based label propagation algorithm for community detection in networks. *Sci. World J.* 2014 (2014), 1–14.

[30] X. You, Y. Ma, and Z. Liu. 2020. A three-stage algorithm on community detection in social networks. *Knowl.-Based Syst.* 187 (2020), 104822:1–12.

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

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