Parallel Louvain Algorithms with Convergence Guarantee

Jason Niu*¶, M. Yusuf Özkaya†§, Ahmet Erdem Sarıyüce‡§, Ümit V. Çatalyürek†§
*JHU APL, Laurel, MD, USA
†Georgia Institute of Technology, School of Computational Science and Engineering, Atlanta, GA, USA
‡University at Buffalo, Buffalo, NY, USA
jniu0881@gmail.com, myozka@gatech.edu, erdem@buffalo.edu, umit@gatech.edu

***Abstract*—Community detection is a fundamental problem in network science. The Louvain algorithm is a widely used hierarchical method applied to scientific, social, biological, and commercial networks. Scaling Louvain to large networks requires parallelization, but existing parallel implementations lack convergence guarantees and often rely on outdated modularity data due to concurrent updates. We present the first parallel Louvain algorithms with provable convergence. We give two distinct parallelization schemes for Louvain's local-move phase: (1) distance-2 community coloring, an expository approach that prevents concurrent access to the same community at high cost, and (2) community versioning, which enforces consistency through version numbers. Both ensure convergence, filling a key theoretical gap. Our OpenMP-based implementation shows that community versioning achieves strong empirical performance on large, real-world networks, matching the speed and modularity quality of state-of-the-art methods while providing formal convergence guarantees.**

***Index Terms*—community detection, Louvain, modularity, coloring, versioning**

I. Introduction

Identifying communities is a key challenge in network science, with uses spanning from analyzing social networks [1] to studying biological systems [2]. Communities represent dense node clusters where nodes within a community are more tightly connected to each other than to nodes outside the community [3]–[6]. Among the most widely used methods for this task is the Louvain algorithm, which optimizes modularity, a measure of partition quality [7].

The Louvain algorithm is a hierarchical method for community detection, with applications spanning analysis of scientific, social, biological, and commercial networks, such as identifying influencer groups and similar communities, identifying protein-protein interaction communities, and identifying fraud rings [8]–[19] In epidemiology, Louvain reveals how ecological factors influence viral spread [18]; in healthcare, it identifies physicians' specialties via prescription data [8]; and in bioinformatics it detects functional and disease pathways in protein networks and analyzes brain circuits [14], [15]. Similarly, Louvain supports bibliometric clustering in tools like CitNetExplorer and VOSviewer [9] and refines U.S. labor market delineations [13]. Beyond these fields, Louvain has been applied to social media [12], computer vision [10], and time-series clustering [11]. Notably, Louvain (and its improved variant, Leiden [17]) now serves as fundamental components of the emerging GraphRAG framework, which integrates knowledge graph construction with query-focused summarization in LLMs to enhance human sense-making [19]. Together, these applications underscore the Louvain algorithm's enduring significance as a unifying tool for revealing hidden structures and advancing understanding across an extraordinary range of complex systems.

Louvain algorithm has an iterative two phase process: (1) local-move, where nodes are greedily moved to communities to maximize modularity, and (2) aggregation, where communities are collapsed into super-nodes to form a coarser graph. After each phase, a new level of communities is obtained. Blondel et al. [7] proved that Louvain algorithm converges after a finite number of levels, and empirical evaluations showed a small number of levels is sufficient for convergence. Leiden [17] is proposed to improve Louvain by introducing an additional refinement phase before aggregation to yield communities with better quality. While Louvain (and Leiden) has been shown to be highly effective, it suffers from scalability to large networks, motivating the need for parallel implementations.

There have been extensive research on parallelizing Louvain on shared-memory [20]–[23], distributed-memory [24], [25], and accelerators [26], [27]. For shared-memory parallelization, Staudt and Meyerhenke [22] introduced PLMR, which has a refinement phase that checks for further node movements after a parallel local-move phase where communities are updated atomically. Halappanavar et al. [20], [21] introduced the Grappolo framework that employs distance-1 graph coloring for concurrent node processing during the local-move phase. Grappolo partitions nodes into color groups such that no two adjacent nodes share the same color, allowing nodes of the same color to update their communities in parallel. Therefore, since nodes of any distance can be in the same community, there may be instances of stale community information when

§Amazon Web Services. This publication describes work performed at the Georgia Institute of Technology and University at Buffalo, and is not associated with AWS.

¶The work was done when the author was at University at Buffalo.

multiple nodes are considering the same community at once. Halappanavar et al. have shown that no parallel approach that uses a distance-k coloring, for any k, can guarantee convergence. Tithi et al. [23] proposed a parallel pull-and-push Louvain algorithm that prunes unnecessary edge explorations in the local-move phase. However, these parallel algorithms often operate with stale modularity information to improve the runtime and concurrency and hence cannot provide convergence guarantees, a key feature in the original sequential Louvain. This limits their reliability on large-scale networks and can result in unpredictable behavior. In this work, we give the first parallel Louvain algorithms for which convergence is guaranteed, filling a key gap in prior work.

We introduce two parallelization schemes for Louvain's local-move phase: distance-2 community coloring-based parallelization and community versioning via two-phase locking. In the former, we devise a new coloring scheme where two vertices get different colors if their communities or neighboring communities overlap. Albeit inefficient, this static scheduling mechanism highlights the key idea to prevent simultaneous access to the same community, and hence prevents the use of stale modularity information. We prove that our distance-2 community coloring-based parallel approach is serializable—can be matched to a sequential Louvain run—and hence guarantees convergence. In the community versioning approach, we apply the same idea but in a dynamic way: each community employs version numbers to facilitate a two-phase locking mechanism so that the local move updates always use the latest community information. We again prove that our approach is serializable. Our OpenMP-based shared-memory implementation shows that community versioning achieves practical performance on large real-world networks, matching state-of-the-art parallel Louvain methods—while also providing formal convergence guarantees. Our algorithms are also applicable to Leiden which uses the same local-move phase.

Provable convergence ensures that our new parallel algorithms deliver stability, reproducibility, and faithful adherence to the theoretical guarantees of the original Louvain method, even at modern, industrial scales. Unlike existing high-performance parallel heuristics, which can be fast but unpredictable, our approach provides outcomes that are dependable, verifiable, and consistent across runs and platforms. This reliability is not merely a theoretical result: it can enable industry practitioners to adopt, trust, and further build upon our algorithms for safe and predictable deployment in large-scale systems. In regulated domains where correctness and traceability are essential, such as finance, healthcare, and security, as well as mission-critical environments, these guarantees offer meaningful operational value that current heuristic methods cannot provide. Ultimately, our methods introduce a new point on the design spectrum: we trade a modest amount of speed for rigorous correctness and reproducibility, unlocking a stability-focused alternative that complements rather than competes with purely performance-driven heuristics.

Our contributions can be summarized as follows:

- *Parallel Louvain methods.* We propose two parallelization schemes for the local-move phase in Louvain: distance-2 community coloring and community versioning.
- *Theoretical convergence guarantee.* Unlike the state-of-the-art parallel Louvain methods which are not guaranteed to converge in the local-move phase, we prove that our methods do converge.
- *Evaluation on real-world networks.* We evaluate our approach on various real-world networks. We compare the runtime of our algorithms for differing numbers of threads to demonstrate scalability and practical runtimes.

II. Preliminaries

We work on a simple and undirected graph $G = (V, E)$ where V is the set of nodes and E is the set of edges. The neighbors of a node v are denoted by $N(v)$. deg_{max} is the maximum degree in the graph. We denote $|V|$ as n (number of nodes) and $|E|$ as m (number of edges). $C : V \to \mathbb{N}$ is a mapping of nodes to communities where the community of a node v is denoted as $C(v)$. $\mathcal{C}$ denotes the set of all communities, also called a partition. In the later phases of Louvain, we form aggregated graphs where super-nodes have weighted self loops and edges. w_{uv} represents the edge weight between two connected super nodes u, v. The weighted degree k_v is the sum of weights of all edges incident to $v \in V$, i.e., $k_v = \sum_{u \in N(v)} w_{uv}$. For parallel time and space complexities, we represent the number of processing units as p.

A. Louvain

The Louvain algorithm [7] is a greedy method for community detection that optimizes the modularity Q:

$$Q = \frac{1}{2m} \sum_{u,v \in V} \left[w_{uv} - \frac{k_u k_v}{2m} \right] \delta(C(u), C(v)) \qquad (1)$$

where $\delta(C(u), C(v)) = 1$ if $C(u) = C(v)$ and 0 otherwise.

Algorithm 1 outlines Louvain. Initially, each node is a community of its own, called singleton communities. The algorithm performs in a loop until there is no change in the community structure. Each iteration in the main loop (lines 3-5) is called a *level*, representing a coarser level of hierarchy in the community detection process. Computation in each level contains two alternating phases, a local-move phase (LocalMove) and a community aggregation phase (Aggregate).

In LocalMove, each vertex is attempted to be moved to a neighbor's community if the modularity score improves. The whole computation is again performed iteratively, where each pass through vertices is called an *iteration* (lines 9-14). In each iteration, we go over all the vertices, evaluate their move to all the neighboring nodes' communities by computing the modularity gain, and perform the move if beneficial. Note that the order of processing the vertices is not fixed, and any order

Algorithm 1: LOUVAIN $(G = (V, E))$

```
1  C ← {{v} | v ∈ V} // Singleton communities
2  while true do
       // Move nodes to best communities
3      change ← LocalMove (G, C)
       // Terminate if no change happens
4      if change = false then break
       // Each community becomes a node
5      Aggregate (G, C)
6  return C
```

LocalMove $(G = (V, E), \mathcal{C})$:

```
7  change ← false // Track the change
8  while true do
9      Q_old ← Q(C)
       // Process all nodes in any order
10     (parallel) foreach v ∈ V do
           // Find the best move for v ...
11         C' ← arg max_{C∈{C(w)|w∈N(v)}} ΔQ_C(v ↦ C)
           // ... apply if it is positive
12         if ΔQ_C(v ↦ C') > 0 then v ↦ C'
       // Stop when modularity stops improving
13     if Q(C) ≤ Q_old then break
14     change ← true // Flag the change
15 return change
```

Aggregate $(G = (V, E), \mathcal{C})$:

```
   // Each community becomes a node
16 V ← C
   // Inter-community edges are set
17 E ← {(C_1, C_2) | (u, v) ∈ E, u ∈ C_1, v ∈ C_2}
   // Partitioning is reset to singletons
18 C ← {{v} | v ∈ V}
```

yields an acceptable output. The modularity gain ΔQ from moving u into v's community is expressed as follows:

$$\Delta Q = \left[\frac{\Sigma_{in} + k_{u,in}^{C(v)}}{2m} - \left(\frac{\Sigma_{tot} + k_u}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2 - \left(\frac{k_u}{2m}\right)^2\right] \quad (2)$$

where Σ_{in} is the sum of weights inside v's community, Σ_{tot} is the sum of weights incident to v's community, and $k_{u,in}^{C(v)}$ is the sum of weights from u to nodes in v's community. This process is repeated until no vertex can be moved.

In AGGREGATE, community aggregation is performed where communities are condensed into new super-nodes. The edges within a community are aggregated and denoted as a self-loop to the super-node. Edge weights among theses super-nodes are also aggregated and computed. If G is initially unweighted, then the weight of all edges is set to one. Each iteration in LOCALMOVE takes $O(m)$ time. The convergence of the algorithm has been proven [7]. In practice, only tens of iterations and fewer levels are needed to terminate on most real world inputs. This can be further improved by defining a threshold on the change in modularity for early termination. Blondel et al. [7] argue that Louvain is observed to run in $O(n \log n)$ time for most real-world networks.

B. Convergence Issue in Parallel Louvain

All the parallel Louvain algorithms parallelize the computation over all vertices in LOCALMOVE (line 10). However, this approach breaks the convergence guarantee of the sequential Louvain algorithm. The following two lemmas, by Halappanavar et al. [21], explain the reasoning:

Lemma 1 (Negative gain)**.** *If community updates for nodes are performed in parallel, then the net modularity gain achieved cannot be guaranteed to be always positive.*

Nodes that use stale modularity information may move to communities that do not increase the total modularity, resulting in no theoretical guarantee that the algorithm will terminate.

Lemma 2 (Infinite swaps)**.** *If nodes are moved to neighboring communities using stale modularity information, then nodes may enter an infinite cycle of swaps without modularity improvement.*

If two neighboring nodes are moved to each other's community concurrently, there may be no modularity gain. Future iterations may swap them back, potentially resulting in an infinite loop. Both issues are caused by stale modularity information. In practice, however, this lack of convergence guarantee does not impact the termination of the algorithm. Using thresholds for early termination also helps with practical runs.

C. Coloring

Distance-k coloring is an assignment of colors to the vertices of a graph such that no vertices within distance k of each other share the same color. When $k = 1$, it ensures no adjacent nodes have the same color. The goal in a coloring process is to minimize the total number of colors used while maintaining a valid coloring. However, minimizing the number of colors in a distance-k coloring is an NP-Hard problem [28]. Greedy (or first fit) coloring is an effective heuristic that iterates over the vertices in some order and assigns the minimum available color to the processed node. A color class is the set of all vertices in the graph that share the same color.

III. RELATED WORK

Here we review related work on shared-memory parallel Louvain computation. Halappanavar et al. [20], [21] introduced Grappolo which uses distance-1 coloring to process nodes in parallel. The authors showed that any distance-k coloring-based approach cannot preserve the convergence guarantees of the sequential Louvain algorithm—we give their results in Lemma 1 and Lemma 2 in Section II-B. Staudt and Meyerhenke [22] proposed PLMR, which adds a refinement phase after the parallel local-move phase where community updates are performed atomically. The refinement phase addresses possible concurrency issues with parallel updates during the local-move phase by performing additional node movement iterations which check for further node movements using updated community information. They also proposed to compute incremental modularity gains instead of moving a

vertex and recomputing the modularity from scratch. Tithi et al. [23] proposed a parallel pull-and-push Louvain algorithm that prunes unnecessary edge explorations. For the local-move phase, the authors proposed processing only the neighbors of nodes that have moved as those nodes are the only ones that may have to change communities in the next iteration.

None of the existing parallel Louvain algorithms provide convergence guarantees. They are shown to perform well and converge in practice, with the help of additional threshold-based early termination conditions.

IV. Algorithms

We introduce two parallelization methods for LocalMove in Louvain: distance-2 community coloring in Section IV-A and community versioning in Section IV-C. Our proposed methods only consider the local-move phase of Louvain. For both methods, we prove convergence guarantees in Section IV-B and Section IV-D, respectively.

For the community aggregation phase and the full Louvain setup, we use the same approach as Halappanavar et al. [21]. Each community is assigned a lock to avoid duplicate processing. The summation of edge weights across communities is then done in parallel with the locking mechanism to avoid race conditions where multiple threads may update the edge weights for the same community. Finally, communities are aggregated into super-nodes in parallel with updated edge weights.

For both of our algorithms, we adopt two optimizations previously introduced for parallel Louvain computation:

1) When finding the best move for a vertex (line 11 inAlgorithm 1), there is no need to do from scratch modularity computation for every attempted move. Staudt and Meyerhenke [22] introduced a modularity change equation that computes gain without any initial removal step. We use the same formula in our algorithms:
$$\Delta Q = k_{u,\text{in}}^{C(v)} - k_{u,\text{in}}^{C(u)} + \frac{k_u(\Sigma_{\text{tot}}^{C(u)} - \Sigma_{\text{tot}}^{C(v)})}{2m} \quad (3)$$
where $\Sigma_{\text{tot}}^{C(v)}$ is the sum of weights of edges incident to community $C(v)$, and $k_{u,\text{in}}^{C(v)}$ is the sum of weights from u to nodes in community $C(v)$.
2) Tithi et al. [23] studied how some nodes can be pruned during the local-move iterations without a notable loss in modularity. They show that it is safe to prune nodes that are not neighbors of a previously moved node. They also show that the majority of node movement occurs in the first few iterations. Inspired by these findings, after two iterations in LocalMove, we only consider the neighbors of nodes whose communities have changed.

Note that these optimizations provide practical speedup and have no effect on the convergence or overall time and space complexities. The modularity change equation eliminates Louvain's initial node removal step, simplifying the calculation of modularity gain without altering the correctness or complexity. The pruning strategy reduces the number of processed nodes and does not affect the theoretical time or space complexities nor alter Louvain's convergence properties.

A. D2CC: Community Coloring Approach

We now describe our coloring-based parallel Louvain algorithm, D2CC. We propose to use a custom coloring procedure in the local-move phase to create workloads that can be run in parallel *without breaking the convergence guarantees of the Louvain algorithm*. The second phase of Louvain stays the same. As in the typical coloring schemes for parallel graph algorithms, nodes with the same color are processed in parallel. Our coloring scheme prevents nodes from being moved using stale community information and also does not let multiple nodes be moved from/to the same community concurrently.

Halappanavar et al. [21] considered distance-1 coloring for parallelizing Louvain. However, it does not guarantee convergence as nodes which are exactly 2-hop neighbors may have the same color and can be moved to the same community at the same time. This may result in negative gains in modularity and hence violates the convergence guarantee (see Lemma 1 in Section II of [21]). Furthermore, Halappanavar et al. showed that using even distance-k coloring does not guarantee convergence because the impact on community structure can cascade through any distance. For instance, in local-move phase of the first iteration, distance-2 coloring ensures that nodes with the same color cannot be moved to the same community, as each node is a singleton community. However, in future iterations, communities may contain nodes further than distance-2 away and as such, the coloring needs to be updated to ensure there are no conflicts. Therefore, any coloring scheme that ignores communities cannot guarantee convergence in Louvain.

We introduce **the distance-2 community coloring** to ensure that the nodes with the same color are not involved in conflicting computations. In the distance-2 community coloring, two nodes have different colors if the set of their communities and neighboring communities are *not* disjoint. In other words, we color communities and enforce distance-2 constraint among them. We give the formal definition as follows.

Definition 1. *Given a graph $G = (V, E)$ with a community assignment $C : V \to \mathbb{N}$, a **distance-2 community coloring** is a node coloring assignment $\mathbf{d2cc} : V \to \mathbb{N}$ such that for any two nodes $u, v \in V$, $\mathbf{d2cc}(u) \neq \mathbf{d2cc}(v)$ if the following condition holds:*
$$\left(\bigcup_{w \in \{u\} \cup N(u)} C(w) \right) \cap \left(\bigcup_{w \in \{v\} \cup N(v)} C(w) \right) \neq \varnothing.$$
In other words, if $d2cc(u)$ and $d2cc(v)$ are the same, then the intersection of the sets of their communities and neighboring communities is disjoint.

When each node is a singleton community, as at the beginning of the algorithm, this corresponds to distance-2 coloring. Furthermore, at any point of the algorithm, all the nodes in the

Algorithm 2: DISTANCE-2 COMMUNITY COLORING (G, C, col, u)

Input: $G\ (V, E)$: graph, C: communities of nodes, $color$: colors of nodes, u: node

1 $NC \leftarrow \{C(u)\}$ // Communities to avoid
2 **foreach** $v \in N(u)$ **do** add $C(v)$ to NC
3 $used \leftarrow \emptyset$ // Colors to avoid conflict
4 **parallel foreach** $c \in NC$ **do**
5 | **foreach** $v \in c$ *(s.t.* $u \neq v$*)* **do**
6 | | add $color[x]$ to $used$ where $x \in v \cup N(v)$
7 $color[u] \leftarrow$ minimum color not in $used$

same community have a unique color. For simplicity, we utilize greedy (first-fit) coloring where each node is assigned the first unused color, and the order of processing is arbitrary. For distance-2 coloring, the natural lower bound for the number of colors is the maximum degree + 1. We incorporate our distance-2 community coloring method into the local-move phase as follows:

1) In the first iteration, simply compute the greedy distance-2 coloring of nodes, as it is identical to distance-2 community coloring because each node is a singleton community. Then, process the nodes with the same color in parallel during the Louvain local-move phase and continue with community aggregation.
2) Starting in the second iteration, compute the distance-2 community coloring of nodes **incrementally**, by simply updating the colors of the nodes that changed communities after the previous level. Algorithm 2 outlines the **incremental distance-2 community coloring** procedure. Then, process the nodes with the same color in parallel in the local-move phase and continue with the community aggregation.
3) Repeat step 2 until no node has moved communities.

Algorithm 2 is performed for each node that has changed its community. We obtain a valid distance-2 community coloring by only considering the new community of the affected node, and do not optimize the number of colors. Given a node whose community changed, we first collect its community and the communities of its adjacent nodes in NC (Lines 1-2). Then, for each considered community, we go over the nodes it has and collect the colors of those nodes as well as their neighbor nodes (Lines 3-6). Lastly, the color of the moved node is updated to the minimum color that has not been collected (Line 7).

Lemma 3. *Assume V has a valid distance-2 community coloring. After the communities are updated in the local-move phase, applying Algorithm 2 to each node whose communities have changed, $V' \subseteq V$, maintains a valid distance-2 community coloring.*

Proof. Say $u \in V'$ has changed its community. By Definition 1, we need to ensure that u's new color is different from (1) all the nodes that belong to any community in the set $S = \bigcup_{w \in \{u\} \cup N(u)} C(w)$, and (2) all the other nodes that are not in any community in S but are neighbors to a node in any community in S. Lines 1-2 ensure (1). Going over the neighbors of v in Line 6 ensures (2). □

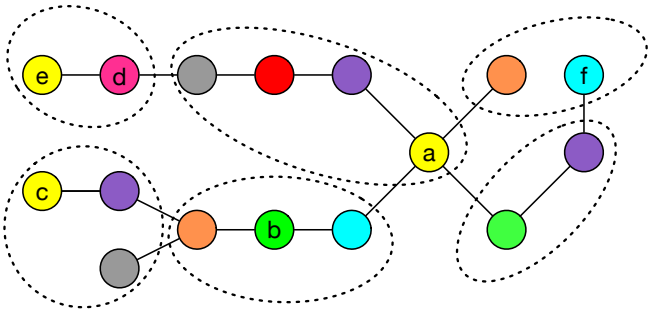

Fig. 1: An example toy graph with a valid distance-2 community coloring. Dashed regions denote communities.

Figure 1 illustrates a toy graph with a valid distance-2 community coloring. Communities are denoted as dashed regions. Note that for any pair of nodes that share the same color; set of its communities and neighbors' communities are disjoint. The color of node a cannot be the same with any node in the four communities on the right (like b and f) as well as any node that has a neighbor in those communities (like d). The only nodes that do not satisfy these constraints are c and e, hence they can be assigned the same color with a.

Time complexity. In the first iteration, we apply greedy distance-2 coloring which takes $O(n \cdot deg_{max}^2/p)$ when parallelized with p processors. In the following iterations, a single run of Algorithm 2 first goes over the neighbors (Line 2 and 4), $O(deg_{max})$ in the worst case, then iterates over the nodes in each community (Line 5), taking $O(|C_{avg}|)$ time where $|C_{avg}|$ is the average community size, and finally goes over neighbors in the inner-most loop (Line 6), taking $O(deg_{max})$ in the worst case. In total Algorithm 2 takes $O(|C_{avg}| \cdot deg_{max}^2/p)$ as it is parallelized with p processors. In situations where the majority of nodes have been moved, Algorithm 2 needs to be applied to up to $O(n)$ nodes, hence the additional time complexity due to coloring in the local-move phase is $O(n \cdot |C_{avg}| \cdot deg_{max}^2/p)$, or $O(n^2 \cdot deg_{max}^2/p)$ as $|C_{avg}|$ is $O(n)$. Overall, overhead of our coloring method is $O(k \cdot n^2 \cdot deg_{max}^2/p))$ where k is the number of iterations. Note that this is higher than sequential Louvain, which takes $O(k \cdot m)$.

Space complexity. We use three additional containers of size $O(n)$—NC, $used$, and $coloring$—which do not change the space complexity of the local-move phase in sequential Louvain, which is $O(n + m)$.

B. Proof of Convergence for D2CC

We prove that DISTANCE-2 COMMUNITY COLORING-based parallel Louvain algorithm, D2CC, guarantees the convergence at the end of computation by showing that the processing order of vertices can be serialized to a sequential Louvain computation proposed by [7]. To achieve that, we first show that the vertices with the color can be processed in any

order and all the color classes can be serialized and mapped to a sequential Louvain computation.

We first show that for any parallel step S that the coloring allows to run concurrently, the set S satisfies the following independence guarantee.

Lemma 4 (Pairwise independence in a coloring class)**.** *In a given color class S and any pair of nodes $u, v \in S$, the set of communities updated by u's move is disjoint from the set of communities updated by v's move.*

Proof. If u and v have the same colors, then $\left(\bigcup_{w\in\{u\}\cup N(u)} C(w)\right) \cap \left(\bigcup_{w\in\{v\}\cup N(v)} C(w)\right) = \varnothing$, by Definition 1. The left (right) part contains all the communities that are read or could be updated when vertex u (v) is being processed in the local-move phase. As the intersection of those two sets is empty, the proof follows. □

This guarantee allows formal reasoning about the ordering of vertex-move operations, as defined below.

Definition 2. *Given current partition $\mathcal{C}$, and a vertex v; a* **vertex-move operation is defined as** *$\mathcal{C}' = mv_v(\mathcal{C})$ where v is moved to its chosen best community (computed by Louvain's local gain formula), and a new partition $\mathcal{C}'$ is returned.*

We first show the commutativity of a pair of vertex-move operations in a given color class S:

Lemma 5 (Independence $\rightarrow$ Commutativity)**.** *Let $u, v \in S$ be two distinct vertices that are allowed by the coloring to be processed concurrently in a coloring class S that satisfies Lemma 4. Then the operations mv_u and mv_v commute for any partition $\mathcal{C}$: $mv_u(mv_v(\mathcal{C})) = mv_v(mv_u(\mathcal{C}))$.*

Proof. By Lemma 4, the communities that $mv_u(C)$ updates are disjoint from those $mv_v(C)$ updates, and neither operation reads values that the other is simultaneously writing. Therefore performing one move does not change the inputs used by the other move (both see either the same values or disjoint values), and the two updates modify disjoint parts of the global state. Hence, the resulting partition after applying both updates are identical regardless of order. □

Next, we generalize the same property to all vertices in a given color class S.

Lemma 6 (Collective independence in a coloring class)**.** *Let $S = v_1, \ldots, v_k$ be a coloring class processed concurrently in one parallel step. For any permutation π of $1, \ldots, k$ and any partition $\mathcal{C}$, $mv_1(mv_2(...mv_k(\mathcal{C}))) = mv_{\pi(1)}(mv_{\pi(2)}(...mv_{\pi(k)}(\mathcal{C})))$.*

Proof. We can repeatedly apply pairwise commutativity from Lemma 5 to reorder any pair $i, j \in \{1, ..., k\}$ when computing $mv_1(mv_2(...mv_k(C)))$, hence any permutation π of $1, \ldots, k$ results in the same partition. □

We now prove that a parallel computation by our algorithm can be serialized to a sequential Louvain computation.

Lemma 7 (Serializability of a single iteration)**.** *In a single iteration, consider a complete parallel execution that is a sequence of parallel steps $S_1, S_2, , \ldots, S_t$ where each S_i is a color class and has a collective independence as described in Lemma 6. Then, any permutation of this sequence can be serialized to a sequential Louvain computation proposed by Blondel et al. [7].*

Proof. Each step S_i can be replaced by a sequential permutation of its moves, to S_i without changing the resulting partition, by Lemma 6. Then, concatenating permutations of each $S_i \in \{S_1, S_2, , \ldots, S_t\}$ obtains a sequential Louvain computation which is guaranteed to converge when applied as a whole until there is no community change occurs. □

Lastly, we give our main theorem that generalizes the previous lemma to the entire computation.

Theorem 1 (Serializability)**.** *D2CC is serializable and the processing order of vertices corresponds to a sequential Louvain computation.*

Proof. Proof simply follows from Lemma 7. The color classes produced by DISTANCE-2 COMMUNITY COLORING based parallel Louvain computation have the collective independence property. In our algorithm we do not enforce any ordering on processing the color classes, and any arbitrary order of colors will give a valid sequential execution of Louvain. □

C. VLM: Community Versioning Approach

The distance-2 community coloring based approach in Section IV-A outlines the race conditions very well but makes the worst case assumption that significantly limits opportunities for parallelism. In general, even distance-2 coloring in real-world networks is known to yield too many colors, hence increasing the computation depth, and also most of the resulting color classes are small [29]. Furthermore, coloring procedure, albeit being incremental, has a computational overhead.

We propose to resolve the community conflicts, outlined in Definition 1, in a dynamic way during the computation by utilizing two-phase locking mechanism with community versioning [30]. We maintain a version number for each community and ensure that a community is in stable state (i.e., not being updated) when we read/write it. We define that a community is in stable state if it has an even version number. For read operations, we only make sure that the community being read is in a stable state (i.e., has an even version number). For write (update) operations, we ensure that the version of the community being updated has not changed since it is read. Prior to a write operation, we use atomic compare-and-swap (CAS) operations to check the version number of the community and obtain a lock (i.e., increment version number).

Algorithm 3, VERSIONEDLOCALMOVE (VLM) gives our community versioning algorithm for the local-move phase in Louvain; it simply replaces LOCALMOVE procedure in Algorithm 1. We initialize version numbers for each community to 0 in ver (line 1) and also keep track of active nodes that will

Algorithm 3: VLM: VERSIONEDLOCALMOVE $(G, \mathcal{C})$

```
1  ver ← [0] // Versions of comms; even=stable
2  iter ← 0 // Iteration counter
3  hasChanged ← True // Flag for comm. change
4  curr ← [True] // Active nodes
5  updated ← False // Return value
6  while hasChanged do
7      iter++, hasChanged ← False
8      next ← [False] // To visit in next iter.
       // Process active nodes in parallel
9      parallel foreach u ∈ V | curr(u) do
10         C_self ← C(u), V_self ← ver(C_self)
           // PART I: Skip if unstable
11         if V_self is odd then
12             next(u) ← True // Defer
13             hasChanged ← True
14             continue
           // PART II: Evaluate neighbor comms
15         bestMod ← 0
16         C_best, V_best ← C_self, V_self // Best comm.
17         foreach v ∈ N(u) do
18             C_neig ← C(v), V_neig ← ver(C_neig)
               // Stop processing if not stable
19             if V_neig is odd then
20                 bestMod ← 0
21                 next(u) ← True // Defer
22                 break
23             else if V_neig is even and C_neig ≠ C_self then
                   // Compute modularity gain
24                 Q ← ΔQ_C(u ↦ C_neig)
                   // Update best if improves
25                 if Q > bestMod then
26                     bestMod ← Q
27                     C_best, V_best ← C_neig, V_neig
           // PART III: Attempt the move
28         if bestMod > 0 then
               // Lock C_self and C_best for update
               // Ordering to prevent livelock
29             C_1, V_1 ← C_self, V_self; C_2, V_2 ← C_best, V_best
30             if C_best < C_self then
31                 C_1, V_1 ← C_best, V_best; C_2, V_2 ← C_self, V_self
               // Atomic compare for C_1 ...
32             if V_1 = ver(C_1) then
33                 ver(C_1)++ // ... update
                   // Atomic compare for C_2 ...
34                 if V_2 = ver(C_2) then
35                     ver(C_2)++ // ... update
36                     C(u) ← C_best // Move u to C_best
37                     ver(C_1)++, ver(C_2)++ // Unlock
                       // Mark self and neighbors
38                     next(v) ← True ∀ v ∈ {u ∪ N(u)}
39                     updated ← True // To return
40                     hasChanged ← True
41                 else
42                     ver(C_1)-- // Revert
43                     next(u) ← True // Defer
44             else
45                 next(u) ← True // Defer
       // Set to deferreds/markeds after it. 2
46     if iter ≥ 2 then curr ← next
47 return updated
```

be processed in the current iteration in $curr$ (line 4). The main loop in line 6 is performed in iterations until no community update happens. Before each iteration, the $next$ is initialized to keep track of active nodes that will be processed in the next iteration (line 8). In each iteration, the nodes that are marked as active (in $curr$) are shared among the threads for processing (lines 9-45).

When processing a node u, there are three main parts: (I) Checking whether the community of u is unstable (i.e., being updated by another thread) and skipping if so (lines 11-14); (II) Evaluating the neighbor communities for possibly moving u (lines 15-27); and (III) Attempting to move u to the best found neighbor community (lines 28-45). Part I is simply done by making sure that community of u has an even version number; otherwise u is deferred to be processed in the next iteration and the thread continues with the next vertex in line 9. In part II, a possible move of u to each neighboring community is evaluated and the best move is chosen. Note that when a neighboring community is evaluated, we ensure that it is in a stable state (line 23), and we also save the best community and its version for part III (lines 16 and 27). If any neighboring community is observed to be in an unstable state (line 19), we stop processing the neighbors and simply defer the processing of u to the next iteration.

In part III, we attempt to move vertex u to the best neighboring community, if the gain is positive. The move happens from u's old community, C_{self}, to its new community, C_{best}, hence we need to update both and make sure that they have not changed since we read them in the earlier parts. Before the actual update in line 36, we utilize atomic CAS operations to verify that the saved versions of C_{self} and C_{best} are up-to-date (lines 32 and 34). After the update, we mark the impacted nodes for processing in the next iteration (line 38), this optimization comes from [23] as described early in Section IV. We enforce a total ordering to process them in order to prevent potential livelock issues. In both cases, we secure the lock for both communities by incrementing their version numbers, in lines 33 and 35; both become an odd number. To release the lock, we increment them again in line 37. Note that those incrementing operations do not need to be atomic. If we cannot secure the lock for C_{self} or C_{best}, we simply defer processing u to the next iteration (lines 43 and 45) If one is locked but the other is not, we release the acquired lock by decrementing its version number (line 42). After all the active nodes are processed, we set $curr$ to $next$ to obtain next set of active nodes (line 46). As the modularity changes the most for the first two iterations, it is recommended to process all the nodes in the first two iterations [23].

Time complexity. Our changes do not incur any additional time complexity to sequential Louvain described in Algorithm 1 which takes $O(n \log n)$. As we parallelize in line 9, the time complexity of the new parallel Louvain algorithm that uses VLM stays as $O(\frac{n \log n}{p})$ where p threads are used.

Space complexity. In Algorithm 3, we use additional global containers to keep community related information, such as ver, $curr$, and $next$, that takes $O(n)$ space. Locally, each parallel thread has an additional $O(n)$ container (not shown in the pseudocode) to keep track of the weighted degrees from u to each community, taking a total of $O(p \cdot n)$ space. As the graph takes $O(n + m)$ space, the total space complexity of VLM is $O(p \cdot n + m)$.

D. Proof of Convergence for VLM

Our proof is based on showing that VLM is serializable to a sequential Louvain computation. Each successful vertex-move obtains exclusive rights to its source and destination communities by incrementing their ver registers (odd). Because readers only consult communities with even versions and the moving thread re-checks versions via CAS before acquiring locks, the modularity gain is computed on a snapshot that is guaranteed unchanged at commit. The commit (release making versions even again) is the linearization point. CAS ensures two moves touching the same community cannot both commit from the same snapshot, so all commits touching the same community are serialized. Moves touching disjoint communities commute. Therefore every concurrent execution is equivalent to executing committed moves in commit-order in a sequential Louvain local-move phase. Aborted/deferred attempts do not modify global state and thus do not affect serializability.

We first give four invariants about the atomicity of the operations in VLM.

- Iff $ver(C)$ is even, then community C is stable, i.e., there is no move currently holding lock and making changes. Iff $ver(C)$ is odd, then some thread holds a lock and will either commit (make changes and bump $ver(C)$ to even) or revert.
- Any thread that reads community data when $ver(C)$ is even has read a stable snapshot of C's content and connections to itself and other communities.
- If a thread succeeds in CASing $ver(C)$ from v to $v + 1$, then no other thread can succeed in a CAS from v simultaneously; CAS ensures mutual exclusion for version v.
- When a thread has incremented $ver(C)$ (from even to odd), it holds exclusive rights to mutate the corresponding C and its edge information to itself and other communities. No other thread can read these communities as stable (they will see odd and defer) and no other thread can acquire the same $ver(C)$ lock because CAS requires equality to $ver(C)$.

We now define the commit point in our algorithm that helps characterizing the serialization of vertex-move operations.

Definition 3 (Commit point)**.** *Commit point (CP) for a successful vertex-move of u to C_{best} is the point where VLM finishes releasing locks in line 37, i.e., the last increment operations that make both community versions even again. At this moment the move's changes become visible to any thread that later reads even versions of those two communities.*

Next, we argue that the community snapshot before the gain computation and at CP are the same.

Lemma 8 (Snapshot validity)**.** *If a thread T computes gain for vertex u using a community snapshot defined by stable versions V_{self} and V_{best} (line 24), and later manages to CAS both community versions such that they are still equal to those values (lines 33 and 35) (so, locks succeed), then the modularity gain computation is based on the same community snapshot that the commit will apply to.*

Proof. T read community and connections only when versions were even, saved the V values and community ids. Before committing, T verifies the saved V values by CAS (ensuring current versions are still equal to saved Vs), then increments to lock. If CASs succeed, no other committed move could have changed those communities since the saved V (any such change would have incremented ver and the CAS would fail). Thus the community information and connections used for computing gain is the same data at commit time. □

The following two lemmas show the exclusivity on conflicting communities and commutativity of non-conflicting moves.

Lemma 9 (Mutual exclusion on conflicting communities)**.** *Two moves that touch the same community cannot both commit simultaneously or interleave to produce a conflicting community snapshot—CAS + lock ensures commits that touch the same community are serialized.*

Proof. Suppose two moves try to acquire lock for the same community version ver. CAS only allows one succeed. The other fails, aborts (defers), and retries, and eventually will see a different version. Therefore commits touching the same community occur in some total order determined by which thread successfully acquired the necessary CAS and completed release. □

Lemma 10 (Commutativity of non-conflicting vertex-moves)**.** *Vertex-moves that touch disjoint sets of communities are commutable and can be ordered arbitrarily.*

Proof. If two committed vertex-moves touch disjoint communities (no community is shared) then they do not rely on each other's community snapshot and each acquires locks over different community sets. Because they touch disjoint parts of snapshot, applying them in either order yields the same final community snapshot. □

We now show that deferring some computations in a given iteration to the next iteration does not break the convergence guarantee of the Louvain computation.

Theorem 2 (Convergence of Louvain with deferred vertices)**.** *Consider the local-move phase of the sequential Louvain algorithm applied where in each iteration some vertices may be deferred and processed in later iterations. Note that (i) each executed move strictly increases modularity, and (ii) every vertex is eventually considered (fair scheduling). Then the*

algorithm terminates in a finite number of moves at a partition that is locally optimal with respect to single-vertex moves.

Proof. Let $Q(\mathcal{C})$ denote the modularity of partition (set of communities) $\mathcal{C}$. Each executed vertex move yields a strictly positive gain $\Delta Q > 0$, so the sequence of modularity values $Q_0 < Q_1 < Q_2 < \cdots$ is strictly increasing. Since modularity is bounded above and there are finitely many distinct partitions of V, only finitely many distinct modularity values can occur. Hence, only finitely many improving moves can be executed. Deferring vertices changes only the order of consideration, not the set of admissible moves, and by the fairness assumption every vertex is eventually processed. Therefore the process terminates after finitely many moves at a partition where no single-vertex move yields a positive modularity gain, i.e., a local optimum. □

Finally, we give our main theorem that argues that VLM guarantees convergence.

Theorem 3 (Serializability of VLM)**.** *Parallel Louvain computation by VLM is serializable: the processing order of vertices corresponds to a sequential Louvain execution.*

Proof. Take any concurrent execution with committed vertex-moves $mv_1, ..., mv_k$. Order them by the real time of their commit points (CP)—i.e., the time when the releasing increments make versions even. For each committed move mv_i, its computed gain was based on a community snapshot right before its CP (by Lemma 8) and no earlier commit that occurs after that snapshot could have modified any of the communities used by mv_i (otherwise the CAS would have failed). Therefore applying the sequence of committed moves in CP order yields exactly the same final community snapshot as the concurrent execution. Non-commuting moves are already ordered by lock acquisition/commit (by Lemma 9), commuting ones can be arbitrarily ordered (by Lemma 10). Deferred computations only change the order of processing, by Theorem 2. Hence there exists a sequential Louvain local-move execution (applying the same moves in CP order), with deferred vertices processed in later iterations, that is observationally equivalent (i.e., producing identical community structures). □

V. Experiments

In this section, we present experimental evaluation of our Distance-2 Community Coloring-based parallel approach (D2CC) and VersionedLocalMove (VLM) approaches presented in Section IV against the baselines: sequential Louvain [7], Grappolo [20], [21] (Grap.), and PLMR [22], and, to be able to present a more-fair comparison, our custom implementation of Grappolo that uses the incremental gain computation [22] and pull/push improvement [23] that we called Grappolo++ or Grap++ in short.

Table I presents the statistics of the real-world datasets from SNAP [31], that we used in our experiments [1]. `Facebook` contains friendships on Facebook [32]. `Enron` contains email addresses which sent an email to each other using Enron [33]. `DBLP` connects pairs of co-authors with papers in the DBLP computer science bibliography [34]. `Skitter` is an internet topology graph collected by skitter in 2005 [35]. `Youtube`, `LiveJournal`, and `Orkut` are the networks of users with friendships on Youtube, LiveJournal, and Orkut [34].

TABLE I: Statistics of the real-world networks used in the experiments. deg_{max} is the maximum degree of a node and d_c^2 is the number of distinct colors after distance-2 node coloring.

Networks	n	m	deg_{max}	d_c^2
`Facebook`	4,039	88,234	1,045	1,046
`Enron`	36,692	183,831	1,383	1,384
`DBLP`	317,080	1,049,866	343	344
`Youtube`	1,134,890	2,987,624	28,754	28,755
`Skitter`	1,696,415	11,095,298	35,455	35,456
`LiveJournal`	3,997,962	34,681,189	14,815	14,816
`Orkut`	3,072,441	117,185,083	33,313	33,314

TABLE II: Average modularity over four runs (best in bold). D2CC timed out after 24 hours for `LiveJournal` and `Orkut`.

Networks	Louv.	Grap.	Grap++	PLMR	D2CC	VLM
`Facebook`	0.835	0.835	0.835	**0.836**	**0.836**	0.835
`Enron`	0.610	0.611	0.586	0.610	0.613	**0.619**
`DBLP`	0.821	0.815	0.816	**0.827**	0.823	0.820
`Youtube`	0.720	0.719	0.718	0.701	**0.725**	0.705
`Skitter`	0.841	0.841	0.837	0.845	**0.847**	0.845
`LiveJournal`	0.750	0.744	0.751	**0.763**	>24h	0.758
`Orkut`	0.666	0.665	0.679	0.682	>24h	**0.683**

All experiments are performed on a Linux operating system running on a machine with Intel Xeon Gold processor at 2.1 GHz and 256 GB DDR4 memory[2]. The server contains 2 CPUs with each having 20 cores for a total of 40 cores. We implemented our algorithms on top of Grappolo codebase in C++ with OpenMP 4.5 [37] and compiled using GCC 10.2.0 at the -O3 level. Our code is publicly available at **https://github.com/erdemUB/IPDPS26.** PLMR runs each level for at most 32 iterations.

A. Community Quality

We first check the resulting communities by our methods and the baselines. Regarding the community quality, Table II lists the average modularity results of sequential Louvain, our algorithms, and parallel baselines on 40 threads. All methods achieve comparable modularity scores across the tested networks, with only minor variations. While the best scores (in bold) are typically achieved by different methods depending on the dataset, the D2CC and VLM approaches consistently produce results that are either the best or very close to the best in each case. This confirms that our convergence guarantees let us maintain competitive modularity performance, even matching or nearly matching the leading algorithms on most networks. As expected, our more theoretical algorithm D2CC proves that it will not be useful in practice, since it is unable to complete the execution in 24h on the two largest datasets (`LiveJournal` and `Orkut`).

[1] One interesting observation is that the number of distance-2 colors is equal to maximum degree + 1 for all datasets, which is the lower bound

[2] This research used resources from the Center for Computational Research at the University at Buffalo [36]

TABLE III: Total number of communities detected by each method. All except Louvain are run with 40 threads. D2CC timed out after 24 hours for `LiveJournal` and `Orkut`.

Networks	Louv.	Grap.	Grap++	PLMR	D2CC	VLM
`Facebook`	16	15	15	14	17	16
`Enron`	1,389	1,180	1,290	1,255	1,248	1,241
`DBLP`	779	167	157	111	227	207
`Youtube`	10,856	5,027	5,256	4,863	6,436	4,672
`Skitter`	3,082	981	1,451	1,196	1,243	1,029
`LiveJournal`	2,323	1,678	3,256	2,409	>24h	1,938
`Orkut`	27	32	21	21	>24h	28

Table III shows the number of communities detected by each method. The original Louvain method (sequential) generally identifies more communities than all parallel methods tested. On `Enron`, `Youtube`, `Skitter`, and `LiveJournal`, Grap++ finds the highest number of communities among the parallel methods, with PLMR and VLM yielding slightly fewer. For `Youtube`, D2CC detects the most granular community structure, whereas VLM and the other parallel methods provide more conservative counts. Overall, the differences between methods are minor on smaller networks but become more pronounced on larger networks.

B. Runtime Comparison

We now provide the runtime results of baseline sequential and parallel Louvain algorithms against our solutions. Table IV gives the average runtime results over four runs of sequential Louvain as well as our algorithms and parallel baselines on 40 threads. As expected, Grap++ is consistently better than Grappolo, $2.8\times$ faster on average. PLMR is the fastest overall method on most networks, partly due to the early termination condition that sets the maximum number of iterations to 32 in each level. D2CC, our distance-2 community coloring approach (Section IV-A), often has the highest runtime due to its significantly greater time complexity ($O(\frac{n^4}{p})$) compared to sequential Louvain ($O(nlogn)$) [7] as well as the expensive overhead of distance-2 coloring operation. VLM, our community versioning approach (Section IV-C), avoids the coloring step and performs quite competitive with the baselines, even gives the best runtime in two networks: `DBLP` and `LiveJournal`. Compared to the original sequential Louvain implementation, the only baseline that guarantees convergence, VLM is around $435\times$ faster on the largest network, `Orkut`. VLM consistently provides a reliable solution in a relatively short time, competitive to the fastest method for that network. On average, PLMR is 1.9x faster than VLM. PLMR might be the fastest in some datasets, however this advantage comes at a cost: PLMR does not prevent stale modularity information and cannot guarantee convergence. While VLM does not always surpass PLMR in modularity and runtime, its provable convergence guarantees make it a more reliable choice for applications requiring deterministic behavior. Future work could bridge this performance gap while preserving VLM's convergence properties.

TABLE IV: Average runtime (in seconds) over multiple runs (best in bold). All methods except Louvain, which is sequential, are run with 40 threads. D2CC timed out after 24 hours for `LiveJournal` and `Orkut`, which we denote as ">24h".

Networks	Louv.	Grap.	Grap++	PLMR	D2CC	VLM
`Facebook`	0.735	0.022	0.026	**0.007**	0.229	0.027
`Enron`	4.126	0.182	0.067	**0.046**	5.662	0.110
`DBLP`	62.78	0.422	0.409	0.411	62.67	**0.388**
`Youtube`	151.8	4.699	8.232	**1.633**	21,831	3.024
`Skitter`	316.8	7.995	2.202	**2.433**	12,389	3.052
`LiveJournal`	2,779	49.85	7.248	10.83	>24h	**10.73**
`Orkut`	12,394	98.32	24.76	**13.84**	>24h	28.46

TABLE V: Total number of iterations for each method. All except Louvain are run with 40 threads. D2CC timed out after 24 hours for `LiveJournal` and `Orkut`.

Networks	Louv.	Grap.	Grap++	PLMR	D2CC	VLM
`Facebook`	7	29	19	28	9	52
`Enron`	7	46	34	53	27	153
`DBLP`	10	51	170	93	55	171
`Youtube`	10	74	116	101	79	401
`Skitter`	8	67	52	91	47	145
`LiveJournal`	58	130	134	111	>24h	441
`Orkut`	53	213	155	97	>24h	361

TABLE VI: Total number of levels for each method. All except Louvain are run with 40 threads. D2CC timed out after 24 hours for `LiveJournal` and `Orkut`.

Networks	Louv.	Grap.	Grap++	PLMR	D2CC	VLM
`Facebook`	3	4	4	4	3	4
`Enron`	3	6	6	5	5	5
`DBLP`	4	8	6	6	6	6
`Youtube`	4	8	8	8	8	8
`Skitter`	3	7	6	6	6	6
`LiveJournal`	5	6	6	6	>24h	6
`Orkut`	4	5	4	4	>24h	5

Total number of iterations (Table V) and levels (Table VI) by each method show consistent trends. Sequential Louvain takes fewer iterations than the parallel methods, which is expected due to conflicts in state-of-the-art parallel methods. While D2CC takes similar number of iterations to other parallel methods, VLM takes significantly more, 2.83x more iterations than PLMR. This is simply due to our deferral mechanism when a community cannot be locked. However, there is no direct correlation between the number of iterations and the computation time: on `LiveJournal`, 441 iterations by VLM takes less time than 111 iterations by PLMR, while whereas on `Orkut` 361 iterations by VLM takes two times more than PLMR which takes 97 iterations. Our deferrals in VLM do not cause any increase in the number of levels (Table VI). Our methods take similar number of levels to the other parallel methods, which is more than the sequential Louvain.

TABLE VII: Strong scaling runtimes on PaRMAT (40M nodes, 400M edges) (in seconds).

# threads	Grap++	PLMR	VLM
1	495.80	1,907.97	627.18
2	588.11	1,826.95	748.62
4	409.45	1,227.44	584.01
8	625.09	779.15	407.99
16	248.24	433.63	312.32
32	222.36	284.97	305.15
40	248.54	247.91	287.14

C. Scalability

We perform strong scalability evaluation of our community versioning approach, VLM, as well as PLMR and Grap++ using a synthetic network. Khorasani et al. [38] introduced PaRMAT, a publicly available software that creates custom graphs in parallel. We use PaRMAT to generate a custom synthetic network containing 40 million nodes and 400 million edges. Table VII presents the strong scaling runtime results. Grap++ reaches 1.99x speedup w.r.t. sequential when run with 40 threads. PLMR demonstrates better scalability, reaching 7.69x speedup, mainly because it has a slower sequential runtime. VLM performs similarly to Grap++, reaching 2.18x on 40 threads. Runtime performance does not improve much after 16 threads for Grap++ and VLM. Grap++ shows an irregular pattern (plateau up to 8 threads, drop between 8 and 16, then another plateau), and we believe this behavior is due to NUMA effects of our architecture on the memory-bound computation—it is likely that threads are better spread across two CPUs starting at 16 threads. Regarding the absolute runtime at 40 threads, PLMR is the best and the VLM is within 15% of PLMR's performance. There is definitely room for better parallelization at larger scales for VLM.

VI. Conclusion and Future Work

In this work, we addressed the lack of theoretical convergence guarantees for parallel Louvain algorithms by introducing two novel parallelization schemes: distance-2 community coloring and community versioning. Our methods ensure conflict-free updates and eliminate instances of stale modularity information, thereby providing convergence guarantees while maintaining competitive performance. Experimental results on real-world networks demonstrate that our community versioning approach achieves significant speedups over sequential Louvain and outperforms some existing parallel implementations, like Grappolo [21], in both runtime and modularity. Although there is room for improvement, especially in terms of weak scaling, our versioning approach is able to process a synthetic graph with 40 million nodes and 400 million edges in under five minutes with 40 threads, making it suitable for large-scale network analysis.

As future work, we will continue experimenting with efficient and scalable strategies to help bridge this performance gap. Another research avenue is to apply our improvements to Leiden algorithm [17], which is shown to give communities with better quality by guaranteeing connectedness via refinement process. The local-move phase is the same in Louvain and Leiden algorithms, so we expect similar improvements in parallel Leiden computation. We will investigate adapting our local-move strategies for alternative parallel Leiden implementations, such as that of [39].

VII. Acknowledgments

Niu and Sarıyüce were supported by NSF-2107089 and NSF-2236789 awards, and used resources of the CCR at the University at Buffalo [36].

References

[1] P. Bedi and C. Sharma, "Community detection in social networks," *Wiley interdisciplinary reviews: Data mining and knowledge discovery*, vol. 6, no. 3, pp. 115–135, 2016.

[2] A. W. Rives and T. Galitski, "Modular organization of cellular networks," *Proceedings of the national Academy of sciences*, vol. 100, no. 3, pp. 1128–1133, 2003.

[3] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3-5, pp. 75–174, 2010.

[4] S. Fortunato and D. Hric, "Community detection in networks: A user guide," *Physics reports*, vol. 659, pp. 1–44, 2016.

[5] H. Gmati, A. Mouakher, A. Gonzalez-Pardo, and D. Camacho, "A new algorithm for communities detection in social networks with node attributes," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–13, 2024.

[6] V. Blondel, J.-L. Guillaume, and R. Lambiotte, "Fast unfolding of communities in large networks: 15 years later," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2024, no. 10, p. 10R001, 2024.

[7] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[8] S. Shirazi, A. Albadvi, E. Akhondzadeh, F. Farzadfar, and B. Teimourpour, "A new application of community detection for identifying the real specialty of physicians," *International journal of medical informatics*, vol. 140, p. 104161, 2020.

[9] N. J. Van Eck and L. Waltman, "Citation-based clustering of publications using citnetexplorer and vosviewer," *Scientometrics*, vol. 111, no. 2, pp. 1053–1070, 2017.

[10] T.-K. Nguyen, M. Coustaty, and J.-L. Guillaume, "A combination of histogram of oriented gradients and color features to cooperate with louvain method based image segmentation." in *VISIGRAPP (4: VISAPP)*, 2019, pp. 280–291.

[11] M. Gregnanin, J. De Smedt, G. Gnecco, and M. Parton, "Signature-based community detection for time series," in *International Conference on Complex Networks and Their Applications*. Springer, 2023, pp. 146–158.

[12] K. Sliwa, E. Kusen, and M. Strembeck, "A case study comparing twitter communities detected by the louvain and leiden algorithms during the 2022 war in ukraine," in *Companion Proceedings of the ACM Web Conference 2024*, ser. WWW '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1376–1381.

[13] W. Zhang, "Improving commuting zones using the louvain community detection algorithm," *Economics Letters*, vol. 219, p. 110827, 2022.

[14] S. J. Brooks, V. O. Jones, H. Wang, C. Deng, S. G. Golding, J. Lim, J. Gao, P. Daoutidis, and C. Stamoulis, "Community detection in the human connectome: Method types, differences and their impact on inference," *Human Brain Mapping*, vol. 45, no. 5, p. e26669, 2024.

[15] S. J. Wilson, A. D WILKINS, C.-H. Lin, R. C. Lua, and O. Lichtarge, "Discovery of functional and disease pathways by community detection in protein-protein interaction networks," in *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, vol. 22, 2016, p. 336.

[16] H. Alves, P. Brito, and P. Campos, "Community detection in interval-weighted networks," *Data Mining and Knowledge Discovery*, vol. 38, no. 2, pp. 653–698, 2024.

[17] V. A. Traag, L. Waltman, and N. J. Van Eck, "From louvain to leiden: guaranteeing well-connected communities," *Scientific reports*, vol. 9, no. 1, pp. 1–12, 2019.

[18] Z. Tang, M. Carrel, C. Koylu, and A. Kitchen, "How human ecology landscapes shape the circulation of h5n1 avian influenza: A case study in indonesia," *One Health*, vol. 16, p. 100537, 2023.

[19] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, D. Metropolitansky, R. O. Ness, and J. Larson, "From local to global: A graph rag approach to query-focused summarization," *arXiv preprint arXiv:2404.16130*, 2024.

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

[21] M. Halappanavar, H. Lu, A. Kalyanaraman, and A. Tumeo, "Scalable static and dynamic community detection using grappolo," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–6.

[22] C. L. Staudt and H. Meyerhenke, "Engineering parallel algorithms for community detection in massive networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 171–184, 2015.

[23] J. J. Tithi, A. Stasiak, S. Aananthakrishnan, and F. Petrini, "Prune the unnecessary: Parallel pull-push louvain algorithms with automatic edge pruning," in *Proceedings of the 49th International Conference on Parallel Processing*, 2020, pp. 1–11.

[24] N. S. Sattar and S. Arifuzzaman, "Parallelizing louvain algorithm: Distributed memory challenges," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2018, pp. 695–701.

[25] X. Que, F. Checconi, F. Petrini, and J. A. Gunnels, "Scalable community detection with the louvain algorithm," in *2015 IEEE international parallel and distributed processing symposium*. IEEE, 2015, pp. 28–37.

[26] R. Forster, "Louvain community detection with parallel heuristics on gpus," in *2016 IEEE 20th Jubilee International Conference on Intelligent Engineering Systems (INES)*. IEEE, 2016, pp. 227–232.

[27] M. Mohammadi, M. Fazlali, and M. Hosseinzadeh, "Accelerating louvain community detection algorithm on graphic processing unit," *The Journal of Supercomputing*, vol. 77, pp. 6056–6077, 2021.

[28] Y.-L. Lin and S. S. Skiena, "Algorithms for square roots of graphs," *SIAM Journal on Discrete Mathematics*, vol. 8, no. 1, pp. 99–118, 1995.

[29] D. Bozdağ, Ü. Çatalyürek, A. H. Gebremedhin, F. Manne, E. G. Boman, and F. Özgüner, "Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation," *SIAM J. Sci. Comput.*, vol. 32, no. 4, p. 2418–2446, Aug. 2010.

[30] P. A. Bernstein, V. Hadzilacos, N. Goodman *et al.*, *Concurrency control and recovery in database systems*. Addison-wesley Reading, 1987, vol. 370.

[31] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, pp. 1–20, 2016.

[32] J. Leskovec and J. Mcauley, "Learning to discover social circles in ego networks," *Advances in neural information processing systems*, vol. 25, 2012.

[33] B. Klimt and Y. Yang, "Introducing the enron corpus." in *CEAS*, vol. 45, 2004, pp. 92–96.

[34] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD workshop on mining data semantics*, 2012, pp. 1–8.

[35] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, pp. 177–187.

[36] Center for Computational Research, University at Buffalo, "Ccr," http://hdl.handle.net/10477/79221, 2026, accessed: 2026-02-19.

[37] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[38] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '15, 2015, pp. 39–50.

[39] S. Sahu, K. Kothapalli, and D. S. Banerjee, "Fast leiden algorithm for community detection in shared memory setting," in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 11–20.

Artifact Description (AD)

VIII. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 Two parallel Louvain algorithms for the local-move phase with provable convergence guarantees: Distance-2 Community Coloring (D2CC) and Community Versioning via two-phase locking (VLM).
C_2 Theoretical convergence proofs showing that both D2CC and VLM are serializable to a sequential Louvain computation, filling a key gap left by all prior parallel Louvain methods.
C_3 Empirical evaluation on seven real-world networks demonstrating that VLM matches the speed and modularity quality of state-of-the-art parallel methods (Grappolo, PLMR) while providing formal convergence guarantees, and achieves up to 435× speedup over sequential Louvain.

B. Computational Artifacts

A_1 https://github.com/erdemUB/IPDPS26

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3	Tables II-VII, Algorithms 2-3

IX. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

A_1 contains the full OpenMP-based C++ implementation of the D2CC and VLM algorithms, built on top of the Grappolo codebase. It directly realizes contributions C_1 (the two parallel algorithms), supports verification of C_2 (convergence guarantees, observable via consistent modularity results across runs), and enables reproduction of all experimental results in C_3.

Expected Results

Running A_1 on the seven SNAP real-world networks should reproduce the modularity scores (Table II), community counts (Table III), runtimes (Table IV), iteration counts (Table V), and level counts (Table VI) reported in the paper. Note that Louvain (and hence all the evaluated heuristics that are built on it) is a randomized algorithm and characteristics of the resulting communities can demonstrate variety, yielding certain amount of variance. VLM should achieve competitive modularity with Grappolo and PLMR while exhibiting provably consistent results across runs. D2CC should time out (>24h) on LiveJournal and Orkut. On the PaRMAT synthetic graph (40M nodes, 400M edges), VLM should complete in under 5 minutes using 40 threads, reproducing Table VII. These results substantiate C_1 and C_3 directly and corroborate C_2 by demonstrating stable, reproducible modularity outcomes

Expected Reproduction Time (in Minutes)

Artifact Setup: ∼30–60 minutes (downloading datasets, compiling code).

Artifact Execution: Sequential Louvain computation would take ∼8 hours for the seven real-world networks. Parallel runs of Grappolo, Grappolo++, PLMR, D2CC, and VLM would take ∼4 hours for the seven real-world networks using 40 threads (excluding D2CC on LiveJournal and Orkut, which exceed 24h). It will take ∼8 hours to reproduce the full strong-scaling results—∼30 minutes if only reproducing the 40-thread row. Note that reported times in tables do not include I/O time. Overall, ∼20 hours needed for a single run; we did four runs in the paper.

Artifact Analysis: ∼15–30 minutes (collecting outputs and comparing to reported tables).

Artifact Setup (incl. Inputs)

Hardware: A shared-memory multicore machine is required. Experiments were conducted on a dual-socket server with 2× Intel Xeon Gold CPUs at 2.1 GHz (20 cores each, 40 cores total) and 256 GB DDR4 RAM. A machine with at least 8 cores and 32 GB RAM is recommended for partial reproduction. NUMA-aware architectures are preferable for accurate scalability results.

Software:

- GCC 10.2.0 or later (https://gcc.gnu.org), compiled with -O3
- OpenMP 4.5 (included with GCC; https://www.openmp.org)
- PaRMAT graph generator for synthetic scalability experiments (https://github.com/farkhor/PaRMAT)
- Sequential Louvain (https://github.com/jlguillaume/louvain)
- Grappolo (https://github.com/ECP-ExaGraph/grappolo)
- PLMR (https://networkit.github.io/dev-docs/notebooks/Community.html); it is the PLM variant with refine set to true
- The artifact itself: https://github.com/erdemUB/IPDPS26. It includes the codes for D2CC, VLM, and Grap++, which is our custom implementation of improved Grappolo, we called Grappolo++ or Grap++ in short, that uses the incremental gain computation and pull/push improvement.

Datasets / Inputs: All seven real-world networks (Facebook, Enron, DBLP, Youtube, Skitter, LiveJournal, Orkut) are available for download from the SNAP repository at https://snap.stanford.edu/data. The synthetic scalability graph (40M nodes, 400M edges) is generated using PaRMAT; instructions for generation are available in the PaRMAT repository.

Installation and Deployment: Clone the repository at https://github.com/erdemUB/IPDPS26. Follow the README instructions to compile using GCC 10.2.0+ with OpenMP support and the -O3 flag. Download and format the SNAP datasets as required by the build system. For the PaRMAT graph, compile and run PaRMAT with the appropriate node/edge parameters before executing the scalability experiments.

Artifact Execution

The experiment workflow consists of three tasks:

T_1 (Data Preparation): Download the seven SNAP datasets and convert them to the required input format. Optionally, generate the PaRMAT synthetic graph (40M nodes, 400M edges) using the PaRMAT tool.

T_2 (Algorithm Execution): ($T_1 \rightarrow T_2$)Run each algorithm (sequential Louvain, Grappolo, Grap++, PLMR, D2CC, VLM) on each input graph. For real-world networks, use 40 threads. For the strong-scaling experiment (Table VII), run Grap++, PLMR, and VLM on the PaRMAT graph using 1, 2, 4, 8, 16, 32, and 40 threads. Each configuration is run four times and results are averaged. PLMR is capped at 32 iterations per level.

T_3 (Result Collection and Analysis): ($T_2 \rightarrow T_3$) Collect modularity scores, community counts, runtimes, iteration counts, and level counts from T_2 outputs and tabulate them to reproduce Tables II–VII.

Key experimental parameters: 40 threads for all parallel runs on real-world networks; 4 repeated runs per configuration for averaging; PLMR maximum 32 iterations per level; VLM processes all nodes in the first two iterations before switching to the neighbor-pruning strategy.

Artifact Analysis (incl. Outputs)

The primary outputs are: average modularity scores (Table II), total community counts (Table III), average runtimes in seconds (Table IV), total iteration counts (Table V), total level counts (Table VI), and strong-scaling runtimes in seconds (Table VII). Results should be compared directly against the reported tables. Minor numerical variation across machines is expected due to differences in hardware and thread scheduling; however, VLM's convergence guarantee means modularity results should be stable and reproducible across runs on the same machine. D2CC results exceeding 24h on LiveJournal and Orkut are expected and confirm the practical limitations discussed in Section V.