Memory Efficient GPU-based Label Propagation Algorithm (LPA) for Large-scale Community Detection on Shared-memory systems

First Last
first.last@email.org
Institution
City, State, Country

First Last
first.last@email.org
Institution
City, State, Country

ABSTRACT

Community detection involves grouping nodes in a graph with dense connections within groups, than between them. Recently, efficient multicore (GVE-LPA) and GPU-based (ν-LPA) implementations of Label Propagation Algorithm (LPA) for community detection have been proposed. However, these methods incur high memory overhead due to their per-thread/per-vertex hashtables. This makes it challenging to process large graphs on shared memory systems. In this paper, we introduce memory-efficient GPU-based LPA implementations, using weighted Boyer-Moore (BM) and Misra-Gries (MG) sketches. Our new implementation, νMG8-LPA, using an 8-slot MG sketch, reduces memory usage by 98× and 44× compared to GVE-LPA and ν-LPA, respectively. It is also 2.4× faster than GVE-LPA and only 1.1× slower than ν-LPA, with minimal quality loss (4.7%/2.9% drop compared to GVE-LPA/ν-LPA).

KEYWORDS

Community detection, LPA, Memory efficient, GPU-based

1 INTRODUCTION

Research on graph-structured data has surged due to graphs' ability to model complex, real-world interactions and relationships between entities. A key area in this field is community detection, which involves identifying clusters of vertices with stronger internal connections than those to the rest of the network [24]. These clusters, known as communities, are "intrinsic" when based solely on the network's structure and are "disjoint" if each vertex belongs to only one group [18, 28]. Community detection helps understand a network's structure and behavior [1, 24], and has a wide range of applications across various fields, including healthcare [6, 31, 68], biological network analysis [39, 59, 62], machine learning [5, 21], urban planning [15, 82, 84], cloud computing [14], social network analysis [10, 38, 43, 80], ecology [29], drug discovery [50, 79], and other graph-related problems [12, 27, 34, 51, 71, 75, 81].

Community detection is challenging because the number and size of communities are unknown in advance [28]. Consequently, heuristic methods are often used [11, 17, 20, 40, 60, 61, 64, 77, 78]. The modularity metric is commonly used to assess the quality of the detected communities [55]. One widely used heuristic is the Label Propagation Algorithm (LPA), also known as RAK [60], a diffusion-based method known for its simplicity, speed, and scalability. Compared to the Louvain method [11], another leading algorithm known for generating high-quality communities, LPA is significantly faster, but tends to identify communities with lower quality [67]. This makes LPA particularly useful for large-scale networks where speed is more critical, and slight reduction in community quality is acceptable. While LPA typically scores lower in modularity, it has been observed to perform well in terms of Normalized Mutual Information (NMI) against ground truth [58].

Community detection is a well-studied problem, with many efforts focused on improving algorithm performance through optimizations [25, 26, 30, 48, 54, 56, 65, 76, 78, 83] and parallelization techniques [8, 9, 16, 26, 30, 54, 70, 72, 74, 77, 88]. The primary focus of these studies has been on reducing computation time, while memory consumption has often been a secondary concern. Yet, as network sizes continue to grow, managing memory usage is becoming increasingly crucial, especially for processing large-scale graphs in shared-memory settings. A proposed multicore implementation of LPA, GVE-LPA [66], offers state-of-the-art performance on shared-memory systems, but it also incurs significant memory overhead due to the reliance on per-thread hashtables. Recently, the same author introduced a GPU-based implementation of LPA, ν-LPA [67], which employs a novel open-addressing per-vertex hashtable with hybrid quadratic-double probing for efficient collision resolution. However, the memory demand for hashtables scales with $O(|E|)$, where $|E|$ is the number of edges — which can be substantial. For a graph with 4 billion edges, ν-LPA necessitates approximately 64 GB of GPU memory for the hashtables alone. On a 3.8 billion-edge graph, the combined memory requirement of ν-LPA — including the graph storage — surpasses the 80 GB device memory limit of an NVIDIA A100 GPU. These limitations motivated us to explore ways to lower memory usage in community detection algorithms, even if it means sacrificing some performance.

This paper presents νBM-LPA and νMG8-LPA[1], our memory-efficient GPU-based implementations of LPA. νBM-LPA is based on weighted Boyer-Moore majority vote, while νMG8-LPA relies on the weighted Misra-Gries heavy hitters method with $k = 8$ slots. Key optimizations in νMG8-LPA include: **(1)** Storing sketches in shared memory; **(2)** Using cooperative thread groups (CGs) to reduce thread divergence; **(3)** Applying warp-level vote functions for faster sketch updates; **(4)** Employing multiple sketches for high-degree vertices, which are later merged, to reduce contention; and **(5)** Avoiding rescans of the top-k labels without compromising community detection quality. Both implementations incorporate a Pick-Less (PL) strategy for symmetry breaking to prevent repeated label swaps. In the paper, we also discuss techniques to identify free slots in the sketches, handle high-degree vertices where multiple threads may access the same slot, selection of optimal k, partitioning thresholds for high-degree vertices, and CG configurations for workload balancing. These optimizations allow our algorithms to achieve good performance and quality of identified communities, at a significantly smaller working set size.

[1] Source code to be made available online after review

2 RELATED WORK

Label Propagation Algorithm (LPA) is widely used in various fields, such as cross-lingual knowledge transfer for part-of-speech tagging [19], 3D point cloud classification [87], sectionalizing power systems [3], finding connected components [75], graph compression [12], link prediction [53, 89], and graph partitioning [4, 51, 71]. Significant work has also been conducted to improve the original LPA through various modifications [7, 22, 23, 44, 63, 69, 91–94].

Some open-source tools for LPA-based community detection include the Fast Label Propagation Algorithm (FLPA) [77], which speeds up LPA by only processing vertices with recently updated neighbors. NetworKit [73], a large-scale graph analysis package with a Python interface, features a parallel LPA implementation that tracks active nodes and uses guided parallel processing.

Sahu [66] has proposed a high-performance multicore implementation of LPA, GVE-LPA. It uses collision-free per-thread hashtables — each hash table has a key list, a values array (sized to the number of vertices, $|V|$), and a key count. Values are stored or accumulated at indices corresponding to their keys. To prevent cache conflicts, the key count is updated independently and allocated separately in memory. This approach substantially reduces conditional branching and minimizes the number of instructions needed for inserting or accumulating entries. GVE-LPA achieves performance improvements of 139× over FLPA and 40× over NetworKit LPA.

Although GVE-LPA is computationally efficient, it comes with a significant memory overhead. Its space complexity, not counting the input graph, is $O(T|V|)$, where $|V|$ is the number of vertices and T is the number of threads used. For example, processing a graph with 200 million vertices using 64 threads requires between 102 and 205 GB of memory just for the hashtables.

Some GPU-based implementations of LPA have been introduced. Soman and Narang [72] proposed a parallel GPU algorithm for weighted LPA, while Kozawa et al. [42] developed a GPU-accelerated LPA that can handle datasets too large for GPU memory. More recently, Ye et al. [90] introduced GLP, a GPU-based LPA framework. However, despite the utility of LPA, there was a lack of efficient and widely available GPU-based implementation of LPA. To address this, Sahu [67] proposed ν-LPA. It employs asynchronous execution, a pick-less strategy to reduce community swaps, and a novel per-vertex hashtable with hybrid quadratic-double probing for collision resolution. Running on an NVIDIA A100 GPU, it outperformed FLPA, NetworKit LPA, and cuGraph Louvain by 364×, 62×, and 37×, respectively. However, as mentioned earlier, the memory demand of ν-LPA's hashtables scale with $O(|E|)$, which is significant.

We now review studies in the edge streaming setting, where graphs are processed as sequences of edges in a single pass. These algorithms aim to minimize runtime and memory usage for efficient graph processing. Hollocou et al. [35, 36] introduced SCoDA, which tracks only a few integers per node by noting that edges are more likely to connect nodes within the same community. Wang et al. [85] focused on finding local communities around query nodes by sampling neighborhoods and using an approximate conductance metric. Liakos et al. [45, 46] explored expanding seed-node sets as edges arrive without storing the full graph. Although these methods are efficient, the single-pass limit may reduce community quality compared to multi-pass algorithms.

3 PRELIMINARIES

Consider an undirected graph $G(V, E, w)$, where V is the set of vertices, E is the set of edges, and w_{ij} is the weight of the edge between vertices i and j (with $w_{ij} = w_{ji}$). For an unweighted graph, each edge has a unit weight ($w_{ij} = 1$). The neighbors of vertex i are $J_i = \{j \mid (i, j) \in E\}$, and the weighted degree of vertex i is $K_i = \sum_{j \in J_i} w_{ij}$. The graph has $N = |V|$ vertices, $M = |E|$ edges, and the total sum of edge weights is $m = \frac{1}{2} \sum_{i,j \in V} w_{ij}$.

3.1 Community detection

Disjoint community detection seeks to assign each vertex $i \in V$ to a community c from a set Γ, via a community membership function $C : V \to \Gamma$. The set of vertices in community c is denoted as V_c, and the community to which vertex i belongs is denoted as C_i. For a given vertex i, its neighbors in community c are represented as $J_{i \to c} = \{j \mid j \in J_i \text{ and } C_j = c\}$, and the sum of the edge weights between i and its neighbors in c is $K_{i \to c} = \sum_{j \in J_{i \to c}} w_{ij}$. The total weight of edges within community c is denoted by $\sigma_c = \sum_{(i,j) \in E \text{ and } C_i = C_j = c} w_{ij}$, while the total edge weight associated with community c is given by $\Sigma_c = \sum_{(i,j) \in E \text{ and } C_i = c} w_{ij}$.

3.2 Modularity

Modularity is a measure of the quality of communities identified. It calculates the difference between the actual fraction of edges within communities and the expected fraction if edges were randomly assigned, with values ranging from $[-0.5, 1]$. Higher values indicate stronger community structure. The modularity Q is computed using Equation 1 and involves the Kronecker delta function ($\delta(x, y)$), which equals 1 when $x = y$ and 0 otherwise. Additionally, the *delta modularity* for moving vertex i from community d to community c is denoted as $\Delta Q_{i:d \to c}$, calculated with Equation 2.

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

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

3.3 Label Propagation Algorithm (LPA)

LPA [60] is a fast, scalable diffusion-based method for detecting moderate-quality communities in large networks, outperforming Louvain [11] in terms of speed. Initially, each vertex has a unique label (community ID). In each iteration, vertices update their labels by adopting the one with the highest total connecting weight, as described in Equation 3. This process continues until a consensus is reached, forming communities. The algorithm stops when at least $1 - \tau$ of the vertices (where τ is a tolerance parameter) keep their

labels unchanged. LPA has a time complexity of $O(L|E|)$ and space complexity of $O(|V|+|E|)$, where L is the number of iterations [60].

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

3.4 Boyer-Moore (BM) majority vote algorithm

The Boyer-Moore majority vote algorithm efficiently identifies the majority element in a sequence, which is an element that appears more than $n/2$ times in a list of n elements. Developed by Boyer and Moore [13] in 1981, the algorithm tracks a candidate and a vote count. Initially, the candidate is set, and the count starts at zero. As the list is traversed, if the count is zero, the current element becomes the new candidate with a count of one. If the current element matches the candidate, the count increases; if it differs, the count decreases. At the end, the candidate holds the potential majority element. It runs in $O(n)$ time and uses $O(1)$ space.

3.5 Misra-Gries (MG) heavy hitters algorithm

The Misra-Gries (MG) heavy hitters algorithm, introduced in 1982 by Misra and Gries [52], extends the Boyer-Moore majority algorithm to identify elements that occur more than $\frac{n}{k+1}$ times, where n is the total number of elements and $k+1$ is a user-defined threshold. The algorithm uses up to k counters to track candidate elements and their counts. If a candidate appears, its counter is incremented; if not, a new counter is created if space allows. When all counters are full, each counter is decremented, and elements with zero counts are removed. After processing, the remaining candidates are likely to exceed the $\frac{n}{k+1}$ threshold, though a verification step is needed. The algorithm runs in $O(n)$ time and uses $O(k)$ space, making it efficient for limited-resource environments.

3.6 Fundamentals of a GPU

The core unit of NVIDIA GPUs is the Streaming Multiprocessor (SM), which contains multiple CUDA cores for parallel processing. Each SM also has shared memory, registers, and specialized function units. The number of SMs varies by GPU model, and each operates independently. The memory hierarchy includes global memory (largest but slowest), shared memory (low-latency, used by threads within an SM), and local memory (private storage for threads when registers are full). Threads on a GPU are organized into warps (32 threads executing together), thread blocks (groups of threads on the same SM), and grids (collections of thread blocks). Warps execute in lockstep, and SMs schedule warps alternately if threads stall. Threads within a block communicate via shared memory, while blocks in a grid exchange data through global memory.

3.7 Warp-level primitives

NVIDIA GPUs offer warp-level primitives that allow operations to be executed in parallel across all threads in a warp, enabling efficient communication. A key function is the *warp vote* operation, which performs logical operations across all threads in a warp. These include `__all_sync()`, `__any_sync()`, and `__ballot_sync()`. `__all_sync()` checks if all threads satisfy a condition, `__any_sync()` checks if at least one does, and `__ballot_sync()` collects each thread's boolean result in a 32-bit integer.

4 APPROACH

Sahu [67] recently introduced ν-LPA, a GPU-optimized version of LPA based on GVE-LPA [66]. It uses per-vertex open-addressing hashtables, as shown in Figure 1, where the size of the hashtables is proportional to each vertex's degree. ν-LPA does this because allocating large fixed-size hashtables per thread, similar to GVE-LPA, is infeasible on GPUs, which support massive parallelism but have a limited memory. However, as brought up earlier, ν-LPA still requires significant memory, with a space complexity of $O(|E|)$, where $|E|$ is the number of edges. For instance, processing a graph with 4 billion edges requires 64 GB of GPU memory for the hashtables alone. Memory demand can escalate rapidly, as graphs grow. This highlights the need to explore ways of reducing the memory footprint of the hashtables, even at the cost of some performance.

Note that in every iteration of LPA, each vertex $i \in V$ iterates over its neighbors J_i, excluding itself, and calculates the total edge weight $K_{i \to c}$ for each unique label $c \in \Gamma_i$ among its neighbors. These weights are stored in a hashtable. The label c^* with the highest weight $K_{i \to c^*}$ is then selected as the new label for vertex i.

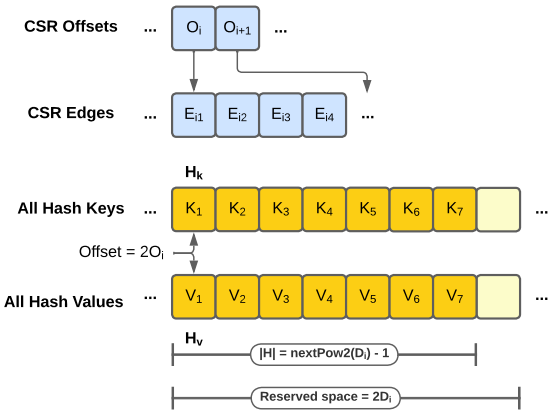


Figure 1: Illustration of per-vertex open-addressing hashtables in ν-LPA [67]. Each vertex i has a hashtable H with a key array H_k and a value array H_v. Memory for all hash key and value arrays is allocated together. The offset for vertex i's hashtable is $2O_i$, where O_i is its CSR offset. The total memory for the hashtable is $2D_i$, where D_i is the vertex's degree. The hashtable's capacity is $nextPow2(D_i) - 1$.

To reduce the memory usage of LPA, we focus on a "sketch" of neighboring community labels rather than storing a fully populated map. Instead of keeping all labels $c \in \Gamma_i$ for each vertex i and their associated linking weights $K_{i \to c}$, we only track labels with a linking weight greater than $\frac{K_i}{k+1}$, where K_i is the weighted degree of i, and k is a user-defined parameter. The intuition is that the label with the highest linking weight, c^*, will likely be among the k most significant labels. To achieve this, we use a weighted version of the Misra-Gries (MG) heavy-hitter algorithm [52] with k slots. Instead of counting occurrences of neighboring community labels, we accumulate the edge weights between vertex i and its neighbors, grouped by community label. We then identify up to k candidate labels. Not all of these labels will necessarily have a linking weight

above $\frac{K_i}{k+1}$, so some entries may correspond to non-majority labels or remain empty if there are fewer than k labels. In a second scan, we may calculate the total linking weight between i and the candidate labels, and select the label $c^{\#}$ with the highest weight. While $c^{\#}$ may differ from the highest weight label c^{*}, the two are expected to align in most cases when k is appropriately chosen .

Furthermore, we investigate reducing the memory usage of our GPU implementation of LPA by employing a weighted variant of the Boyer-Moore (BM) majority vote algorithm [13]. This approach represents a minimal case of the weighted MG algorithm with $k = 1$, where only a single majority candidate label is tracked.

We now present the design of our GPU-based implementation of LPA, called *v*MG-LPA, which uses k slots to maintain a sketch or summary of neighboring community labels for each vertex, leveraging a weighted version of the Misra-Gries (MG) heavy hitters algorithm. Building on this, *v*BM-LPA — our GPU-based LPA implementation based on the weighted Boyer-Moore (BM) majority vote algorithm — is described later, in Section 4.7.

4.1 Design of MG Sketch

We utilize k slots in our weighted MG sketch S, which is composed of two distinct arrays: S_k and S_v. The S_k array holds the candidate community labels for the current vertex being processed, while the S_v array stores the corresponding sketch weights. Each slot in the sketch is identified by an index s, where $s < k$. A slot s is deemed empty if its associated weight is zero, meaning $S_v[s] = 0$.

We now explain how a key-value pair (c, w) is accumulated into the MG sketch S. The process begins by checking whether the community label c already exists as a candidate label in the sketch, i.e., if $S_k[s] = c$ for some s. If c is found, the associated weight of the slot is incremented by the edge weight w, updating $S_v[s] \leftarrow S_v[s] + w$. If c does not exist as a candidate label, an attempt is made to populate a free slot s_ϕ (where $S_v[s_\phi] = 0$) with the key-value pair by setting $S_k[s_\phi] = c$ and $S_v[s_\phi] = w$. If no free slot is available (i.e., all slots are occupied, with $S_v[s] \neq 0$ for all s), the associated weights of all slots in the sketch are decremented by the edge weight w, applying $S_v[s] \leftarrow S_v[s] - w$ for all s. This decrement ensures that less frequent labels are gradually removed, freeing up space for labels that may become more frequent.

To efficiently accumulate key-value pairs into the MG sketch using threads, we avoid assigning the task of updating/managing an MG sketch to a single thread, as this would not leverage the parallelism of GPUs and would introduce significant warp divergence. Instead, we assign at least one thread group g to manage each MG sketch, where each slot s in the sketch is exclusively managed a unique thread t within the thread group, in parallel. Cooperative groups [33] are employed to partition the threads in a thread block (see Section 3.6 for more details) into smaller thread groups. This partitioning is essential because cooperative groups allow for the creation of thread groups smaller than a warp (32 threads), which would otherwise be restricted to MG sketches with at least 32 slots. We use `tiled_partition()` to partition threads in a thread block, with the size of each cooperative group / thread group, being fixed at compile time to k. Additionally, given that each MG sketch is relatively small but subjected to many updates, we store the MG sketch for each thread group in shared memory, which acts as a user-managed cache, providing significantly higher memory bandwidth than global memory (which is external).

When accumulating a key-value pair (c, w) into the MG sketch, two communication points between threads in the thread group are required: one to check if a community label c already exists as a candidate label in the sketch, and another to find a free slot s_ϕ to store (c, w) if c is not already present. This intra-group communication can be handled using shared memory variables. We would like to note that, with a suitable choice of representative values, it is possible to get away with a single shared memory variable, which we refer to as *has*. In particular, we initialize *has* to -1 to indicate that c does not exist as a candidate label, and set to 0 if a matching candidate label is found. If no candidate label exists (i.e., $has = -1$), each thread t in the thread group executes an atomic max operation on *has* to determine the last free slot in the sketch, which can then be populated with (c, w). If no free slot is found, *has* remains set to -1. Afterward, all threads decrement their respective slots by w. Note that we must synchronize all threads in the group, before accessing *has*, to ensure proper conditional branching.

An MG sketch may however be shared between multiple thread groups, i.e., each slot in the sketch may no longer be updated by a single thread exclusively but rather operated upon by multiple threads (one per thread group). For such shared sketches, atomic operations must be employed when updating them. Specifically, atomic addition (`atomicAdd()`) is required to increment the associated weight $S_v[s]$ of a matching candidate label at slot s by edge weight w when accumulating a key-value pair (c, w), and to decrement the weights of all slots in the sketch by w when no free slots exist. In addition, atomic compare-and-swap (`atomicCAS()`) must be used when populating a free slot s_ϕ with (c, w). Since multiple threads may attempt to populate the same free slot, the `atomicCAS()` operation can fail, necessitating a retry loop to find another available free slot. Furthermore, shared variables like *has*, which are used for communication between threads, should be updated atomically, although atomicity is not always required in certain cases, such as with a boolean-like variable that only changes in one direction. This applies to the shared variable *has*, when it is used to identify whether a matching candidate label exists in the sketch for the key-value pair being accumulated.

We now determine a suitable value for k, the number of slots in each MG sketch, for our GPU implementation of LPA. A larger value of k is expected to improve community quality, as it increased the likelihood of identifying the most weighted label for a given vertex. However, increasing k also requires a larger number of threads per thread group, which reduces the number of vertices processed per unit time. Additionally, a higher k leads to increased communication and synchronization costs between threads, as well as lower occupancy of Symmetric Multiprocessors (SMs). We experiment with k values ranging from 2 to 32 (in powers of 2) on large, real-world graphs (see Table 1), ensuring each graph is undirected and weighted, with edge weights set to 1. Figure 2 illustrates the relative runtime and modularity of communities for varying values of k. The results show that using MG sketches with $k = 8$ slots runs 2.2× faster than those with $k = 32$, while the community quality only decreases by 1.6%, striking a balance between runtime and community quality. Therefore, we select $k = 8$ slots for each sketch.

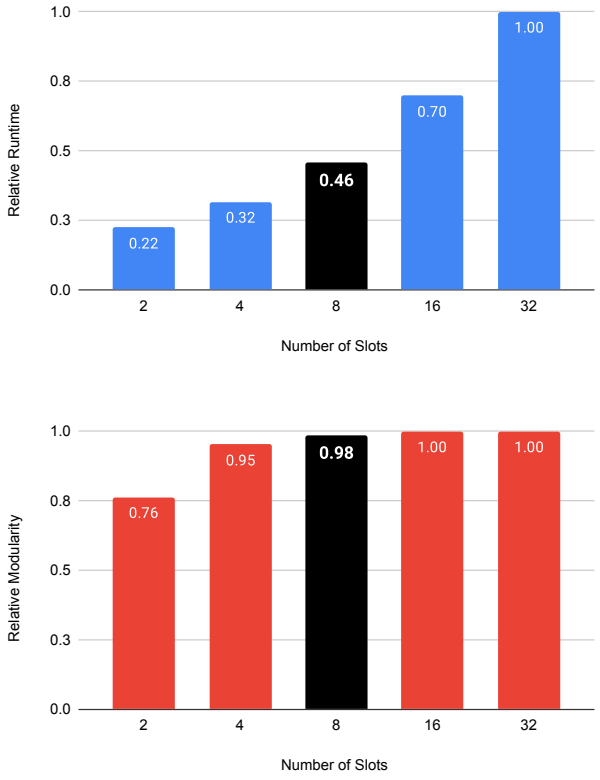


Figure 2: Relative runtime and Modularity of obtained communities using *v*MG-LPA, with varying number of slots k in the Misra-Gries (MG) sketch, ranging from 2 to 32.

Since CUDA 9, NVIDIA has introduced warp-level primitives [47], which enable threads within a warp to directly exchange data, perform collective operations, and coordinate their execution without relying on shared memory or synchronization primitives like barriers. These include warp-level vote functions, such as `__all_sync()` and `__ballot_sync()`, which, as detailed in Section 3.7, allow threads in a warp to check whether all threads in the warp (or a subset, depending on selected thread flags) satisfy a condition, or to collect each thread's boolean result in a 32-bit integer. Additionally, cooperative groups [33] provide a simpler API for these warp-level primitives. To leverage this, we replace the use of the shared memory variable *has* as follows: Given a key-value pair (c, w) to accumulate into the sketch S, each thread t in the thread group g (managing the sketch) checks if its respective slot s in the sketch has c as the candidate label, i.e., $S_k[s] = c$. A call to the `g.ballot()` function is used to check if any thread in g successfully found c as the candidate label in their respective slots. A similar procedure is followed to find a free slot, where threads collaborate using the ballot function, and `__ffs()` (which finds the first set bit) is used to determine the first available slot in the sketch. We also use `g.all()` to check if all threads in g failed to populate a free slot, in which case, the weights of all slots in the sketch are decremented by the edge weight w. Note that using warp-level primitives limits the number of slots k in our MG sketch to 32, the size of a warp. However, this is not problematic, as we have already identified $k = 8$ slots to be suitable for our algorithm.

Based on the above discussion, we analyze two variations of our GPU-based implementation of LPA: the *Shared variables* approach and the *Warp-vote* approach. In the *Shared variables* approach, each neighbor of a vertex, along with its associated edge weight, is accumulated into the MG sketch using shared memory variables for intra-group communication. In contrast, the *Warp-vote* approach uses warp-level voting functions, instead of shared memory, to support thread cooperation in populating the sketch. We conduct experiments on large graphs (shown in Table 1), ensuring that each graph is undirected and weighted, as earlier. Figure 3 illustrates the relative runtime of the *Shared variables* and *Warp-vote* approaches for populating/accumulating MG sketches. As the figure shows, with the *Warp-vote* approach is 1.2× faster than *Shared variables* approach. Both approaches result in communities with the same modularity, and hence, is not shown in the figure. Accordingly, we use the *Warp-vote* approach to populate the sketch.

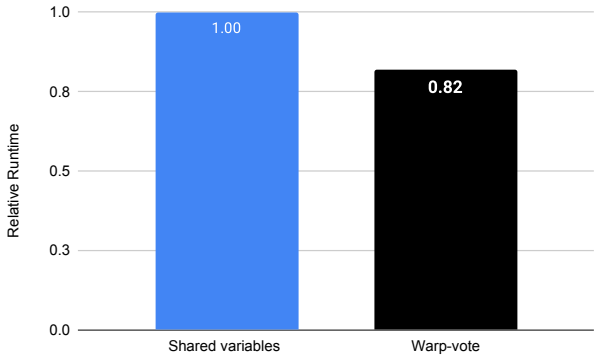


Figure 3: Relative Runtime of *Shared variables* and *Warp-vote* approaches for populating weighted Misra-Gries (MG) sketches from the neighborhood of each vertex.

4.2 Organization of Thread Groups

We now discuss how we organize thread groups to update the labels of vertices in the input graph. A key step in this process is obtaining an MG sketch of each vertex's neighborhood, which helps identify the candidate label, $c^{\#}$, with the highest linking weight. This label is then selected as the updated label for the vertex. A simple approach to achieve this would be to assign a thread group to each vertex, where the group is responsible for managing/updating the sketch. However, many real-world graphs follow a power-law degree distribution, where a small set of vertices have high degrees while the majority have low degrees. If high-degree vertices are processed by a single thread group, it would likely lead to significant load imbalance. To address this, a better approach is to assign multiple thread groups to high-degree vertices, ideally in proportion to their degree. This improves parallelism but can introduce increased contention and repeated retries for occupying free slots in the sketch.

However, to keep things simple, and avoid the overhead of multiple kernel calls (each corresponding to a specific number of thread groups per vertex), we instead partition the vertices into two sets: high-degree and low-degree vertices — based on a degree threshold D_H, where vertices with degree $\geq D_H$ are classified as high-degree. We then process low-degree vertices using a group-per-vertex kernel, where each vertex is handled by a single thread group; and high-degree vertices using a block-per-vertex kernel, where each vertex is processed by one thread block, consisting of R_H thread groups. For high-degree vertices processed by the block-per-vertex kernel, the MG sketch is "shared", i.e., each slot in the sketch is updated by multiple threads (one in each thread group), necessitating the use of atomic operations and retry loops when updating, as detailed in Section 4.1. In contrast, for low-degree vertices processed by the group-per-vertex kernel, the MG sketch is not shared — eliminating the need for atomic operations or retry loops.

We now discuss a few additional operations needed to update the label for each vertex. In order to find the candidate label $c^{\#}$ with the most linking weight for each vertex $i \in V$, once the MG sketch S has been populated, we do a second scan to calculate the total linking weight between i and the candidate labels, and then perform a pairwise max block-reduce on the sketch labels array S_k and weights array S_v. If $c^{\#}$ differs from the current label of the vertex, $C[i]$, we update $C[i]$ to $c^{\#}$ and, in parallel, mark the neighbors of i as unprocessed. Additionally, one thread in a thread group/thread block tracks the number of label updates observed, denoted as ΔN_G, which is then atomically added to ΔN, a global variable that counts the number of changed vertices in the current iteration. To ensure correctness and avoid race conditions when threads within a block share data or depend on each other for computations, appropriate synchronization barriers are employed throughout the kernel.

To optimize the parameters of our algorithm, which include the degree threshold D_H for high-degree vertices (processed by the block-per-vertex kernel), the number of thread groups R_H used per vertex in the block-per-vertex kernel, and the kernel launch configurations for both the group- and block-per-vertex kernels, we performed manual gradient descent. This involved iteratively adjusting each parameter slightly and observing its impact on runtime, with random cycling through parameters until further optimization was no longer achievable. After numerous adjustments, we determined optimal values as follows: a degree threshold D_H of 128, a per-vertex thread group count R_H of 32 for the block-per-vertex kernel, and kernel launch configurations of 32 threads per block for the group-per-vertex kernel and 256 threads per block for the block-per-vertex kernel. Recognizing the potential for settling in a local minimum, we aim to explore auto-tuning of the kernels in the future, which is especially important for AMD GPUs [49].

4.3 Consolidation of Sketches

As discussed earlier, we assign multiple thread groups to high-degree vertices in the block-per-vertex kernel to process multiple edges and update the MG sketch in parallel. However, shared sketches can experience significant contention, especially when the number of cooperating thread groups, R_H, is large — such as the value $R_H = 32$ in our case. Moreover, with a large R_H, operations like finding a free slot may frequently fail as threads compete to fill any remaining free slots in the sketch, leading to increased retries and potential performance degradation due to warp divergence, as the size of each thread group ($k = 8$) is smaller than the warp size.

However, Misra-Gries (MG) sketches, also known as MG summaries, are mergeable [2]. Therefore, to address the above issue, we consider the use of separate sketches $S[g]$ for each thread group g processing a vertex (covering a subset of its neighbors). We also refer to these separate sketches as *partial* sketches, as they represent the sketch of a subset of neighbors of each vertex. Once all edges of the vertex are processed, and all the partial sketches are populated, they are merged — in the block-per-vertex kernel. Using separate sketches per thread group implies that we no longer need atomic operations and retry loops when operating on such sketches. However, this does involve additional work of merging the independent sketches, and could lower the occupancy of SMs due to the increased shared memory needed per thread block.

To merge the independent partial sketches $S[g]$ from each thread group g into a single consolidated sketch, all thread groups except the first ($g \neq 0$) work in parallel to merge their private sketches $S[g]$ into the sketch of the first thread group, $S[0]$. Specifically, each thread group g iteratively accumulates non-empty slots, which contain candidate labels and their associated weights, from its own sketch $S[g]$ into $S[0]$ until all its slots are processed. During this process, each thread within g is assigned to operate on a slot in $S[0]$ in a shared manner using atomic operations and retry loops.

The merging step discussed above introduces some contention, but since the work involved is minimal, we believe the cost is negligible. To confirm this, we conduct an experiment comparing the performance of two approaches: the *Shared sketch* approach, which uses warp-level voting functions to populate a single shared sketch (in the block-per-vertex kernel), and the *Partial sketches* approach, which also utilizes warp-level voting functions but employs separate sketches for each thread group processing a vertex, where each group populates its own sketch based on the neighbors and associated edge weights it observes, and later merges these into a consolidated sketch in parallel. It is important to note that the group-per-vertex kernel is identical for both approaches. The experiment was conducted on the graphs from Table 1, ensuring that each graph was undirected and weighted, with a weight of 1 for each edge. Figure 4 presents the mean relative runtime of the *Shared sketch* and *Partial sketches* approaches — showing that the *Partial sketches* approach is 1.7× faster than the *Shared sketch* approach. Since both approaches yield communities with the same modularity, this is not shown in the figure. As a result, we opt to use the *Partial sketches* approach for the block-per-vertex kernel.

4.4 A Single Scan is Sufficient

Note that the k candidate labels we obtain for a vertex i in an MG sketch will include labels with a linking weight greater than $\frac{K_i}{k+1}$, where K_i is the weighted degree of i. However, not all of these labels will necessarily exceed this threshold, i.e., some entries may correspond to non-majority labels, or remain empty, if there are fewer than k labels. A second scan is then performed over the neighbors of vertex i to compute the total linking weight between i and the candidate labels, selecting the label $c^{\#}$ with the highest linking weight. This second scan involves first clearing the associated

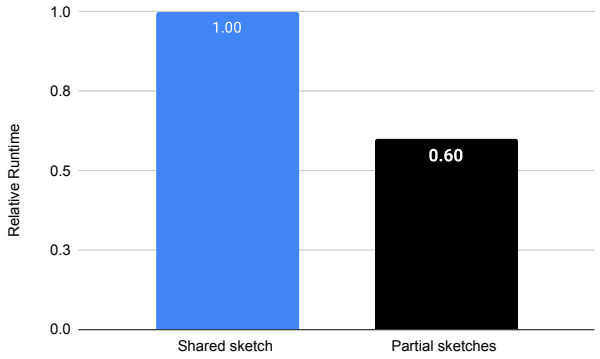


Figure 4: Relative Runtime of *Shared sketch* and *Partial sketches* approaches for populating weighted Misra-Gries (MG) sketches from the neighborhood of each vertex.

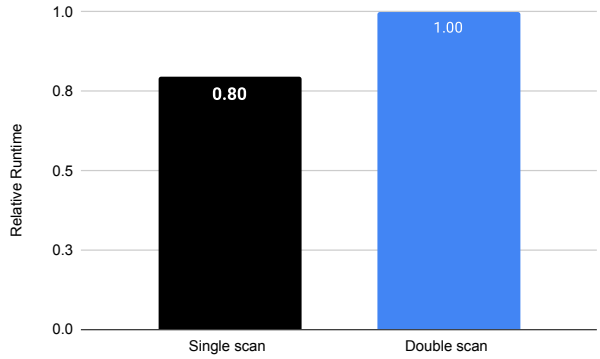


Figure 5: Relative Runtime of *Single scan* vs. *Double scan* approaches for selecting the updated label of each vertex.

weights in the consolidated sketch and then accumulating the total linking weights for each candidate label by adding the edge weight w of each neighbor with label c into the corresponding slot in the sketch. This accumulation process adds additional computational cost. Furthermore, in the block-per-vertex approach, R_H thread groups process R_H edges of the vertex in parallel within the shared sketch, leading to contention between thread groups.

However, it is likely that the most weighted candidate label in the MG sketch, which we refer to as $c^{@}$, will align with the label $c^{\#}$ that has the highest linking weight after a second scan. This eliminates the need to calculate the total linking weight $K_{i\to c}$ between the current vertex i and each of its k-majority communities. To test this theory, we conduct an experiment comparing the performance of two approaches: the *Single scan* approach, where we select $c^{@}$, the most weighted candidate label in the MG sketch, as the new label for the vertex, and the *Double scan* approach, where we perform a second scan on the vertex's edges to calculate the total linking weight between i and the candidate labels in the sketch, and then select the label $c^{\#}$ with the highest linking weight. The experiment is conducted on the graphs from Table 1, ensuring that each graph is undirected and weighted. Figure 5 illustrates the mean relative runtime of the *Single scan* and *Double scan* approaches. As the results show, the *Single scan* approach is 1.3× faster than the *Double scan* approach. Both approaches yield nearly identical modularity values for the resulting communities, which is why modularity is not shown in the figure. Based on these results, we adopt the *Single scan* approach to select the new label for each vertex.

4.5 Mitigating Community Swaps

Note that GPU-based LPA can fail to converge due to vertices getting stuck in cycles of community label swaps. This can happen when two interconnected vertices keep adopting each other's labels, especially in symmetrical situations where vertices are equally connected to each other's communities. Such swaps are more likely because GPUs execute in lockstep, and symmetrical vertices may end up repeatedly swapping labels, preventing convergence. Therefore, symmetry-breaking techniques are essential.

In a previous work [67], the Pick-Less (PL) approach was introduced to address this issue, where a vertex can only switch to a lower community ID, preventing community swaps. However, using PL too frequently can reduce the algorithm's ability to identify high-quality communities. We found that applying PL every $\rho = 4$ iterations, starting from the first iteration, results in the highest modularity communities. However, after further testing, we found that a ρ value of 8 is slightly more effective (in fact, it seems that applying PL in the first iteration resolves most community swap problems). However, conservatively, we use a value of $\rho = 8$.

4.6 Our Memory Efficient GPU-based LPA employing Misra-Gries (MG) Sketch

The optimizations discussed above significantly reduce the memory usage of our weighted Misra-Gries (MG) based GPU implementation of LPA, νMG-LPA, while maintaining competitive performance in terms of runtime and community quality (modularity), when compared to ν-LPA [67]. A high-level overview of νMG-LPA is shown in Figure 6, which demonstrates how νMG-LPA selects the best candidate community label for each vertex, comparing the group-per-vertex kernel in Figure 6(a) and the block-per-vertex kernel in Figure 6(b). In both cases, each MG sketch contains $k = 4$ slots, and in the block-per-vertex kernel, each vertex is processed by 3 thread groups (as an example). In the group-per-vertex kernel, a single thread group is assigned to each vertex, which populates the sketch in parallel using k threads, and the vertex's new label is the one with the highest weighted sketch value. In the block-per-vertex kernel, multiple thread groups (in this example, 3) are assigned to each vertex. Each thread group populates its own private sketch based on a subset of neighbors, with each group using k threads, and then the separate sketches are merged into a single consolidated sketch in parallel. The vertex's new label is then chosen based on the highest weighted candidate label in this merged sketch.

Since the MG sketches of νMG-LPA are of fixed size and reside on the shared memory of the GPU, the space complexity of νMG-LPA is $O(|V|)$, excluding the input graph — in contrast to ν-LPA's space complexity of $O(|E|)$. Both algorithms have the same time complexity of $O(K|E|)$, where K represents the number of LPA

iterations performed. The pseudocodes for νMG-LPA and for populating the MG sketches is given Algorithms 1, 2. We also refer to our algorithm as νMG8-LPA, as we use $k = 8$ slots for the sketches.

4.7 Our Memory Efficient GPU-based LPA employing Boyer-Moore (BM) Algorithm

We now discuss the design of νBM-LPA, our GPU-based implementation of LPA based on a weighted version of the Boyer-Moore (BM) majority vote algorithm. The algorithm processes a key-value pair (c, w) by first checking if the community label c matches the current majority weighted label $c^{\#}$. If $c^{\#} = c$, the associated majority weight $w^{\#}$ is incremented by w. If $c^{\#} \neq c$, it checks whether $w^{\#} > w$; if so, $w^{\#}$ is decremented by w; otherwise, both $c^{\#}$ and $w^{\#}$ are updated to c and w, respectively. For load balancing, as in νMG-LPA, vertices in the input graph are partitioned into low- and high-degree sets. However, unlike νMG-LPA, low-degree vertices are processed with a thread-per-vertex kernel, as the update can be handled by a single thread. High-degree vertices, on the other hand, are processed using a block-per-vertex kernel, where a thread block subdivides processing of the vertex's edges among multiple threads, with each thread maintaining its own $c^{\#}$ and $w^{\#}$ based on the subset of edges it observes. After all edges are processed, the threads collaborate in a pair max block-reduce operation to determine the majority $c^{\#}$ and $w^{\#}$ across all threads. As with νMG-LPA, νBM-LPA we apply a manual gradient descent to optimize the parameters. However, we arrive at the same parameter values as νMG-LPA, i.e., a degree threshold D_H of 128, and kernel launch configurations of 32 threads per block for the thread-per-vertex kernel and 256 threads per block for the block-per-vertex kernel. Additionally, νBM-LPA mitigates community swaps in the same way as νMG-LPA.

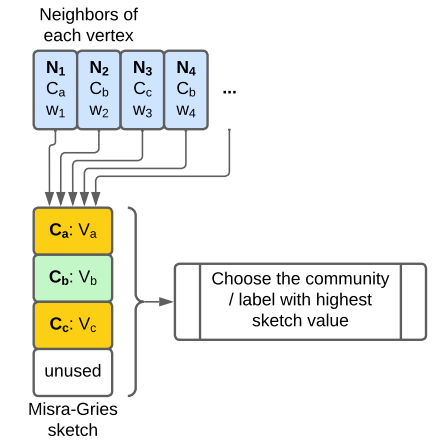
Like νMG-LPA, νBM-LPA also has a space complexity of $O(|V|)$ and time complexity of $O(K|E|)$. The psuedocode of νBM-LPA, along with its detailed explanation is given in our appendix.

4.8 Our Weighted Misra-Gries (MG) based GPU Implementation of LPA

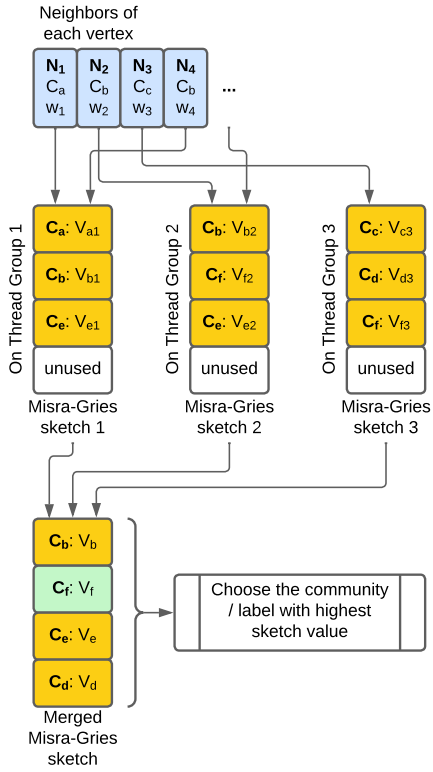
Algorithm 1 outlines the pseudocode for our GPU-based implementation of LPA using the weighted Misra-Gries (MG) heavy hitters algorithm, which we call νMG-LPA. Here, the `lpa()` function takes a graph G as input and outputs the labels C for each vertex.

In `lpa()`, we start by assigning each vertex a unique community label, setting $C[i]$ to the vertex ID (line 2). We then run LPA iterations up to a maximum of *MAX_ITERATIONS* (line 3). Every ρ iterations, we activate PL mode (line 5) to reduce ineffective label swaps. Next, in each iteration, we invoke `lpaMove()`, which updates the community labels based on local neighborhood information (line 6) and returns ΔN, the number of vertices with changed labels. If the fraction of changed vertices $\Delta N/N$ falls below the specified tolerance τ and PL mode is inactive, the algorithm has converged, and thus terminates (line 7). Otherwise, the process repeats until convergence. Finally, the community labels C are returned (line 8).

Each iteration of the LPA is handled by the `lpaMove()` function (line 9). In this function, the community label of each unprocessed vertex i in the graph G is updated. To do this, each vertex i is assigned one or more thread groups based on its degree. A thread



(a) Group-per-vertex kernel of νMG-LPA



(b) Block-per-vertex kernel of νMG-LPA

Figure 6: Illustration of how νMG-LPA selects the best candidate community label for each vertex, with the group-per-vertex kernel shown in (a), and the block-per-vertex kernel shown in (b). Here, the number of slots in each MG sketch is assumed to be $k = 4$, and in the block-per-vertex approach, the number of thread groups processing the vertex is assumed to be 3. In the figure, N_* represents the neighbors of a vertex, C_* represents the community labels of those neighbors, and w_* represents the edge weights associated with each neighbor. Additionally, each slot in the MG sketch is associated with labels/keys C_* and values/weights V_*.

Algorithm 1 *v*MG-LPA: Our GPU-based implementation of LPA, based on weighted Misra-Gries (MG) heavy hitters algorithm.

▷ $G(V, E)$: Input graph
▷ C: Community label of each vertex
□ N: Number of vertices in G, i.e., $|V|$
□ $S(S_k, S_v)$: Labels, weights array of the MG sketch
□ $c^@$: Most weighted candidate label in the sketch
□ ΔN: Number of changed vertices, overall
□ ΔN_G: Changed vertices per thread group/block
□ R_H: Number of thread groups processing a vertex
□ k: Number of slots in the MG sketch
□ s: Slot index for the current thread
□ g: Current thread group ID
□ t: Current thread ID
□ ρ: Iteration gap for pick-less mode
□ τ: Iteration tolerance

1: **function** LPA(G)
2: $C \leftarrow [0..|V|)$
3: **for all** $l_i \in [0 \ \ldots$ MAX_ITERATIONS$)$ **do**
4: ▷ Mitigate community swaps with **pick-less**
5: **if** $l_i \bmod \rho = 0$ **then** employ **pick-less** mode
6: $\Delta N \leftarrow lpaMove(G, C)$
7: **if not pick-less and** $\Delta N/N < \tau$ **then break**
8: **return** C

9: **function** LPAMOVE(G, C)
10: $S_k \leftarrow \{\{\}\}$; $S_v \leftarrow \{\{\}\}$
11: $\Delta N \leftarrow 0$; $\Delta N_G \leftarrow \{0\}$
12: $s \leftarrow t \bmod k$; $g \leftarrow \lfloor t/k \rfloor$ **on each thread**
13: **for all unprocessed** $i \in V$ **in parallel do**
14: ▷ Scan communities connected to vertex i
15: $sketchClear(S[g])$ **in parallel**
16: **for all** $(j, w) \in G.neighbors(i)$ **in parallel do**
17: **if** $j = i$ **then continue**
18: $sketchAccumulate(S[g], C[j], w, s)$ **in parallel**
19: ▷ Merge multiple sketches into one
20: **if** $R_H > 2$ **then** use **shared** mode below
21: **if** $g > 0$ **then in parallel**
22: **for all** $s \in [0 \ \ldots \ k)$ **do**
23: $c \leftarrow S_k[g, s]$; $w \leftarrow S_v[g, s]$
24: **if** $w = 0$ **then continue**
25: $sketchAccumulate(S[0], c, w, s)$ **in parallel**
26: ▷ Find best community label for vertex i
27: $c^@ \leftarrow sketchMaxKey(S[0])$ **in parallel**
28: ▷ Change label of vertex i to most weighted label $c^@$
29: **if** $c^@ \neq C[i]$ **and** (**not pick-less or** $c^@ < C[i]$) **then**
30: $C[i] \leftarrow c^@$; $\Delta N_G[g] \leftarrow \Delta N_G[g] + 1$
31: **for all** $j \in G.neighbors(i)$ **in parallel do**
32: Mark j as unprocessed
33: $atomicAdd(\Delta N, \Delta N_G[g])$ **in parallel**
34: **return** ΔN

group contains exactly k threads, with each thread being responsible for a specific slot in the MG sketch. Each thread's slot index — the slot it is responsible for — is calculated as $s = t \bmod k$, where t is the thread ID, and k is both the number of slots in the sketch and the number of threads in the group. Additionally, each thread group has a unique ID $g = \lfloor t/k \rfloor$. At the start of `lpaMove()`, we initialize the MG sketch arrays for labels S_k and weights S_v, in addition to the overall count ΔN of changed vertices, and the counts ΔN_G of changed vertices for each thread group (lines 10-12). Each vertex i in G is then processed in parallel (line 13), starting with a scan of its neighboring communities to determine the top-k weighted labels. For this, each thread group g clears its private sketch $S[g]$ and then accumulates labels from the vertex's neighbors $j \in J_i$ based on edge weights $w = w_{ij}$ using the `sketchAccumulate()` function (lines 16-18). The psuedocode of `sketchAccumulate()` is given in Algorithm 2. After the neighborhood scan, N_V/k sketches, one from each thread group, have now been populated. Here, N_V is the total number of threads per vertex, and N_V/k is the number of thread groups assigned to that vertex. We now proceed to merge these sketches into a single, consolidated sketch in $S[0]$. For this, all thread groups except the first thread group assigned to each vertex ($g > 0$), start to accumulate their top-k identified labels into the sketch $S[0]$ belonging to the first thread group ($g = 0$) of each vertex in parallel (lines 20-25). If more than two thread groups handle a vertex ($N_V/k > 2$), merging is done in "shared" mode, using appropriate atomic operations to manage shared updates. An alternative approach is to use a single shared sketch for each vertex i, accessible by all thread groups. This requires atomic operations due to concurrent access, which can lead to increased contention. Despite avoiding the overhead of merging multiple sketches, this shared approach has shown lower performance, as shown in Section 4.3. Therefore, we employ the multi-sketch merging approach.

After merging the MG sketches from each thread group into a single consolidated sketch for vertex i, we identify the most weighted candidate label $c^@$ in the sketch (line 27). We do not perform a rescan to find the label with the highest weight for i because it does not improve performance or community quality, as discussed in Section 4.4. Next, we check if $c^@$ differs from i's current label and if it meets the conditions set by the PL mode (e.g., $c^@$ is smaller than $C[i]$ if PL mode is active). If it does, we update i's label to $c^@$, adjust the count of changed vertices for the current thread group ΔN_G (noting that only the first thread group updates this count when multiple groups manage the same vertex), and mark all neighboring vertices of i as unprocessed to allow label updates to propagate (lines 29-32). After all vertices are processed, the count of changed vertices from each thread group ΔN_G is summed into a global count ΔN using atomic addition (line 33). Finally, we return ΔN (line 34), allowing `lpa()` to determine convergence.

4.9 Populating Misra-Gries (MG) sketch

Algorithm 2 presents the pseudocode for accumulating a label and its associated weight into a weighted Misra-Gries (MG) sketch, using warp-level primitives [47]. Here, the `sketchAccumulate()` function takes as input a sketch S represented by label S_k and weight S_v arrays, a label c, a weight w to be accumulated, and a slot index s in the sketch, that is specific to the current thread.

Algorithm 2 Accumulating a label, and its associated weight, in a weighted Misra-Gries (MG) sketch — using warp-level primitives.

▷ $S(S_k, S_v)$: Labels, weights array of the MG sketch
▷ c, w: Label, weight to accumulate into the MG sketch
▷ s: Slot index for the current thread
□ *has*: MG sketch has label c?
□ s_ϕ: Free slot index

1: **function** SKETCHACCUMULATE(S, c, w, s)
2: ▷ Add edge weight to community label
3: **if** $S_k[s] = c$ **then**
4: **if not shared then** $S_v[s] \leftarrow S_v[s] + w$
5: **else** $atomicAdd(S_v[s], w)$
6: $has \leftarrow groupBallot(S_k[s], c)$
7: ▷ Done if label is already in the sketch
8: **if** $has \neq 0$ **then return** *done*
9: ▷ Find and empty slot, and populate it
10: ▷ Retry if some other thread reserved the free slot
11: **repeat**
12: ▷ Find empty slot
13: $B_\phi \leftarrow groupBallot(S_v[s] = 0)$
14: $s_\phi \leftarrow findFirstSetBit(B_\phi) - 1$
15: **if** $B_\phi = 0$ **then break**
16: ▷ Add community label to sketch
17: **if** $s_\phi = s$ **then**
18: **if not shared then**
19: $S_k[s] \leftarrow c$
20: $S_v[s] \leftarrow w$
21: **else**
22: **if** $atomicCAS(S_v[s], 0, w) = 0$ **then** $S_k[s] \leftarrow c$
23: **else** $B_\phi \leftarrow 0$
24: ▷ B_ϕ may have been updated
25: **if is shared then** $B_\phi \leftarrow groupAll(B_\phi \neq 0)$
26: **until not shared or** $B_\phi \neq 0$
27: ▷ Subtract edge weight from non-matching labels
28: **if** $B_\phi = 0$ **then**
29: **if not shared then** $S_v[s] \leftarrow S_v[s] - w$
30: **else** $atomicAdd(S_v[s], -w)$
31: **return** *done*

In the algorithm, we start by checking whether the current slot s already holds the target label c. If it does, the weight w is added to the current weight stored in $S_v[s]$ (lines 3–5). In "shared" mode, where the sketch is shared among multiple thread groups, this addition is performed atomically. If the label c is already present, the `groupBallot()` function is used to broadcast c's presence across threads within the warp, updating the bits in *has* accordingly. If $has \neq 0$, indicating the label is found, no further action is needed since the sketch already contains the label (line 8). If the label is not found, we proceed to locate an empty slot in the sketch to store c and w. This search is performed iteratively until a free slot is successfully reserved. Here, we use the `groupBallot()` function to identify free slots by checking for zero-valued weights (lines 13–15), and use the `findFirstSetBit()` function to determine the first available slot. Once a slot is identified, we attempt to populate it (lines 17–23). In *non-shared* mode, where the sketch is exclusive to a single thread group, the slot is directly assigned. In *shared* mode, an atomic compare-and-swap operation ensures that the assignment only occurs if the slot is still free, preventing race conditions. If another thread claims the slot simultaneously, the search process is restarted. If no empty slot is found ($B_\phi = 0$), weight adjustment is performed in order to maintain the MG sketch. In this case, w is subtracted uniformly from the weights of all existing labels (lines 28–30). Finally, the algorithm returns a *done* status (line 31).

5 EVALUATION

5.1 Experimental Setup

5.1.1 System used. We use a server with a 64-core AMD EPYC-7742 processor running at 2.25 GHz, and NVIDIA A100 GPU which has 80 GB of global memory (1935 GB/s bandwidth), 164 KB of shared memory/SM, 108 SMs, and 64 CUDA cores/SM. The server also has 512 GB of DDR4 RAM, and runs Ubuntu 20.04. For CPU-only LPA evaluations, we use a separate server with two 16 core Intel Xeon Gold 6226R processors running at 2.90 GHz. Each core has 1 MB L1 cache, 16 MB L2 cache, and a 22 MB shared L3 cache. This system also has 512 GB of RAM and runs CentOS Stream 8.

5.1.2 Configuration. We use 32-bit integers for vertex IDs, community/label IDs, and sketch keys/labels, and use 32-bit floating-point numbers for edge weights and sketch values. In our GPU implementation of LPA with weighted MG sketches, *v*MG8-LPA, we use 8 slots per sketch, avoid rescanning the top-k community labels, utilize warp-level voting functions, and employ a merge-based kernel (where each thread group creates a sketch for a vertex, which is later merged). For both *v*MG8-LPA and our weighted BM algorithm implementation, *v*BM-LPA, vertices are split into low and high-degree sets: degrees below 128 are low, and the rest are high. For low-degree vertices, *v*BM-LPA uses a thread-per-vertex kernel with a thread block size of 32, while high-degree vertices use a block-per-vertex kernel with a thread block size of 256, where threads collectively find the majority community label using shared memory. For *v*MG8-LPA, low-degree vertices are processed with a group-per-vertex kernel with a thread block size of 32, where the 32 threads are split into 4 cooperative groups of 8 threads each. High-degree vertices use a block-per-vertex kernel with a thread block size of 256, where the 256 threads are divided into 32 cooperative groups of 8 threads each — after scanning, these thread groups merge their local sketches into a single weighted MG sketch after scanning all edges. Both algorithms use a Pick-Less (PL) setting of 8 to reduce frequent label swaps, only allowing label changes to lower-ID labels every 8 iterations — starting from the first iteration. Further, we use an iteration tolerance of $\tau = 0.05$ and cap iterations at $MAX\_ITERATIONS = 20$ [66]. For compilation we use the `-O3` optimization flag, and employ CUDA 11.4 on the GPU system. On the CPU-only system, we rely on GCC 8.5 and OpenMP 4.5. All multicore implementations of LPA are executed with 64 threads.

5.1.3 Dataset. The graphs used in our experiments, shown in Table 1, are from the SuiteSparse Matrix Collection [41]. These graphs range from 3.07 million to 214 million vertices and 25.4 million to 3.80 billion edges. All edges are undirected and weighted, with a

default weight of 1. We did not use publicly available real-world weighted graphs due to their smaller size, although our parallel algorithms can handle weighted graphs without changes. We also exclude SNAP datasets with ground-truth communities, as they are non-disjoint, while our focus is on disjoint communities. It is worth noting that community detection is not just about matching ground truth, which may not accurately reflect a network's real structure and could miss meaningful patterns [57].

Table 1: List of 13 graphs obtained SuiteSparse Matrix Collection [41] (directed graphs are marked with $*$). Here, $|V|$ is the number of vertices, $|E|$ is the number of edges (after adding reverse edges), and D_{avg} is the average degree, and $|\Gamma|$ is the number of communities obtained with *νMG8-LPA*.

Graph	$\|V\|$	$\|E\|$	D_{avg}	$\|\Gamma\|$
Web Graphs (LAW)				
indochina-2004*	7.41M	341M	41.0	385K
uk-2002*	18.5M	567M	16.1	863K
arabic-2005*	22.7M	1.21B	28.2	476K
uk-2005*	39.5M	1.73B	23.7	1.55M
webbase-2001*	118M	1.89B	8.6	12.7M
it-2004*	41.3M	2.19B	27.9	1.50M
sk-2005*	50.6M	3.80B	38.5	633K
Social Networks (SNAP)				
com-LiveJournal	4.00M	69.4M	17.4	175K
com-Orkut	3.07M	234M	76.2	1.91K
Road Networks (DIMACS10)				
asia_osm	12.0M	25.4M	2.1	2.86M
europe_osm	50.9M	108M	2.1	8.04M
Protein k-mer Graphs (GenBank)				
kmer_A2a	171M	361M	2.1	41.5M
kmer_V1r	214M	465M	2.2	50.4M

5.2 Performance Comparison

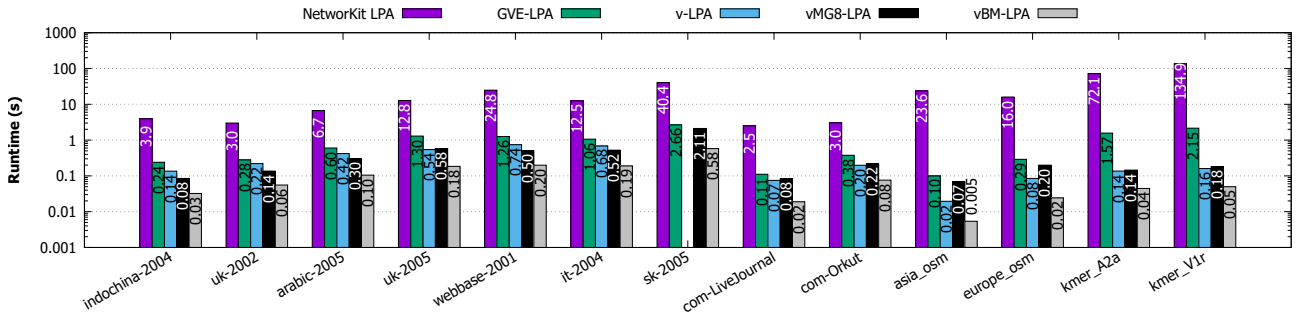
We evaluate the performance of our algorithms, *ν*MG8-LPA and *ν*BM-LPA, in comparison with NetworKit LPA [73], GVE-LPA [66], and *ν*-LPA [67]. NetworKit LPA and GVE-LPA are parallel multicore implementations, while *ν*-LPA is GPU-based. We do not compare with Gunrock's LPA [86] as it produces low-quality communities. Other GPU-based LPA implementations have either unavailable due to access restrictions [72, 90] or suffer from runtime failures on large graphs [42]. cuGraph Louvain [37] is also excluded, as *ν*-LPA is around 37× faster — though it idnetifies communities with 9.6% lower quality, consistent with typical LPA behavior.

For NetworKit LPA, we use a Python script to run `PLP` (Parallel Label Propagation) and measure total runtime with `getTiming()`. To measure memory usage, we monitor the Resident Set Size (RSS) before running `PLP` and the peak memory usage during execution by repeatedly reading `/proc/self/status`. Note that NetworKit LPA might use more memory than reported, as it allocates several small buffers which are likely to have been already reserved by the runtime (from the OS). In contrast, our implementations use large, contiguous memory blocks. For GVE-LPA and *ν*-LPA, we run their respective scripts. We measure GVE-LPA's memory usage by checking the RSS before and after memory allocation. For *ν*-LPA, we use `cudaMemGetInfo()` to measure memory before and after allocation. We exclude memory used to store the input graph, focusing only on memory used by the algorithm itself, including community labels. Neither GVE-LPA nor *ν*-LPA allocate additional memory during iterations, so memory tracking during execution is unnecessary. We perform five runs per graph to calculate average runtimes and modularity of detected communities for each implementation.

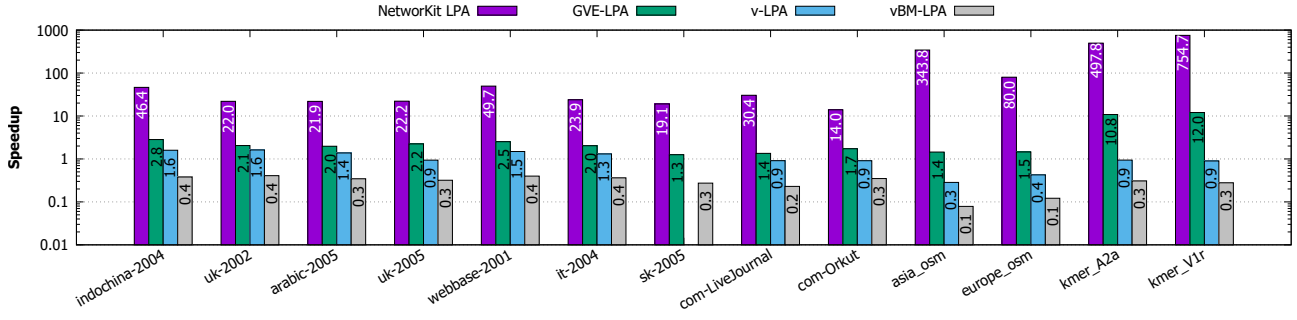
Figure 7(a) compares the runtimes of NetworKit LPA, GVE-LPA, *ν*-LPA, *ν*MG8-LPA, and *ν*BM-LPA across different graphs. Figure 7(b) highlights the speedup of *ν*MG8-LPA relative to other methods. Figure 7(c) displays the modularity scores of the detected communities, while Figure 7(d) shows the memory usage of each method (excluding storage for the input graph). Due to an out-of-memory error, *ν*-LPA results for the *sk-2005* graph are omitted. In terms of memory usage, both *ν*MG8-LPA and *ν*BM-LPA achieve, on average, 2.2×, 98×, and 44× lower memory usage than NetworKit LPA, GVE-LPA, and *ν*-LPA. Note how this allows *ν*MG8-LPA and *ν*BM-LPA to successfully run on the *sk-2005* graph. Further, on average, *ν*BM-LPA is 186×, 9.0×, 3.5×, and 3.7× faster than NetworKit LPA, GVE-LPA, *ν*-LPA, and *ν*MG8-LPA, respectively, but its community quality is 27%, 24%, 23%, and 20% lower than those methods, respectively. In comparison, *ν*MG8-LPA is 51× and 2.4× faster than NetworKit LPA and GVE-LPA, but 1.1× and 3.7× slower than *ν*-LPA and *ν*BM-LPA. It identifies communities that are 8.4%, 4.7%, and 2.9% lower in quality than NetworKit LPA, GVE-LPA, and *ν*-LPA, but 25% higher than *ν*BM-LPA. In particular, we observe that *ν*MG8-LPA identifies communities of high-quality on web graphs and social networks, but yields lower-quality communities on road networks and protein k-mer graphs. In contrast, *ν*BM-LPA obtains communities of moderate-quality on web graphs but performs poorly on the other graph types. We plan to address this discrepancy in future work. Despite this, the current findings indicate that *ν*MG8-LPA is a strong candidate for web graphs and social networks. For road networks, however, GVE-LPA proves to be the most effective, while NetworKit LPA is recommended for protein k-mer graphs. *ν*BM-LPA may be considered for web graphs.

6 CONCLUSION

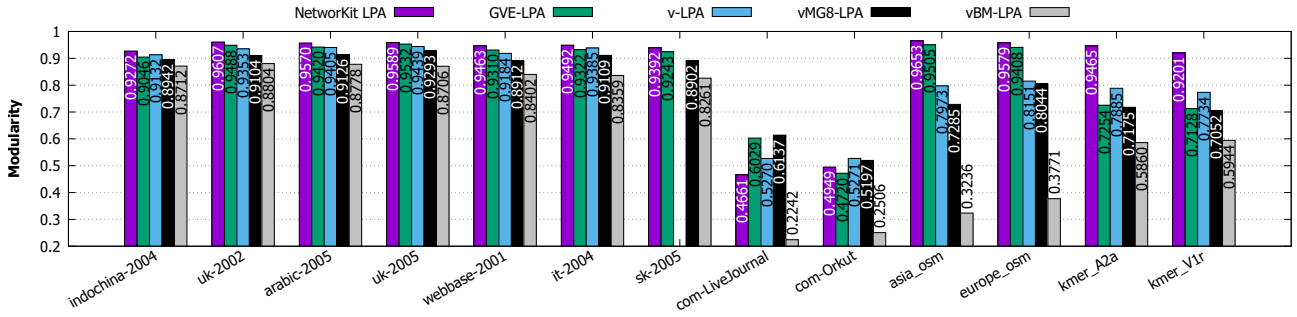
In summary, this paper presents a memory-efficient, GPU-based implementation of LPA for community detection, addressing the high memory demands of previous methods like GVE-LPA and *ν*-LPA. By using weighted Boyer-Moore (BM) and Misra-Gries (MG) sketches, we reduce memory usage without sacrificing performance. The proposed algorithms, *ν*MG8-LPA and *ν*BM-LPA, use 2.2×, 98×, and 44× less memory than NetworKit LPA, GVE-LPA, and *ν*-LPA, respectively. Further, *ν*BM-LPA is 186×, 9.0×, 3.5×, and 3.7× faster than these methods but results in lower community quality (up to 27% less). *ν*MG8-LPA is 51× and 2.4× faster than NetworKit LPA and GVE-LPA, with only a small quality decrease (up to 8.4%) compared to these methods. It performs best on web graphs and social networks, while *ν*BM-LPA is faster on web graphs but less effective on other graph types. When leveraging unified memory [32] to store the input graph, we hope the reduced working set of our algorithms facilitate processing of massive graphs.



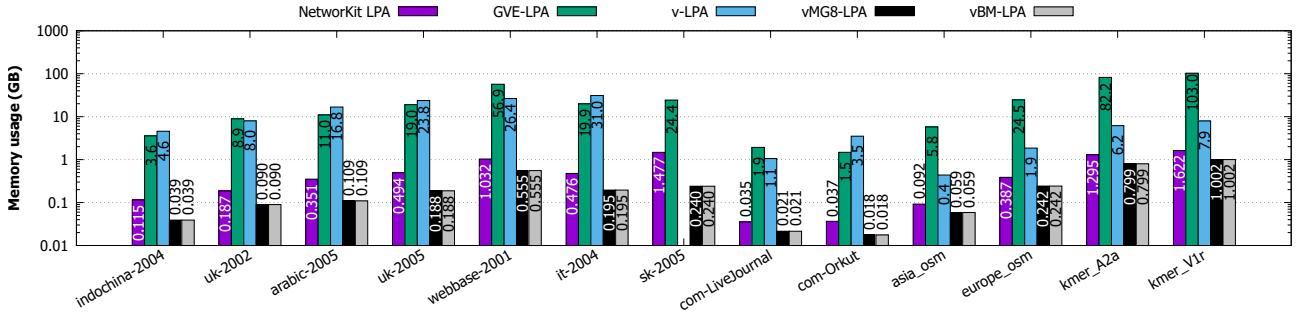
(a) Runtime in seconds (logarithmic scale) with *NetworKit LPA*, *GVE-LPA*, *v-LPA*, *vMG8-LPA*, and *vBM-LPA*



(b) Speedup of *vMG8-LPA* (logarithmic scale) with respect to *NetworKit LPA*, *GVE-LPA*, *v-LPA*, and *vBM-LPA*.



(c) Modularity of communities obtained with *NetworKit LPA*, *GVE-LPA*, *v-LPA*, *vMG8-LPA*, and *vBM-LPA*.



(d) Memory usage in gigabytes of *NetworKit LPA*, *GVE-LPA*, *v-LPA*, *vMG8-LPA*, and *vBM-LPA*.

Figure 7: Runtime in seconds (log-scale), speedup (log-scale), modularity of obtained communities, and memory usage in gigabytes (log-scale) with *NetworKit LPA*, *GVE-LPA*, *v-LPA*, *vMG8-LPA*, and *vBM-LPA* for each graph in the dataset.

REFERENCES

[1] Emmanuel Abbe. 2018. Community detection and stochastic block models: recent developments. *Journal of Machine Learning Research* 18, 177 (2018), 1–86.

[2] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. 2013. Mergeable summaries. *ACM Transactions on Database Systems (TODS)* 38, 4 (2013), 1–28.

[3] Tarique Aziz, Muhammad Waseem, Shengyuan Liu, Zhenzhi Lin, Yuxuan Zhao, and Kaiyuan Pang. 2023. A novel power system sectionalizing strategy based on modified label propagation algorithm. In *2023 6th International Conference on Energy, Electrical and Power Engineering (CEEPE)*. IEEE, 807–812.

[4] Minho Bae, Minjoong Jeong, and Sangyoon Oh. 2020. Label propagation-based parallel graph partitioning for large-scale graph data. *IEEE Access* 8 (2020), 72801–72813.

[5] Yuhe Bai, Camelia Constantin, and Hubert Naacke. 2024. Leiden-Fusion Partitioning Method for Effective Distributed Training of Graph Embeddings. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 366–382.

[6] Joel J Bechtel, William A Kelley, Teresa A Coons, M Gerry Klein, Daniel D Slagel, and Thomas L Petty. 2005. Lung cancer detection in patients with airflow obstruction identified in a primary care outpatient practice. *Chest* 127, 4 (2005), 1140–1145.

[7] Kamal Berahmand and Asgarali Bouyer. 2018. LP-LPA: A link influence-based label propagation algorithm for discovering community structures in networks. *International Journal of Modern Physics B* 32, 06 (2018), 1850062.

[8] A. Bhowmick, S. Vadhiyar, and V. PV. 2022. Scalable multi-node multi-GPU Louvain community detection algorithm for heterogeneous architectures. *Concurrency and Computation: Practice and Experience* 34, 17 (2022), 1–18.

[9] A. Bhowmik and S. Vadhiyar. 2019. HyDetect: A Hybrid CPU-GPU Algorithm for Community Detection. In *IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, Goa, India, 2–11.

[10] Ivan Blekanov, Svetlana S Bodrunova, and Askar Akhmetov. 2021. Detection of hidden communities in twitter discussions of varying volumes. *Future Internet* 13, 11 (2021), 295.

[11] V. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (Oct 2008), P10008.

[12] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. 587–596.

[13] Robert S Boyer and J Strother Moore. 1991. MJRTY—a fast majority vote algorithm. In *Automated reasoning: essays in honor of Woody Bledsoe*. Springer, 105–117.

[14] Lingli Cao and Cheng Zhang. 2022. Implementation of domain-oriented microservices decomposition based on node-attributed network. In *Proceedings of the 2022 11th International Conference on Software and Computer Applications*. 136–142.

[15] Wendong Chen, Xize Liu, Xuewu Chen, Long Cheng, and Jingxu Chen. 2023. Deciphering flow clusters from large-scale free-floating bike sharing journey data: a two-stage flow clustering method. *Transportation* (2023), 1–30.

[16] C. Cheong, H. Huynh, D. Lo, and R. Goh. 2013. Hierarchical Parallel Algorithm for Modularity-Based Community Detection Using GPUs. In *Proceedings of the 19th International Conference on Parallel Processing* (Aachen, Germany) *(Euro-Par'13)*. Springer-Verlag, Berlin, Heidelberg, 775–787.

[17] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. 2004. Finding community structure in very large networks. *Physical review E* 70, 6 (2004), 066111.

[18] Michele Coscia, Fosca Giannotti, and Dino Pedreschi. 2011. A classification for community discovery methods in complex networks. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 4, 5 (2011), 512–546.

[19] Dipanjan Das and Slav Petrov. 2011. Unsupervised part-of-speech tagging with bilingual graph-based projections. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*. 600–609.

[20] Jordi Duch and Alex Arenas. 2005. Community detection in complex networks using extremal optimization. *Physical review E* 72, 2 (2005), 027104.

[21] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. 2024. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130* (2024).

[22] Imen Ben El Kouni, Wafa Karoui, and Lotfi Ben Romdhane. 2021. WLNI-LPA: Detecting Overlapping Communities in Attributed Networks based on Label Propagation Process.. In *ICSOFT*. 408–416.

[23] Golnoosh Farnadi, Zeinab Mahdavifar, Ivan Keller, Jacob Nelson, Ankur Teredesai, Marie-Francine Moens, and Martine De Cock. 2015. Scalable adaptive label propagation in Grappa. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 1485–1491.

[24] S. Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.

[25] O. Gach and J. Hao. 2014. Improving the Louvain algorithm for community detection with modularity maximization. In *Artificial Evolution: 11th International Conference, Evolution Artificielle, EA , Bordeaux, France, October 21-23, . Revised Selected Papers 11*. Springer, Springer, Bordeaux, France, 145–156.

[26] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarria-Miranda, A. Khan, and A. Gebremedhin. 2018. Distributed louvain algorithm for graph community detection. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Vancouver, British Columbia, Canada, 885–895.

[27] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2021. Scalable Shared-Memory Hypergraph Partitioning. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 16–30.

[28] S. Gregory. 2010. Finding overlapping communities in networks by label propagation. *New Journal of Physics* 12 (10 2010), 103018. Issue 10.

[29] Roger Guimerà, DB Stouffer, Marta Sales-Pardo, EA Leicht, MEJ Newman, and Luis AN Amaral. 2010. Origin of compartmentalization in food webs. *Ecology* 91, 10 (2010), 2941–2951.

[30] M. Halappanavar, H. Lu, A. Kalyanaraman, and A. Tumeo. 2017. Scalable static and dynamic community detection using Grappolo. In *IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA USA, 1–6.

[31] Nandinee Haq and Z Jane Wang. 2016. Community detection from genomic datasets across human cancers. In *2016 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 1147–1150.

[32] Mark Harris. 2017. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. [Online; accessed 2024-11-02].

[33] Mark Harris and Kyrylo Perelygin. 2017. Cooperative Groups: Flexible CUDA Thread Programming. https://developer.nvidia.com/blog/cooperative-groups/. [Online; accessed 2024-11-02].

[34] Vitali Henne. 2015. *Label propagation for hypergraph partitioning*. Ph. D. Dissertation. Karlsruher Institut für Technologie (KIT).

[35] Alexandre Hollocou, Julien Maudet, Thomas Bonald, and Marc Lelarge. 2017. A linear streaming algorithm for community detection in very large networks. *arXiv preprint arXiv:1703.02955* (2017).

[36] Alexandre Hollocou, Julien Maudet, Thomas Bonald, and Marc Lelarge. 2017. A streaming algorithm for graph clustering. *arXiv preprint arXiv:1712.04337* (2017).

[37] S. Kang, C. Hastings, J. Eaton, and B. Rees. 2023. cuGraph C++ primitives: vertex/edge-centric building blocks for parallel graph computing. In *IEEE International Parallel and Distributed Processing Symposium Workshops*. 226–229.

[38] Arnav Kapoor, Rishi Raj Jain, Avinash Prabhu, Tanvi Karandikar, and Ponnurangam Kumaraguru. 2021. "I'll be back": Examining Restored Accounts On Twitter. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*. 71–78.

[39] Pan-Jun Kim, Dong-Yup Lee, and Hawoong Jeong. 2009. Centralized modularity of N-linked glycosylation pathways in mammalian cells. *PloS one* 4, 10 (2009), e7317.

[40] K. Kloster and D. Gleich. 2014. Heat kernel based community detection. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, New York, USA, 1386–1395.

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

[42] Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa. 2017. Gpu-accelerated graph clustering via parallel label propagation. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 567–576.

[43] Massimo La Morgia, Alessandro Mei, Alberto Maria Mongardini, and Jie Wu. 2021. Uncovering the dark side of Telegram: Fakes, clones, scams, and conspiracy movements. *arXiv preprint arXiv:2111.13530* (2021).

[44] Rongrong Li, Wenzhong Guo, Kun Guo, and Qirong Qiu. 2015. Parallel multi-label propagation for overlapping community detection in large-scale networks. In *Multi-disciplinary Trends in Artificial Intelligence: 9th International Workshop, MIWAI 2015, Fuzhou, China, November 13-15, 2015, Proceedings 9*. Springer, 351–362.

[45] Panagiotis Liakos, Alexandros Ntoulas, and Alex Delis. 2017. COEUS: community detection via seed-set expansion on graph streams. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 676–685.

[46] Panagiotis Liakos, Katia Papakonstantinopoulou, Alexandros Ntoulas, and Alex Delis. 2020. Rapid detection of local communities in graph streams. *IEEE Transactions on Knowledge and Data Engineering* 34, 5 (2020), 2375–2386.

[47] Yuan Lin. 2018. Using CUDA warp-level primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/. [Online; accessed 2024-11-02].

[48] H. Lu, M. Halappanavar, and A. Kalyanaraman. 2015. Parallel heuristics for scalable community detection. *Parallel computing* 47 (Aug 2015), 19–37.

[49] Milo Lurati, Stijn Heldens, Alessio Sclocco, and Ben van Werkhoven. 2024. Bringing Auto-Tuning to HIP: Analysis of Tuning Impact and Difficulty on AMD and Nvidia GPUs. In *European Conference on Parallel Processing*. Springer, 91–106.

[50] Jun Ma, Jenny Wang, Laleh Soltan Ghoraie, Xin Men, Benjamin Haibe-Kains, and Penggao Dai. 2019. A comparative study of cluster detection algorithms in protein–protein interaction for drug target discovery and drug repurposing. *Frontiers in pharmacology* 10 (2019), 109.

[51] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2017. Parallel graph partitioning for complex networks. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2625–2638.

[52] Jayadev Misra and David Gries. 1982. Finding repeated elements. *Science of computer programming* 2, 2 (1982), 143–152.

[53] Anuraj Mohan, R Venkatesan, and KV Pramod. 2017. A scalable method for link prediction in large real world networks. *J. Parallel and Distrib. Comput.* 109 (2017), 89–101.

[54] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo. 2017. Community detection on the GPU. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Orlando, Florida, USA, 625–634.

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

[56] N. Ozaki, H. Tezuka, and M. Inaba. 2016. A simple acceleration method for the Louvain algorithm. *International Journal of Computer and Electrical Engineering* 8, 3 (2016), 207.

[57] Leto Peel, Daniel B Larremore, and Aaron Clauset. 2017. The ground truth about metadata and community detection in networks. *Science advances* 3, 5 (2017), e1602548.

[58] Chengbin Peng, Tamara G Kolda, and Ali Pinar. 2014. Accelerating community detection by using k-core subgraphs. *arXiv preprint arXiv:1403.2226* (2014).

[59] Ovidiu Popa, Einat Hazkani-Covo, Giddy Landan, William Martin, and Tal Dagan. 2011. Directed networks reveal genomic barriers and DNA repair bypasses to lateral gene transfer among prokaryotes. *Genome research* 21, 4 (2011), 599–609.

[60] 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), 036106–1–036106–11.

[61] Jörg Reichardt and Stefan Bornholdt. 2006. Statistical mechanics of community detection. *Physical review E* 74, 1 (2006), 016110.

[62] Corban G Rivera, Rachit Vakil, and Joel S Bader. 2010. NeMo: network module identification in Cytoscape. *BMC bioinformatics* 11 (2010), 1–9.

[63] Hamid Roghani, Asgarali Bouyer, and Esmaeil Nourani. 2021. PLDLS: A novel parallel label diffusion and label Selection-based community detection algorithm based on Spark in social networks. *Expert Systems with Applications* 183 (2021), 115377.

[64] M. Rosvall and C. Bergstrom. 2008. Maps of random walks on complex networks reveal community structure. *Proceedings of the national academy of sciences* 105, 4 (2008), 1118–1123.

[65] R. Rotta and A. Noack. 2011. Multilevel local search algorithms for modularity clustering. *Journal of Experimental Algorithmics (JEA)* 16 (2011), 2–1.

[66] Subhajit Sahu. 2023. GVE-LPA: Fast Label Propagation Algorithm (LPA) for Community Detection in Shared Memory Setting. *arXiv preprint arXiv:2312.08140* (2023).

[67] Subhajit Sahu. 2024. *v*-LPA: Fast GPU-based Label Propagation Algorithm (LPA) for Community Detection. *arXiv preprint arXiv:2411.11468* (2024).

[68] Marcel Salathé and James H Jones. 2010. Dynamics and control of diseases in networks with community structure. *PLoS computational biology* 6, 4 (2010), e1000736.

[69] Mohammad Sattari and Kamran Zamanifar. 2018. A spreading activation-based label propagation algorithm for overlapping community detection in dynamic social networks. *Data & Knowledge Engineering* 113 (2018), 155–170.

[70] J. Shi, L. Dhulipala, D. Eisenstat, J. Łącki, and V. Mirrokni. 2021. Scalable community detection via parallel correlation clustering.

[71] George M Slota, Cameron Root, Karen Devine, Kamesh Madduri, and Sivasankaran Rajamanickam. 2020. Scalable, multi-constraint, complex-objective graph partitioning. *IEEE Transactions on Parallel and Distributed Systems* 31, 12 (2020), 2789–2801.

[72] Jyothish Soman and Ankur Narang. 2011. Fast community detection algorithm with gpus and multicore architectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 568–579.

[73] C.L. Staudt, A. Sazonovs, and H. Meyerhenke. 2016. NetworKit: A tool suite for large-scale complex network analysis. *Network Science* 4, 4 (2016), 508–530.

[74] Christian L Staudt and Henning Meyerhenke. 2015. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2015), 171–184.

[75] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting label propagation for distributed connected components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 540–546.

[76] V. Traag. 2015. Faster unfolding of communities: Speeding up the Louvain algorithm. *Physical Review E* 92, 3 (2015), 032801.

[77] V.A. Traag and L. Šubelj. 2023. Large network community detection by fast label propagation. *Scientific Reports* 13, 1 (2023), 2701.

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

[79] Lucreția Udrescu, Paul Bogdan, Aimée Chiș, Ioan Ovidiu Sîrbu, Alexandru Topîrceanu, Renata-Maria Văruț, and Mihai Udrescu. 2020. Uncovering new drug properties in target-based drug–drug similarity networks. *Pharmaceutics* 12, 9 (2020), 879.

[80] Joshua Uyheng, Aman Tyagi, and Kathleen M Carley. 2021. Mainstream consensus and the expansive fringe: characterizing the polarized information ecosystems of online climate change discourse. In *Proceedings of the 13th ACM Web Science Conference 2021*. 196–204.

[81] Alan Valejo, Thiago Faleiros, Maria Cristina Ferreira de Oliveira, and Alneu de Andrade Lopes. 2020. A coarsening method for bipartite networks via weight-constrained label propagation. *Knowledge-Based Systems* 195 (2020), 105678.

[82] Ann Verhetsel, Joris Beckers, and Jeroen Cant. 2022. Regional retail landscapes emerging from spatial network analysis. *Regional Studies* 56, 11 (2022), 1829–1844.

[83] L. Waltman and N. Eck. 2013. A smart local moving algorithm for large-scale modularity-based community detection. *The European physical journal B* 86, 11 (2013), 1–14.

[84] Changzhen Wang, Fahui Wang, and Tracy Onega. 2021. Network optimization approach to delineating health care service areas: Spatially constrained Louvain and Leiden algorithms. *Transactions in GIS* 25, 2 (2021), 1065–1081.

[85] Meng Wang, Yanhao Yang, David Bindel, and Kun He. 2023. Streaming local community detection through approximate conductance. *IEEE Transactions on Big Data* (2023).

[86] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

[87] Yan Wang, Rongrong Ji, and Shih-Fu Chang. 2013. Label propagation from imagenet to 3d point clouds. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3135–3142.

[88] C. Wickramaarachchi, M. Frincu, P. Small, and V. Prasanna. 2014. Fast parallel algorithm for unfolding of communities in large graphs. In *IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, IEEE, Waltham, MA USA, 1–6.

[89] Xiaolong Xu, Nan Hu, Tao Li, Marcello Trovati, Francesco Palmieri, Georgios Kontonatsios, and Aniello Castiglione. 2019. Distributed temporal link prediction algorithm based on label propagation. *Future generation computer systems* 93 (2019), 627–636.

[90] Chang Ye, Yuchen Li, Bingsheng He, Zhao Li, and Jianling Sun. 2023. Large-Scale Graph Label Propagation on GPUs. *IEEE Transactions on Knowledge and Data Engineering* (2023).

[91] Bagher Zarei, Mohammad Reza Meybodi, and Behrooz Masoumi. 2020. Detecting community structure in signed and unsigned social networks by using weighted label propagation. *Chaos: An Interdisciplinary Journal of Nonlinear Science* 30, 10 (2020).

[92] Weitong Zhang, Ronghua Shang, and Licheng Jiao. 2023. Large-scale community detection based on core node and layer-by-layer label propagation. *Information Sciences* 632 (2023), 1–18.

[93] Xian-Kun Zhang, Jing Ren, Chen Song, Jia Jia, and Qian Zhang. 2017. Label propagation algorithm for community detection based on node importance and label influence. *Physics Letters A* 381, 33 (2017), 2691–2698.

[94] Yu Zheng, Yongxin Zhu, Shijin Song, Peng Xiong, Zihao Cao, and Junjie Hou. 2018. Improved weighted label propagation algorithm in social network computing. In *17th IEEE TrustCom / 12th IEEE BigDataSE*. IEEE, 1799–1803.

A APPENDIX

A.1 Our Weighted Boyer-Moore (BM) based GPU Implementation of LPA

Algorithm 3 presents the pseudocode for our GPU implementation of LPA, which we refer to as *v*BM-LPA. This method leverages the weighted Boyer-Moore (BM) majority voting algorithm. The main function of the algorithm is `lpa()`. It takes a graph G as input and outputs the set of community labels C assigned to each vertex.

In the algorithm, we begin by initializing each vertex i in the graph G with a unique label. In particular, we set $C[i]$ to i (line 2). We then perform LPA iterations, up to a maximum number of *MAX_ITERATIONS* (line 3). During these, we periodically enable the Pick-Less (PL) mode (line 5) every ρ iterations — starting from the first iteration — to reduce the impact of community swaps. Next, in each iteration, we invoke `lpaMove()` (line 6) to update labels based on the local neighborhood information. If the proportion of changed vertices $\Delta N/N$ falls below the specified tolerance τ and the PL mode is not active, convergence has been achieved, and we break out of the loop (line 7). However, if the PL mode is active, it may lead to fewer label updates, which could falsely trigger convergence. Thus, the loop continues during active PL mode. Once convergence is achieved, the final set of community labels is returned (line 8).

Each iteration of LPA is performed in the `lpaMove()` function (line 9). Here, first, the number of changed vertices ΔN is initialized to zero, with separate counters for each thread group (line 10). Each vertex i is then processed in parallel (line 11) to determine the best community label using a weighted BM majority vote. This involves scanning i's neighbors (lines 14-18), and updating the candidate label $c^\#$ and its weight $w^\#$ to reflect the most frequent neighboring community label by weight. If the majority label $c^\#$ differs from i's current label and satisfies criteria defined by the PL strategy, i's label is updated to $c^\#$ (lines 20-23). Additionally, all of i's neighbors are marked as unprocessed to ensure label changes propagate in subsequent iterations. Note that each vertex is processed by a thread group, which can be either a single thread or a thread block, depending on the degree of the vertex. When a thread block is used, the threads within it collaborate using shared memory to determine the best label for the vertex. After label updates, the changed counts ΔN_G from each thread group are combined using atomic addition (line 25). The total count of changed vertices ΔN is then returned (line 26), allowing the main loop in `lpa()` to decide whether to continue iterating or halt (if convergence has been achieved).

A.2 Alternative Weighted Misra-Gries (MG) based GPU Implementation of LPA

Algorithm 4 presents a GPU-based Misra-Gries (MG) implementation of LPA that uses a single shared MG sketch per vertex (*non-merge based*) and supports rescanning the top-k weighted labels to determine the most weighted label for each vertex. While this approach does not improve performance, it is included for comparison. Here, as before, the `lpa()` function takes a graph G as input and outputs the community labels C for each vertex in G.

In `lpa()`, the algorithm starts by assigning each vertex a unique label, setting $C[i]$ to i (line 2). It then iterates up to a maximum of *MAX_ITERATIONS*, or until convergence (lines 3-7). To mitigate unnecessary label swaps, it switches to Pick-Less (PL) mode every ρ iterations, including the first iteration (line 5), as earlier. During each iteration, the `lpaMove()` function updates label assignments (line 6). The algorithm stops early if the fraction $\Delta N/N$ of label changes drops below a threshold τ, indicating convergence (line 7).

Each iteration of the LPA is executed in the `lpaMove()` function (line 9), which updates the community label of each unprocessed vertex i in the graph G. As earlier, each vertex i is assigned one or more thread groups based on its degree. At the start of `lpaMove()`, the MG sketch arrays for labels S_k and weights S_v are initialized, along with the total count of changed vertices ΔN and the counts of changed vertices for each thread group ΔN_G (lines 10-12). Each vertex i in G is then processed in parallel (line 14). The process begins with scanning the neighboring communities of vertex i to identify the top-k weighted labels. During this step, the threads

Algorithm 3 *v*BM-LPA: Our GPU-based implementation of LPA, based on weighted Boyer-Moore (BM) majority vote algorithm.

▷ $G(V, E)$: Input graph
▷ C: Community label of each vertex
□ N: Number of vertices in G, i.e., $|V|$
□ $c^\#$: Majority weighted label for vertex i
□ ΔN: Number of changed vertices, overall
□ ΔN_G: Changed vertices per thread group
□ g: Current thread group ID
□ ρ: Iteration gap for pick-less mode
□ τ: Iteration tolerance

1: **function** LPA(G)
2: $C \leftarrow [0..|V|)$
3: **for all** $l_i \in [0 \ \ldots$ MAX_ITERATIONS$)$ **do**
4: ▷ Mitigate community swaps with **pick-less**
5: **if** $l_i \bmod \rho = 0$ **then** employ **pick-less** mode
6: $\Delta N \leftarrow lpaMove(G, C)$
7: **if not pick-less and** $\Delta N/N < \tau$ **then break**
8: **return** C

9: **function** LPAMOVE(G, C)
10: $\Delta N \leftarrow 0$; $\Delta N \leftarrow \{0\}$
11: **for all unprocessed** $i \in V$ **in parallel do**
12: ▷ Find best community label for vertex i
13: $c^\# \leftarrow C[i]$; $w^\# \leftarrow 0$
14: **for all** $(j, w) \in G.neighbors(i)$ **in parallel do**
15: **if** $i = j$ **then continue**
16: **if** $C[j] = c^\#$ **then** $w^\# \leftarrow w^\# + w$
17: **else if** $w^\# > w$ **then** $w^\# \leftarrow w^\# - w$
18: **else** $c^\# \leftarrow C[j]$; $w^\# \leftarrow w$
19: ▷ Change label of vertex i to majority label $c^\#$
20: **if** $c^\# \neq C[i]$ **and** (**not pick-less or** $c^\# < C[i]$) **then**
21: $C[i] \leftarrow c^\#$; $\Delta N_G[g] \leftarrow \Delta N_G[g] + 1$
22: **for all** $j \in G.neighbors(i)$ **in parallel do**
23: Mark j as unprocessed
24: ▷ Update number of changed vertices
25: $atomicAdd(\Delta N, \Delta N_G[g])$ **in parallel**
26: **return** ΔN

Algorithm 4 A GPU-based implementation of LPA, based on weighted Misra-Gries (MG) heavy hitters algorithm, where all threads update a single shared sketch directly, eliminating the need for a merging step. It also supports rescanning sub-majority labels.

▷ $G(V, E)$: Input graph
▷ C: Community label of each vertex
□ N: Number of vertices in G, i.e., $|V|$
□ $S(S_k, S_v)$: Labels, weights array of the MG sketch
□ $c^{\#}$: Sub-majority weighted label for vertex i
□ ΔN: Number of changed vertices, overall
□ ΔN_G: Changed vertices per thread group/block
□ k: Number of slots in the MG sketch
□ s: Slot index for the current thread
□ g: Current thread group ID
□ t: Current thread ID
□ ρ: Iteration gap for pick-less mode
□ τ: Iteration tolerance

1: **function** LPA(G)
2: $C \leftarrow [0..|V|)$
3: **for all** $l_i \in [0 \ \ldots$ MAX_ITERATIONS$)$ **do**
4: ▷ Mitigate community swaps with **pick-less**
5: **if** $l_i \bmod \rho = 0$ **then** employ **pick-less** mode
6: $\Delta N \leftarrow lpaMove(G, C)$
7: **if not pick-less and** $\Delta N/N < \tau$ **then break**
8: **return** C

9: **function** LPAMOVE(G, C)
10: $S_k \leftarrow \{\}$; $S_v \leftarrow \{\}$
11: $\Delta N \leftarrow 0$; $\Delta N_G \leftarrow \{0\}$
12: $s \leftarrow t \bmod k$; $g \leftarrow \lfloor t/k \rfloor$ **on each thread**
13: **Use shared mode throughout**
14: **for all unprocessed** $i \in V$ **in parallel do**
15: ▷ Scan communities connected to vertex i
16: $sketchClear(S)$ **in parallel**
17: **for all** $(j, w) \in G.neighbors(i)$ **in parallel do**
18: **if** $j = i$ **then continue**
19: $sketchAccumulate(S, C[j], w, s)$ **in parallel**
20: ▷ Rescan sub-majority labels to find the most weighted
21: **if rescan requested then**
22: $sketchClearValues(S)$ **in parallel**
23: **for all** $(j, w) \in G.neighbors(i)$ **in parallel do**
24: **if** $j = i$ **or** $S_k[s] \neq C[j]$ **then continue**
25: $atomicAdd(S_v[s], w)$
26: ▷ Find best community label for vertex i
27: $c^{\#} \leftarrow sketchMaxKey(S)$ **in parallel**
28: ▷ Change label of vertex i to most weighted label $c^{\#}$
29: **if** $c^{\#} \neq C[i]$ **and** (**not pick-less or** $c^{\#} < C[i]$) **then**
30: $C[i] \leftarrow c^{\#}$; $\Delta N_G[g] \leftarrow \Delta N_G[g] + 1$
31: **for all** $j \in G.neighbors(i)$ **in parallel do**
32: Mark j as unprocessed
33: $atomicAdd(\Delta N, \Delta N_G[g])$ **in parallel**
34: **return** ΔN

Algorithm 5 Accumulating a label, and its associated weight, in a weighted Misra-Gries (MG) sketch — without warp-level primitives.

▷ $S(S_k, S_v)$: Labels, weights array of the MG sketch
▷ c, w: Label, weight to accumulate into the MG sketch
▷ s: Slot index for the current thread
□ has: MG sketch has label c? / Free slot index

1: **function** SKETCHACCUMULATE(S, c, w, s)
2: **if** $s = 0$ **then** $has \leftarrow -1$
3: ▷ Add edge weight to community label
4: **if** $S_k[s] = c$ **then**
5: **if not shared then** $S_v[s] \leftarrow S_v[s] + w$
6: **else** $atomicAdd(S_v[s], w)$
7: $has \leftarrow 0$
8: ▷ Done if label is already in the list
9: **if** $has = 0$ **then return** *done*
10: ▷ Find and empty slot, and populate it
11: ▷ Retry if some other thread reserved the free slot
12: **repeat**
13: ▷ Find an empty slot
14: **if** $S_v[s] = 0$ **then** $atomicMax(has, s)$
15: **if** $has < 0$ **then break**
16: ▷ Add community label to list
17: **if** $has = s$ **then**
18: **if not shared then**
19: $S_k[s] \leftarrow c$
20: $S_v[s] \leftarrow w$
21: **else**
22: **if** $atomicCAS(S_v[s], 0, w) = 0$ **then** $S_k[s] \leftarrow c$
23: **else** $has \leftarrow 1$
24: **until not shared or** $has \geq 0$
25: ▷ Subtract edge weight from non-matching labels
26: **if** $has < 0$ **then**
27: **if not shared then** $S_v[s] \leftarrow S_v[s] - w$
28: **else** $atomicAdd(S_v[s], -w)$
29: **return** *done*

first clear the shared sketch S. Subsequently, thread groups collaborate on the shared sketch, accumulating labels from the neighbors $j \in J_i$ of vertex i based on edge weights ($w = w_{ij}$) using the `sketchAccumulate()` function (lines 17-19). An alternative implementation of `sketchAccumulate()` that does not use warp-level primitives is provided in Algorithm 5. It employs atomic operations to ensure thread-safe updates to the shared sketch S. After the neighborhood scan, the shared sketch S is fully populated.

If a rescan is requested (line 21), the algorithm calculates the exact total weight for each of the top-k labels in the sketch S by re-examining i's neighboring vertices, after having cleared the sketch weights (lines 21-25). It then checks if $c^{\#}$ (the most weighted sub-majority label) differs from i's current label and satisfies the PL mode conditions (e.g., $c^{\#} < C[i]$ if PL mode is active). If these conditions are met, i's label is updated to $c^{\#}$, the change count for the thread group ΔN_G is incremented (only by the first thread group for shared vertices), and i's neighbors are marked unprocessed for further updates (lines 29-32). After all vertices are processed, the

thread group counts ΔN_G are combined into a global count ΔN using atomic addition (line 33). The algorithm then returns ΔN.

A.3 Alternative Method for Populating Misra-Gries (MG) sketch

Algorithm 5 presents a method to update a weighted Misra-Gries (MG) sketch, which does *not use warp-level primitives*. Here, the function `sketchAccumulate()` takes as input the MG sketch S, with labels array S_k and weights array S_v, the key-value pair (c, w) to be accumulated, and the current thread's slot index s.

At the beginning of the function, if the current thread is responsible for the first slot ($s = 0$), it initializes *has* to -1, indicating that no match has been found yet. The algorithm proceeds to check if the target label c is already present in the sketch. If the slot s holds the same label ($S_k[s] = c$), the corresponding value $S_v[s]$ is updated by adding w. This operation is performed atomically if the data is shared among threads. Once the key-value pair is updated, the variable *has* is set to 0, indicating that the label was found, and the function returns immediately. If the label is not found, the algorithm attempts to find an empty slot in the sketch where the new key-value pair (c, w) can be inserted. In a loop, each thread checks if its assigned slot is empty ($S_v[s] = 0$). If so, the thread attempts to reserve the slot using an atomic operation, and setting *has* to the index of the slot if it is successful. If no empty slot is found and *has* remains negative, the loop exits. Once a free slot is found (i.e., *has* matches the current thread's slot s), the algorithm inserts the key-value pair (c, w). If data is not shared, the assignment is straightforward: $S_k[s] \leftarrow c$ and $S_v[s] \leftarrow w$. If data is shared among threads, the algorithm uses an atomic compare-and-swap operation to safely set the value. If another thread has already reserved the slot, the function retries until the operation succeeds. Finally, if no suitable slot was available for the label (i.e., *has* remains negative), the algorithm subtracts the weight w from all the slots in the sketch. The subtraction is performed atomically if the sketch is shared. Finally, the function then returns.