2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-9423-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/HiPC56025.2022.00027

Parallel Vertex Color Update on Large Dynamic Networks

Arindam Khanda*, Sanjukta Bhowmick†, Xin Liang*, and Sajal K. Das*
* Missouri University of Science and Technology, Rolla, USA
† University of North Texas, Denton, USA
Email: *{akkcm, xliang, sdas}@mst.edu, †{sanjukta.bhowmick}@unt.edu

***Abstract*—We present the first GPU-based parallel algorithm to efficiently update vertex coloring on large dynamic networks. For single GPU, we introduce the concept of *loosely maintained vertex color update* that reduces computation and memory requirements. For multiple GPUs, in distributed environments, we propose *priority-based ordering* of vertices to reduce the communication time. We prove the correctness of our algorithms and experimentally demonstrate that for graphs of over 16 million vertices and over 134 million edges on a single GPU, our dynamic algorithm is as much as 20x faster than state-of-the-art algorithm on static graphs. For larger graphs with over 130 million vertices and over 260 million edges, our distributed implementation with 8 GPUs produces updated color assignments within 160 milliseconds. In all cases, the proposed parallel algorithms produce comparable or fewer colors than state-of-the-art algorithms.**

***Index Terms*—Parallel/distributed algorithm, Vertex coloring, Dynamic networks.**

I. Introduction

Complex systems of interacting entities are typically modeled using networks or graphs, where the vertices represent the entities while the edges represent dyadic interactions between pairs of vertices. Analyzing properties of such graphs provides insights into the behavior of the underlying systems.

Networks representing complex systems in real-world applications are often very large and their topological structure may change dynamically in terms of addition or deletion of vertices and edges. For example, social network graphs change due to new user registration or account deletion. Recomputing a network property for each (or batch) change being typically expensive, it is more efficient to identify the region of change and accordingly update the property. Two key challenges to achieving desired efficiency are: (i) non-local access pattern of graph algorithms makes them hard to parallelize; and (ii) efficiently identifying the region affected by the change is hard. Recent works [1]–[5] addressed some of these challenges by developing parallel algorithms for updating dynamic networks.

Vertex coloring (i.e., coloring the vertices such that no two adjacent vertices are assigned the same color) is a fundamental problem in graph theory with a wide range of applications including scheduling, channel assignment [6], dynamic community detection [7], and printed circuit testing. Coloring the vertices with the least number of colors is known to be an NP-hard problem [8]. There exist many approximation algorithms and heuristics for coloring static graphs (e.g., [9]–[11]).

Recognizing that the topology of many real-world application graphs may change (e.g., due to the addition or deletion of edges or vertices), the colors assigned to the vertices need to be updated accordingly. For large-scale dynamic graphs, the design of parallel algorithms is essential for efficient update of vertex colors. However, there exist very few algorithms for updating vertex colors on dynamic networks, and even fewer approaches that can scale up to large graphs. This motivates us to design and implement parallel algorithms for updating vertex colors in large dynamic networks.

Our Contributions: The novel contributions of our paper are to develop the *first GPU-based parallel algorithms for large dynamic networks, for both single GPU and multiple GPUs.* Our innovative approach addresses the two key challenges of dynamic network updates as identified above, while improving efficiency. Specifically,

- For single GPU, we design a *loosely-maintained vertex color update* (LVCU) algorithm to minimize computation and use minimum available information for recoloring.
- We propose a novel algorithm for updating colors based on a *priority ordering* of vertices distributed across multiple GPUs. The priority scheme reduces communication across processors and improves the update time.
- Experiments on real-world and synthetic networks demonstrate that colors obtained by our algorithms are *comparable or lower* than those obtained by other state-of-the-art parallel algorithms. Moreover, our implementation shows up to 20X speedup over the baseline GPU implementation for static graphs.

The remainder of the paper is organized as follows: Section II introduces preliminary concepts, and Section III defines the problem. Section IV presents the LVCU parallel algorithm for vertex color update while Section V describes the priority-based distributed algorithm. Section VI evaluates the performance of our dynamic algorithms. Section VII reviews the related work and the final section concludes the paper.

II. Preliminaries

Table I provides the list of symbols used in this paper. Let $G(V, E)$ be an undirected unweighted graph, where V is the vertex-set and E is the edge-set. A (proper) vertex coloring of G is a function that assigns a color (from a set $\mathbb{C}$) to each vertex $u \in V$, such that no two adjacent vertices get the same color. G is k-$colorable$ if it can be properly colored using at most k colors. The chromatic number $\chi(G)$ is the smallest value of k for which G is k-colorable.

2640-0316/22/$31.00 ©2022 IEEE
DOI 10.1109/HiPC56025.2022.00027

An optimal coloring of G using $\chi(G)$ colors is known to be an NP-Hard problem [8]. Various approximation algorithms and heuristics have been proposed for vertex coloring. These methods provide a simple upper bound of $(\delta + 1)$ on the number of colors used, where δ is the maximum degree of a vertex. A greedy coloring picks up vertices following an order maintained by some coloring priority, and assigns the lowest available color from $\mathbb{C}$ to the selected vertex (e.g., [2], [9]–[11]). A *saturation color set* ($\mathcal{SC}$) of a vertex u is the set of colors assigned to its neighbors. Therefore, the set of colors available for u is $\mathbb{C} - \mathcal{SC}(u)$.

TABLE I: List of Symbols

Symbols	Meaning
$G(V, E)$	Undirected unweighted graph
δ	Maximum degree of vertices in G
$\mathbb{C}$	Set of colors
$\mathcal{SC}(u)$	Saturation color set of vertex u
$Part_i$	Partition i
V^i (or E^i)	Set of vertices (or edges) in $Part_i$
ΔE^i	Set of changed edges in $Part_i$
$\mathcal{I}_i$	Set of internal vertices in $Part_i$
$\mathcal{B}_i$	Set of border vertices in $Part_i$
$\mathcal{G}_i^1$	Set of 1-hop ghost vertices in $Part_i$
$\mathcal{G}_i^2$	Set of 2-hop ghost vertices in $Part_i$
ΔE	Set of changed edges in G
Del	Set of deleted edges in G
Ins	Set of inserted edges in G
p	Number of processing units
R_j	j^{th} region in a partition. $1 \leq j \leq 4$

A. Dynamic Graphs

In a dynamic graph, the topology changes with time; the primary changes are edge insertions and deletions. In [1] the authors presented a generic framework for parallel update of dynamic graph property, specifically shortest path updates. The framework starts with a graph and initially computed property. Next, it updates the property by finding the affected sub-graph due to the changes and iteratively recomputes the property for the affected sub-graph. The major steps of this framework are: 1) process the changed edges in batch and update the property; and 2) achieve the correctness iteratively.

In this paper, we will adapt this framework (with modifications) to develop our parallel algorithm for vertex color update in dynamic graphs. Figure 1 illustrates the steps of this template for vertex coloring (See Section III-A and IV).

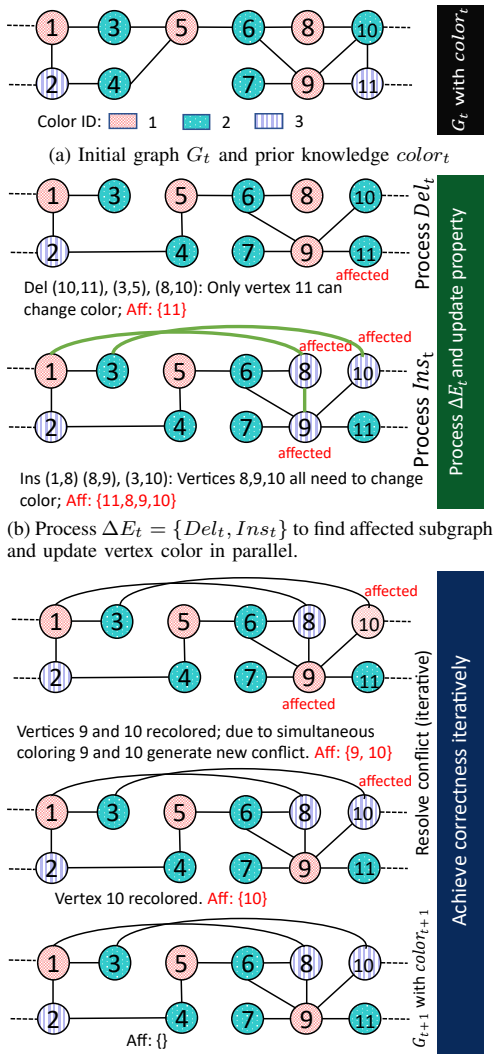
B. Graph Partitioning

A distributed graph algorithm requires partitioning the graph such that each partition contains a disjoint subset of vertices of roughly equal size, and the number of cut edges is minimized. Our distributed approach uses a label propagation-based partitioner, called PuLP [12] that considers an initial partition and iteratively updates the subsequent partitions. This feature is useful for dynamic graph partitioning.

A partition $Part_i$ having vertex-set V^i, for $1 \leq i \leq q$, where $q \in \mathbb{N}$ is the number of processors, contains two types of vertices: (1) ***Internal vertex*** ($\mathcal{I}_i$)***:*** This vertex and all of its neighbors belonging to $Part_i$, and (2) ***Border vertex*** ($\mathcal{B}_i$)***:*** This vertex has at least one neighbor in another partition.

III. Problem Formulation

Without loss of generality, we assume all changes in the dynamic network are due to edge insertions or edge deletions.



(a) Initial graph G_t and prior knowledge $color_t$

(b) Process $\Delta E_t = \{Del_t, Ins_t\}$ to find affected subgraph and update vertex color in parallel.

(c) Color conflict resolution to update color of G_{t+1}.

Fig. 1: A template for dynamic vertex color update.

This is because, it is easy to convert a vertex insertion or deletion into insertion or deletion of edges. At the initial step, the input to the algorithm is (i) a properly colored graph, and (ii) a batch of changed edges that may potentially influence the color of the affected vertices. In subsequent steps, the algorithm only accepts new batches of changed edges and uses the colored output graph from the previous step to update the color. Thus, our algorithm aims to update colors in a dynamic graph rather than re-coloring the changed graph from scratch.

Problem Statement: Let $G_t(V_t, E_t)$ be the graph at time step t and $u.color_t$ be the corresponding color assignment of a vertex u. Let $\Delta E_t = E_{t+1} - E_t$ be the set of changed edges from time step t to $t+1$. It consists of two subsets, Ins_t and Del_t, respectively the set of inserted and deleted edges at time step t. Thus, $E_{t+1} = ((E_t \cup Ins_t) \setminus Del_t)$. Our *goal* is to efficiently compute the color assignment $u.color_{t+1}$ for all $u \in V_{t+1}$, without recomputing from scratch.

A. Overview of the Color Update Template

The insertion or deletion of an edge $e = (u, v)$ in G requires updating the color of at most one of the end vertices u and v to maintain proper coloring. In the Single Instruction Multiple Threads (SIMT) architecture, similar operations are processed simultaneously. Our template first processes the edges from Del in parallel and then the edges from Ins. In course of this process, some vertices may need to be recolored to maintain proper coloring. However, if this recoloring is done in parallel, two adjacent vertices may be assigned the same color due to asynchronous updates, leading to new color conflicts. We resolve this conflict by selecting and recoloring only one of the conflicting vertices, and iteratively recoloring as necessary. In every iteration, the number of potential conflicts decreases, and the process finally converges. The next section provides details on the proposed approach.

IV. Loosely-maintained Vertex Color Update (LVCU) Algorithm

In practice, a graph requires fewer colors than $\delta + 1$ (upper bound) and it gives us an opportunity to use a small subset of saturation color set for recoloring.

While recoloring a vertex u, our algorithm selects the available color with lowest id from a fixed size partial saturation color set $\mathcal{SC}'(u)$, s.t. $\mathcal{SC}'(u) \subseteq \mathcal{SC}(u)$ and maintains the upper bound of $(\delta + 1)$ colors [$\mathcal{SC}'$ in Section VI]. Therefore, a local minimum is used in place of the global minimum to choose a color. Operations on comparatively smaller $\mathcal{SC}'(u)$ decrease memory usage and computation time. We name this coloring as *loosely maintained coloring* and propose Loosely-maintained Vertex Color Update (LVCU) algorithm (see Algorithm 1) for dynamic graphs.

Our proposed algorithm consists of two key steps, namely processing changed edges (ProcessCE), and resolving conflicts and updating neighbor colors, as described below.

A. Step 1–Process Changed Edges

This step processes all changed edges in parallel. For each changed edge, at most one endpoint is selected for recoloring.

For deletion of an edge $(y, z) \in Del$, the endpoint with higher color (say y) is checked and recolored with the other endpoint's color $z.color$, if $z.color$ is not in the saturation set of y. If recolored, the vertex is considered as affected and stored in a set Aff for further processing. We have observed that recoloring the vertex with the higher color may help to maintain better color quality by reducing the total number of used color in long run. This approach does not require us to store $\mathcal{SC}$ as we can check if $z.color$ is used by any neighbor of y, simply by visiting the neighbors.

For edge insertion, the non-trivial case arises when both the endpoints have the same color. Otherwise, no vertex color update is required. For inserted edge $(a, b) \in Ins$, if $a.color = b.color$ then an endpoint is selected as candidate for recoloring. The candidate selection usually follows one of two approaches: (i) *property based* which selects the candidate by comparing the vertex property such as degree, saturation degree or hashed value of the endpoints; and (ii) *randomized*

Algorithm 1: LVCU

Input: $\mathbb{C}$, $G(V, E)$, $\Delta E = \{Ins, Del\}$, user defined max iteration itr_{max}
Output: Updated colored vertices of G at $t + 1$

Function `LVCU_Main(`G, $\mathbb{C}$, ΔE`)`**:**
  Initialize empty sets Aff and S
  $itr \leftarrow 0$
  $Aff \leftarrow$ **ProcessCE(**G, Del, Ins, Aff**)**
  while Aff *is not empty* **do**
    $S \leftarrow Aff$, $Aff \leftarrow \emptyset$
    for *each vertex* $v \in S$, *in parallel* **do**
      $Aff \leftarrow Aff \cup$ **CheckConflict(**G, v**)**
      if $itr < itr_{max}$ **then**
        $Aff \leftarrow Aff \cup$ **UpdateNeighbors(**G, v**)**;
    $itr \leftarrow itr + 1$

Function `ProcessCE(`G, Del, Ins, Aff`)`**:**
  for *each edge* $(a, b) \in Del$, *in parallel* **do**
    $y \leftarrow argmax_{x \in \{a,b\}}(x.color)$
    $z \leftarrow argmin_{x \in \{a,b\}}(x.color)$
    if $z.color \notin \mathcal{SC}(y)$ **then**
      $y.color_{prev} \leftarrow y.color$, $y.color \leftarrow z.color$
      add y to the set Aff
    delete (a, b) from G
  for *each edge* $(a, b) \in Ins$, *in parallel* **do**
    add (a, b) in G
    if $a.color = b.color$ **then**
      $y \leftarrow argmax_{x \in \{a,b\}}(x.id)$
      $y.color_{prev} \leftarrow y.color$
      $y.color \leftarrow$ min. available color in $\mathcal{SC}'(y)$
      add y to the set Aff
  return Aff

Function `CheckConflict(`G, v`)`**:**
  $A \leftarrow \emptyset$ ▷ stores affected vertices locally
  for $u \in V$ *and* u *is neighbor of* v **do**
    if $u.color == v.color$ **then**
      $y \leftarrow argmax_{x \in \{u,v\}}(x.id)$
      $A \leftarrow A \cup y$
      $y.color_{prev} \leftarrow y.color$
      $y.color \leftarrow$ min. available color in $\mathcal{SC}'(y)$
  return A

Function `UpdateNeighbors(`G, v`)`**:**
  $A \leftarrow \emptyset$ ▷ stores affected vertices locally
  for *vertex* u, *where* $v.color_{prev} < u.color$ *and* u *is neighbor of* v **do**
    if $v.color_{prev} \notin \mathcal{SC}(u)$ **then**
      $u.color_{prev} \leftarrow u.color$
      $u.color \leftarrow v.color_{prev}$
      add u to the set A
  return A

which selects one of the endpoints randomly. The former may suffer from high computation and memory overhead while the latter may delay the convergence or lead to color inconsistency in the distributed setup (See Section V-B3).

Instead of storing additional vertex property, LVCU selects the endpoint with higher vertex ID as candidate (say y) and recolors using the minimum available color in $\mathcal{SC}'(y)$.

B. Step 2–Resolve Color Conflict via Iteration

Color conflict between two adjacent vertices may arise due to parallel recoloring. To address this issue, LVCU visits all the recolored vertices in parallel, finds color conflicts and resolves them iteratively by choosing and recoloring the vertex with larger ID.

When a vertex v is recolored, it's previous color, say k, becomes free. It may be possible to reduce the number of colors if we recolor some of the neighbors of v with the color k. This step is not necessary to maintain the color correctness. We therefore limit this search by a user defined maximum iteration, itr_{max} (Algorithm 1, Line 9).

The LVCU reduces the total amount of work by relaxing the efforts to minimize the number of colors used. The maximum number of colors depends mainly on the color range of the input graph and the number of insertions. In a large dense graph, the saturation color set can be large. As the algorithm does not require to store the full saturation color set, it is suitable for parallel architectures with relatively smaller memory, such as GPUs.

C. Theoretical Analysis

We present theorems related to the correctness of our algorithm and a theoretical bound on its computational complexity.

Theorem 1. *In LVCU, the iterations to resolve the color conflict (Algorithm 1, Lines 4-10). will converge.*

Proof. If two asynchronous threads recolor two adjacent vertices, color conflict may arise between these recolored vertices. We do not consider coloring the neighbors based on free colors, as this is an optional step.

Let $\mathcal{A}_i$ is the set of vertex IDs among which color conflict arises at i^{th} iteration. As all vertex IDs are natural numbers, the relation $<$ is a *strict partial order* on $\mathcal{A}_i$. Hence, there should be a minimum vertex ID $u \in \mathcal{A}_i$.

To resolve conflict between any two adjacent vertices from $\mathcal{A}_i$, LVCU chooses the vertex with higher vertex ID and recolors it. Therefore, u should be never selected at any time while selecting a vertex for recoloring. So, in the worst case, LVCU recolors $(|\mathcal{A}_i| - 1)$ vertices at i^{th} iteration of conflict resolution. These new set of recolored vertices will be a subset of $\mathcal{A}_i$ as no new vertex is recolored in the process of conflict resolution. Therefore, in the worst case, color conflict may arise among recolored $(|\mathcal{A}_i|-1)$ vertices at $(i+1)^{th}$ iteration. In this process, the number of vertices with conflicts will keep on decreasing as the iterations increase, and will eventually become zero, resulting in convergence. □

Theorem 2. *If the initial graph G is properly colored by choosing colors uniformly at random from a palette of size $k(1 < k < \delta + 1)$; with p processing units, the average-case time complexity of LVCU algorithm (keeping $itr_{max} = 0$ or assuming the neighbors are not updated) on G and change edges $\Delta E = \{Ins, Del\}$ is $O(\frac{\delta}{p}(|Del| + \frac{|Ins|}{k^2}))$.*

Proof. Deletion. The LVCU algorithm first processes all deleted edges in parallel and selects an end-vertex u of every deleted edge $(u, v) \in Del$ for recoloring. It requires $O(\frac{|Del|}{p})$ time. Then, for each selected vertex u, its neighbors are checked to determine if u can be recolored using the color of vertex v. Given that δ is the maximum degree of the graph, and therefore $\delta + 1$, the upper limit of the colors required, in worst case this step will have complexity $O(\frac{|Del|}{p}\delta)$. As the initial graph uses a palette of k colors, the probability that no neighbor of u is colored with the color of v is $(1-\frac{1}{k})^{\delta}$. So the expected number of deletion affected vertices is $|Del|(1-\frac{1}{k})^{\delta}$.

Insertion. Next the inserted edges are processed in parallel. Based on our assumption that colors are selected uniformly at random, the probability that both the end-vertices of an inserted edge have the same color is $\frac{1}{k^2}$. Therefore the expected number of selected vertices for recoloring is $|Ins|\frac{1}{k^2}$. For recoloring a vertex, LVCU prepares a 32-bit bitmap by visiting the neighbors of the selected vertex in $O(\delta)$ time, and colors the vertex using an available color from the bitmap in $O(1)$ time. Therefore, the total time complexity of processing inserted edges is $O(\frac{|Ins|}{p}\frac{1}{k^2}\delta)$.

Iterative Resolution of Color Conflicts. The expected number of recolored vertices (or affected) at step 1 is $\mathcal{X} = |Del|(1-\frac{1}{k})^{\delta} + |Ins|\frac{1}{k^2}$. Step 2 checks color conflict among recolored vertices in parallel and recolors the vertex with higher ID between two conflicting vertices in every iteration. Visiting the initially affected vertices will take $O(\frac{\mathcal{X}}{p})$ time. At first iteration color conflict may arise for $\frac{\mathcal{X}}{k^2}$ (let $\mathcal{Y}$) vertices. As neighbors are not updated, no new vertex is recolored in any iteration. Therefore, a positive real number λ less than 1 can be found such that at every iteration the average number of recolored vertices be λ times the number of conflicting vertices. Therefore, at first iteration $\lambda\mathcal{Y}$ number of vertices will be recolored.

Similarly, in the second iteration conflict may arise among $\frac{\lambda\mathcal{Y}}{k^2}$ vertices and $\frac{\lambda^2\mathcal{Y}}{k^2}$ vertices will be recolored. Step 2 converges iteratively and total amount of time required to check the conflicting vertices and select candidate vertices to recolor is $\frac{\mathcal{Y}}{p}(1 + \frac{\lambda}{k^2} + \frac{\lambda^2}{k^4} + \frac{\lambda^3}{k^6} + \dots) = \frac{\mathcal{Y}k^2}{p(k^2-\lambda)}$.

The total number of vertices recolored (may contain duplicates as a vertex can be recolored in multiple iterations) in step 2 is $\lambda\mathcal{Y}(1+\frac{\lambda}{k^2}+\frac{\lambda^2}{k^4}+\frac{\lambda^3}{k^6}+\dots) = \frac{\lambda\mathcal{Y}k^2}{k^2-\lambda}$. As recoloring a vertex takes $O(\delta)$ sequential time, step 2 recoloring takes $O(\frac{\lambda\mathcal{Y}\delta k^2}{p(k^2-\lambda)})$ time. Hence, total time requirement for step 2 is $O(\frac{\mathcal{Y}k^2}{p(k^2-\lambda)} + \frac{\lambda\mathcal{Y}\delta k^2}{p(k^2-\lambda)}) = O(\frac{\mathcal{Y}k^2}{p(k^2-\lambda)}(\lambda\delta + 1)) = O(\frac{\mathcal{X}}{p(k^2-\lambda)}(\lambda\delta + 1))$.

Therefore, the average time complexity of the algorithm is $O(\frac{|Del|}{p}\delta)+O(\frac{|Ins|}{p}\frac{1}{k^2}\delta)+O(\frac{\mathcal{X}}{p(k^2-\lambda)}(\lambda\delta+1)) = O(\frac{\delta}{p}(|Del|+\frac{|Ins|}{k^2})(1 + \frac{\lambda+1/\delta}{k^2-\lambda}))$. As $\delta, k \in \mathbb{N}$; $k > 1$, and $0 < \lambda < 1$, $\lambda \in \mathbb{R}$, the term $\frac{\lambda+1/\delta}{k^2-\lambda}$ becomes negligible. Therefore, the time complexity of LVCU becomes $O(\frac{\delta}{p}(|Del| + \frac{|Ins|}{k^2}))$. □

Corollary 1. *The expected number of recolored vertices by LVCU (keeping $itr_{max} = 0$) is $|Del|(1 - \frac{1}{k})^{\delta} + |Ins|\frac{1}{k^2}$.*

Proof. From the proof of Theorem 2, we get the expected number of vertices recolored at step 1 is $|Del|(1 - \frac{1}{k})^{\delta} + |Ins|\frac{1}{k^2}$. Step 2 of LVCU finds color conflict within the vertices colored by step 1 only. Hence, the expected number of vertices, recolored by LVCU is $|Del|(1 - \frac{1}{k})^{\delta} + |Ins|\frac{1}{k^2}$. □

V. Distributed Vertex Color Update

While our proposed algorithm efficiently parallelizes the color update in dynamic graphs, generalizing it to distributed environments is non-trivial due to the following challenges.

Inter-partition Communication: In distributed coloring, each partition can individually update the color of their vertices, except the border vertices [13]. As partitions do not have updated information on all the neighbors of a border vertex, coloring border vertices require inter-partition communication.

Also, the conflict resolution among the border vertices requires several rounds of inter-partition communication [14]. In a distributed vertex color update problem, when the aim is to reduce recoloring time and size of the affected sub-graph; inter-partition communication cost becomes dominant.

Maintaining Color Consistency: In static graph coloring, inter-partition communication is typically reduced by copying the neighbors of border vertices as ghost vertices from other partitions [13], [14]. However, in dynamic setup, where two neighboring partitions do not know about each others topological changes, there can be issues of coloring consistency, i.e. the same border vertex can be assigned different colors by different processors to which it belongs.

To address these challenges, we design a priority based distributed LVCU algorithm.

A. Partitioning

We use a label-propagation based partitioner as implemented in PuLP [12]. To reduce the communication cost in each partition $Part_i$, we store two additional types of vertices beside $\mathcal{I}_i$ (internal vertex) and $\mathcal{B}_i$ (border vertex), as follows.

Ghost vertex ($\mathcal{G}_i^1$): If a neighbor vertex u of a border vertex $b \in B_i$ belongs to another partition $Part_j$, then u is copied to $Part_i$. This copied vertex is called ghost vertex in $Part_i$. We term $Part_j$ as the owner patition of u.

2-hop ghost vertex ($\mathcal{G}_i^2$): If a ghost vertex has neighbor vertices in partitions other than $Part_i$, those vertices are copied to $Part_i$ and called 2-hop ghost vertices.

B. Priority Based Distributed LVCU

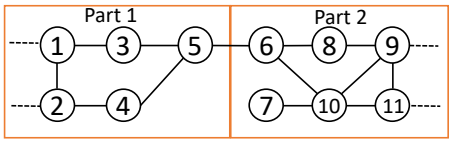
The key insight to our distributed algorithm is as follows. A border vertex that is shared among multiple processors can be simultaneously colored by each of the processors, and yet assigned the same color if (i) the colors of its neighbors are known and is consistent across processors, and (ii) the coloring algorithm is deterministic.

Based on this observation, we propose a novel priority assignment for vertices, such that the vertices for which inter-partition communication becomes unavoidable are the least likely to be recolored. Then we present a distributed coloring algorithm based on the assigned priority to minimize the rounds of inter-partition communications. The proposed distributed LVCU algorithm workflow consists of two phases: 1) Preprocessing, and 2) Color update.

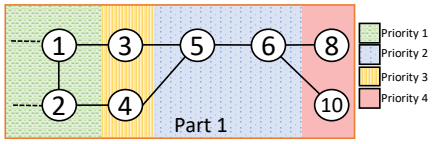
*1) **Preprocessing**:* This phase assigns priority to all the vertices and distributes change edges among the partitions.

Initial Priority Assignment: Initially, in each partition, priority 1 is assigned to all the internal vertices and priority 2 is assigned to the border and ghost vertices. If any internal vertex is adjacent to any border vertex then the priority of that internal vertex is changed to 3. All 2-hop ghost vertices are assigned priority 4 (Algorithm 2).

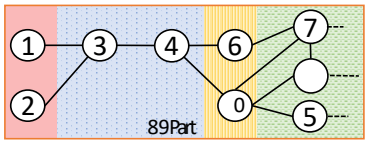
In each partition, the priority assignment creates four non overlapping regions R_1, R_2, R_3, and R_4 containing vertices with priorities 1, 2, 3 and 4 respectively. Figure 2a shows an example graph with two partitions. Figures 2b and 2c show the initial priority assignment in those partitions. Green, blue, yellow, and red areas in these figures indicate R_1, R_2, R_3, and R_4 respectively. The dotted lines indicate the remainder of the subgraph in R_1 which is not shown in the example.



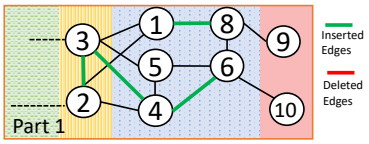
(a) Partition 1 and 2 without any ghost vertex



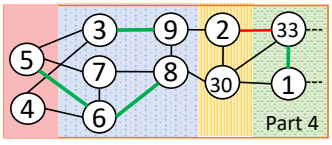
(b) Priority assignment in Partition 1 at t



(c) Priority assignment in Partition 2 at t



(d) Priority assignment in Partition 1 at $t+1$



(e) Priority assignment in Partition 2 at $t+1$

Fig. 2: Priority assignment in different partitions.

Update Priority: Edge insertion can change the initial priority assignment and can move nodes. As seen in Figure 2d, in any partition, nodes can move from R_1 to R_3 (e.g. node 2), from R_1 to R_2 (e.g. node 1), and from R_3 to to R_2. Moreover, based on nodes in the neighboring partition, new

Algorithm 2: Initial Priority Assignment

1 **for** *each partition* $Part_i$ *in* G*, in parallel* **do**
2 Assign priority 1 to each vertex $u \in \mathcal{I}_i$.
3 Assign priority 2 to each vertex $u \in \mathcal{B}_i$ or $u \in \mathcal{G}_i^1$.
4 Assign priority 3 to each vertex $u \in \{x : (x, v) \in E, x \in \mathcal{I}_i, v \in \mathcal{B}_i\}$.
5 Assign priority 4 to each vertex $u \in \mathcal{G}_i^2$.

nodes can move to R_4 (e.g. node 9). Priority change due to edge insertion is described in Algorithm 3. Figures 2d and 2e show the updated priority assignment after considering $Ins_t = \{(1, 8), (3, 2), (4, 6), (7, 11), (3.4)\}$, and $Del_t = \{(9, 11)\}$ in partition 1 and partition 2 respectively. Priority change due to edge deletion does not affect correctness of the algorithm. Therefore, we focus mainly on priority change due to insertion.

Algorithm 3: Change Priority

1 **for** *each partition* $Part_i$ *in* G*, in parallel* **do**
2 $V^{i-} = V^i - \{x : x \in R_4\}$
3 **for** *each edge* $(u, v) \in Ins$ **do**
4 **if** $u \in V^{i-}$ *and* $v \notin V^{i-}$ **then**
5 **if** $u \in \mathcal{G}_i^1$ **then**
6 Assign priority 4 to v and add it in $\mathcal{G}_i^2$.
7 **else**
8 Assign priority 2 to v and add it in $\mathcal{G}_i^1$.
9 Add u in $\mathcal{B}_i$.
10 Reassign priority for the neighbors of u and v.
11 Repeat Line 4–10 by interchanging u and v.

Due to our priority assignment, R_3 always comes between R_1 and R_2, whereas R_3 and R_4 are separated by R_2. It ensures there will be no edge between (R_1,R_2), (R_3,R_4), and (R_1,R_4). Also, considering two adjacent partitions, $Part_i$ and $Part_j$, the set of vertices in R_1 region will be unique to each partition, the set of vertices in region R_2 will be the same for each partition, and the set of vertices in region R_3 of $Part_i$ will be the set of vertices in region R_4 of $Part_j$ and vice-versa. Any vertex $u \in R_2$ belongs to two or more partitions and in all of them the priority of u remains 2 as it is considered as either border or ghost vertex.

The preprocessing step also reads the initial color assignment and distributes the changed edges among different partitions. $Part_i$ stores all changed edges ΔE^i.

2) ***Distributed Color Update****:* It can be observed that in any partition $Part_i$, for all but R_4 vertices, the neighbors are already in $Part_i$. Therefore, if the initial color assignment is known, ΔE^i can be processed and any vertex $u \notin R_4$ can be recolored without additional information from other partitions. Next, if the color for each vertex u is gathered from its owner partition, it will be an updated color as in owner partition, u belongs to R_1, R_2 or R_3. Therefore, information

Algorithm 4: Distributed LVCU

1 **Function** `Dist_LVCU_Main`(G, $\mathbb{C}$, ΔE, itr_{max})**:**
2 Initialize empty set Aff, S, and $\Delta E3$
3 $itr \leftarrow 0$
4 **for** *each edge* $(a, b) \in \Delta E$ *in parallel* **do**
5 **if** $a.priority = b.priority = 3$ **then**
6 Add (a, b) in $\Delta E3$
7 **else if** $a.priority \neq b.priority$ **then**
8 $y \leftarrow (a.priority < b.priority)?a : b$
9 For (a, b) being a non-trivial edge insertion, recolor y using min. available color in $\mathcal{SC}'(y)$
10 For (a, b) being an edge deletion, recolor y using another endpoint's color
11 add y to set Aff
12 **else**
13 $Aff \leftarrow$ **ProcessCE**(G, Del, Ins, Aff)
14 **while** Aff *is not empty* **do**
15 $S \leftarrow Aff$, $Aff \leftarrow \emptyset$
16 **for** *each vertex* $v \in S$*, in parallel* **do**
17 $Aff \leftarrow Aff \cup$ **CheckConflict**(G, v)
18 **if** $itr < itr_{max}$ **then**
19 $Aff \leftarrow Aff \cup$ **UpdateNeighbors**$^+$(v)
20 $itr \leftarrow itr + 1$
21 **ProcessRegion3**($G, \mathbb{C}, \Delta E3$)
22 **Function** `UpdateNeighbors`$^+$`(`G, v`)`**:**
23 $A \leftarrow \emptyset$ ▷ stores affected vertices locally
24 **for** *vertex* n*, where* $v.color_{prev} < n.color$ *and* n *is neighbor of* v **do**
25 **if** $n.priority \neq v.priority$ **then**
26 skip the iteration for n
27 **if** $v.color_{prev} \notin SC(n)$ **then**
28 $n.color_{prev} \leftarrow n.color$, $n.color \leftarrow v.color_{prev}$
29 add n to set A
30 **return** A
31 **Function** `ProcessRegion3`$(G, \mathbb{C}, \Delta E3)$**:**
32 Initialize empty set A and S
33 **for** *each edge* $e(a, b) \in \Delta E3$ *in parallel* **do**
34 **if** $e(a, b)$ *to be inserted* $\wedge$ $a.color = b.color$ **then**
35 $y \leftarrow argmax_{x \in \{a,b\}}(x.id)$
36 $y.color_{prev} \leftarrow y.color$
37 $y.color \leftarrow$ min. available color in $\mathcal{SC}'(y)$
38 add y to set A
39 **while** A *is not empty* **do**
40 $S \leftarrow A$, $A \leftarrow \emptyset$
41 **for** *each vertex* $v \in S$*, in parallel* **do**
42 $A \leftarrow A \cup$ **CheckConflict**(G, v)

about recolored R_4 vertices is not required. However, when for processing the next batch of changes, the updated colors for R_4 vertices are required to recolor R_2 vertices correctly. Therefore, before processing the next batch, updated colors of R_3 vertices need to be sent from the owner partition to the neighboring partitions where those vertices belong to R_4.

We present the following properties of our distributed LVCU algorithm: 1) R_4 is never recolored while processing changes. Color of R_4 is used for recoloring other regions. 2) R_1 is recolored by its owner partition only. 3) R_2 is recolored by multiple partitions, but it gets the same color due to deterministic algorithm. 4) As the updated color of R_3 is exchanged before processing the next batch of changes, R_3 is recolored only when it is utmost necessary to maintain proper coloring. It helps to reduce the size of information exchange.

As the color updates of R_1 and R_2 do not require any inter-partition communication, distributed LVCU processes changed edges and recolor affected vertices completely in R_1 and R_2 at first and then update color in R_3, if necessary.

Step 1–Processing Changed Edges in Regions 1 and 2:

The algorithm processes the changed edges in parallel and inserts the edges with both endpoints in R_3 into a new changed edge set $\Delta E3$ for processing at the end (Algorithm 4 Line 6).

The algorithm also processes other changed edges for which at least one endpoint belongs to R_1 or R_2 and selects the candidate vertices in the lower priority region (R_1 or R_2) for recoloring. For the intra-region changed edges where both the endpoints belong to either R_1 or R_2, the candidate vertices for recoloring are selected using the same logic as in Algorithm 1. Precisely, for edge deletion (resp. insertion) the endpoint with a higher color ID (resp. global ID) is selected. Any candidate vertex is recolored using the minimum available color in $\mathcal{SC}'$. The lowest unset-bit in a fixed bit $\mathcal{SC}'$ with a fixed lowest color ID provides a deterministic way to color R_2 vertices.

Step 2–Resolve conflict in R_1 and R_2: Since no vertex $u \in R_3$ is selected nor recolored at step 1, two conflicting vertices should either be in R_1 or both in R_2. The conflict resolution and neighbor's color update step is similar to the methods in Algorithm 1. The only difference in current approach of neighbor's color update is the constraint prohibiting the process to recolor any R_3 vertex (Algorithm 4 Line 26).

Step 3–Process Changed Edges in R_3: If the color of any vertex $u \in R_3$ is updated at $t+1$, the owner partition requires to send the updated color to its neighbor partitions where u belongs to R_4. This inter-partition communication is not immediately necessary to compute $color_{t+1}$, but necessary before computing $color_{t+2}$. Therefore, this communication can be overlapped with the next preprocessing phase. However, the communication among the partitions can be costly depending on the size of data; hence the number of recolored vertices is minimized by restricting color update only for edge insertion. (Algorithm 4 Line 34). Color conflict in R_3 is resolved locally.

*3) **Discussion on communication reduction**:* Here, we detail how LVCU reduces inter-partition communication costs through the four optimizations below.

2-hop ghosts: Storing the 2-hop ghosts helps to recolor any border vertex and resolve a color conflict between two border vertices without any inter-partition communication.

Deterministic recoloring for R_2: LVCU avoids any randomized variable or operation and uses deterministic approach for recoloring or resolving conflict in R_2. As a result, an owner partition does not require to inform its neighbor about the updated colors in R_2.

Minimized color update in R_3: A novel priority assignment makes R_3 vertices least probable to be recolored. It reduces the number of vertices requiring inter-partition communication.

Use of minimum information: The distributed LVCU uses minimum available information (global ID, priority or color) to select the vertex for recoloring. Unlike existing approaches, it does not store additional vertex properties such as a degree or saturation color set. This helps to reduce the total data transferred for R_4 in the preprocessing phase.

VI. Performance Evaluation

We implemented our algorithms for NVIDIA GPUs using CUDA C++. The adjacency list is stored in a modified compressed sparse row (CSR) format in unified memory. Changed edges are stored using an array of structures where each element stores the endpoints of an edge, and a flag to indicate insertion/deletion status. For lower memory usage and faster recoloring we use a 32-bit bitmap implementation of $\mathcal{SC}'$.

Optimization using 32-bit bitmap: Let G_t be colored using a set of colors $\mathbb{C} = \{1, \dots k\}$. To store $\mathcal{SC}'$ we use a 32-bit bitmap, where i^{th} bit ($1 \leq i \leq 32$) denotes the $(i+j)^{th}$ color in $\mathcal{SC}$. Here, j ($j \in \mathbb{N}$) can be chosen in a way such that $(k-32) < j < k$. In bitmap an unset-bit indicates that the corresponding color is not in the saturation set, $\mathcal{SC}$. Hence, the lowest unset-bit provides the min. available color in the partially stored saturation color set $\mathcal{SC}'$. As CUDA does not have a readily available template for bitmap related operations yet, we use a single integer (32-bit) as a 32-bit bitmap and simple bitwise operators to perform bitmap related operations.

Our algorithm takes advantage of both edge-centric and vertex-centric operations. Each changed edge is processed by each CUDA thread; whereas for conflict checking or neighbor update, each affected vertex is assigned to a single thread. We use a filter kernel after each iteration of Step 2 to uniquely select the affected vertices. This operation reduces redundant work by eliminating duplicates. For distributed implementation, we use MPI for communication among different partitions and use data parallelism at each partition level.

A. Experimental Results

We evaluated the performance of our implementations on Longhorn cluster [15] at Texas Advanced Computing Center. It consists of 104 computing nodes, each equipped with 2 IBM Power System AC922 processors and 8 NVIDIA Tesla V100 GPUs (80 streaming multiprocessors) with 32GB memory. These computing nodes are interconnected via Infiniband and attached to a Lustre file system.

To evaluate the performance of LVCU, we consider five real [16] and four synthesized networks [17], reported in Table II.

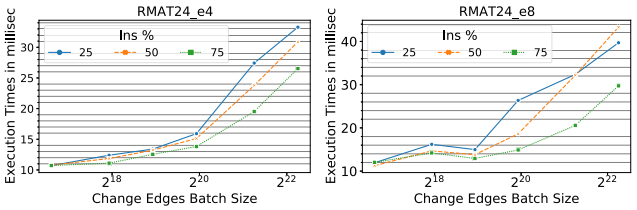
TABLE II: Networks in Our Test Suite [16], [17]

(a) Real Networks

Graph (G)	Alias	Vertices	Edges
IMDB	D1	896,308	3,782,463
soc-orkut	D2	2,997,167	106,349,209
inf-italy-osm	D3	6,686,493	7,013,978
inf-road-usa	D4	23,947,347	28,854,312
soc-sinaweibo	D5	58,655,849	261,321,071

(b) Synthesized Networks

Graph (G)	Alias	Vertices	Edges
RMAT24_e2	D6	16,777,216	33,554,432
RMAT24_e4	D7	16,777,216	67,108,864
RMAT24_e8	D8	16,777,216	134,217,728
RMAT27	D9	134,217,728	268,435,456



(a) RMAT24_e4 (b) RMAT24_e8

Fig. 3: Performance of LVCU on different batch size.

1) Experiment on $|\Delta E|$: Figure 3 shows the execution time of our LVCU CUDA implementation when the batch size of changed edges is varied. Each batch consists of different mixes of edge insertions and deletions. In our experiment, if the batch contains $p\%$ insertions then there is $(100-p)\%$ edge deletion. The result shows our algorithm takes the highest execution time for 25% insertion, i.e., 75% deletion. For every edge deletion, LVCU tries to recolor one endpoint of the edge and visits the neighbors of the endpoint. However, in the case of edge insertion, a vertex is recolored or its neighbors are visited iff both the endpoints of the inserted edge are having the same color. Therefore, the average work for edge deletion is higher than that of edge insertion and a higher percentage of deleted edges in a batch increases the total execution time. In Figure 3, when the batch size is increased exponentially, the time for vertex color update increases gradually.

2) Single-GPU Performance: Figure 4 shows the performance of different steps in the LVCU algorithm. It shows that processing Del takes more time than processing Ins and the deletion processing time is dominant in the whole execution time. We get this result as the average work for processing deleted edges is higher than the average work requirement for processing inserted edges. When the percentage of insertion is increased (or % of deletion is decreased) the whole execution time decreases as the number of deleted edges decreases.

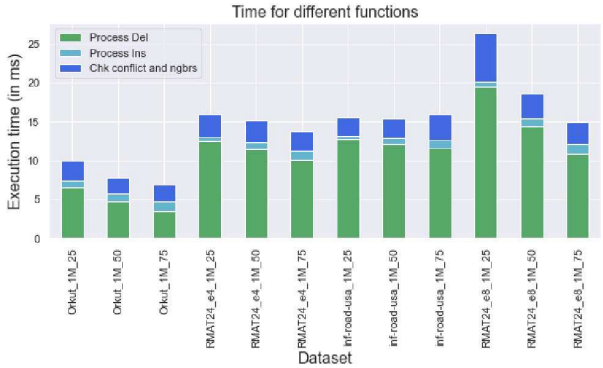
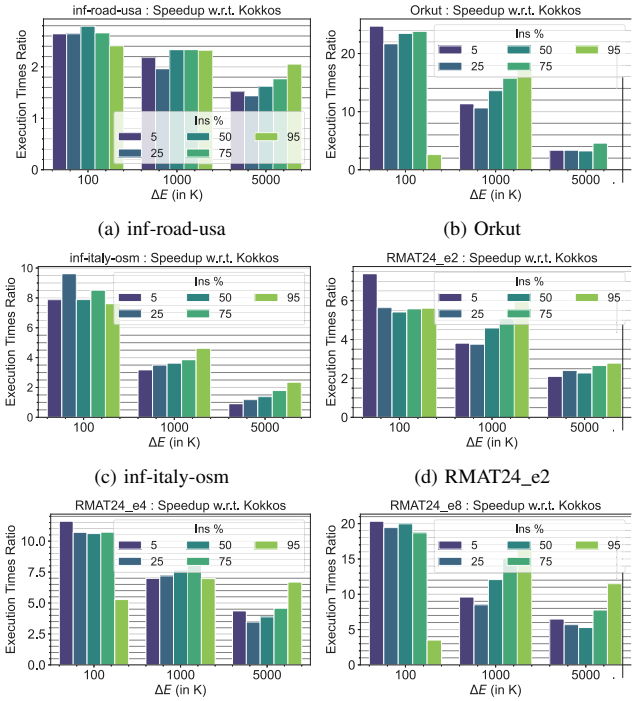


Fig. 4: Execution time of different steps



(a) inf-road-usa (b) Orkut

(c) inf-italy-osm (d) RMAT24_e2

(e) RMAT24_e4 (f) RMAT24_e8

Fig. 5: Single GPU performance comparison.

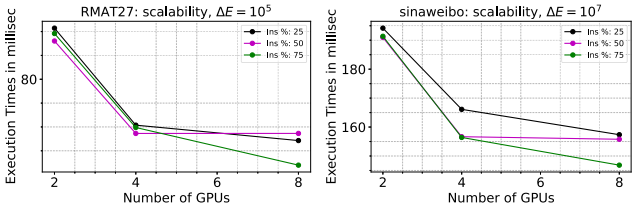
We compare the execution time of a single GPU LVCU implementation with a static graph coloring algorithm in Kokkos [18], [19], because there is no GPU-based vertex coloring implementation for dynamic networks to the best of our knowledge. For instance, the agent-based algorithm proposed in [20] cannot accept rapid changes; the incremental algorithm proposed in [21] does not mention the type of parallel architecture used and their implementation is not available. We tried to compare LVCU with coloring implementation provided by Gunrock [22] library also. However, when tested on medium-scale networks, the Gunrock implementation failed to provide any valid coloring result within a reasonable time.

As Kokkos coloring works on static graphs only, for a batch of changes $\Delta E_t = (Ins_t, Del_t)$, we supply a graph with edge set $E_{t+1} = ((E_t \cup Ins_t) \setminus Del_t)$ for Kokkos implementation.

Figure 5 presents performance results; the Y-axis shows the execution time of Kokkos-coloring divided by that of our implementation (the factor by which we are faster than Kokkos). Although the change edges are generated randomly, the results are mainly affected by two factors. First, the deletion takes more time than insertion in our algorithm. Second, for a fewer percentage of insertion (or higher percentage of deletion) the effective number of edges is reduced for Kokkos-

coloring, requiring comparatively less time to recompute. In most of the plots, LVCU performs better than Kokkos-coloring if the percentage of insertion is increased. The performance comparison shows that our algorithm can provide updated vertex color up to 20X faster than Kokkos-coloring.

3) Distributed Implementation Scalability: Figure 6 shows the strong scaling analysis of distributed LVCU. It also compares the performance for different percentages of insertion when the batch size is fixed. The results show that the execution time decreases drastically when the number of GPUs changed from 2 to 4. However, beyond 4 GPUs the performance improves very slowly. This behavior is justified as the color update part in *ProcessCE* function or visiting neighbors in *CheckConflict* function are sequential in nature and do not execute faster if the number of GPU is increased.



(a) RMAT27 (b) sinaweibo

Fig. 6: Performance with different number of GPUs

TABLE III: Color Quality Comparison ($|C|, |C^k|, |C^h|$: # of colors used by our algorithm, Kokkos-coloring, and Havoqgt respectively; Ins : % of edge insertions; ΔE in million)

(a) Single GPU

G	ΔE	Ins	$\|C\|$	$\|C^k\|$
D6	0.1	25	19	19
D6	0.1	50	19	19
D6	0.1	75	19	20
D6	1	25	47	46
D6	1	50	47	46
D6	1	75	47	47
D2	5	25	102	124
D2	5	50	102	125
D2	5	75	102	126

(b) Distributed

G	ΔE	Ins	$\|C\|$	$\|C^h\|$
D1	1	25	16	16
D1	1	50	16	16
D1	1	75	16	17
D5	10	25	102	126
D5	10	50	102	126
D5	10	75	102	127

4) Experiment on Color Quality: Table III compares the number of colors used by our algorithm and baseline algorithms. In Table IIIa, $|C|$ and $|C^k|$ are respectively the number of colors used by our single GPU implementation and Kokkos-coloring. Since no implementation exists for dynamic graph coloring for distributed GPUs, we use Havoqgt-based dynamic graph coloring implementation for distributed CPUs [2] as baseline in distributed case. Table IIIb compares the maximum number of colors used in our distributed implementation and Havoqgt. Results for LVCU and distributed LVCU show that our algorithm uses fewer or comparable colors. The total number of colors in our algorithm depends on 1) Initial coloring, and 2) Edge deletion. For edge deletion, LVCU aims to reduce the total number of colors by recoloring the endpoint with a higher color ID using the other endpoint's color.

VII. Related Work

A. Static Graph Coloring

A plethora of heuristic algorithms [9]–[11] exists for static graph coloring. For large graphs, the existing parallel vertex coloring algorithms can be broadly classified as: independent set based and speculation based. The former class finds independent subsets in the graph and colors the vertices in each subset in parallel. A Monte Carlo randomized algorithm is proposed in [23] to find a maximal independent set and color all the vertices in the set with the same color. This is improved in [24], where a random number assignment is used for every vertex to find the independent sets and color the vertices concurrently. The algorithm is also combined with Largest Degree First (LDF) heuristic in [25], which is further improved in [26]. An independent set based vertex coloring algorithm for single GPU was implemented in [27].

On the other hand, speculation based algorithms first color the vertices in parallel and iteratively resolve conflicts [13], [28]. A shared memory parallel algorithm based on the speculate and iterate approach is proposed in [29]. The GPU-based Gunrock library uses an advance-filter-compute framework [22] and provides a single GPU vertex coloring implementation. Recently, a multi-GPU implementation is developed for distributed systems [14] that modifies vertex coloring algorithms in the Kokkos-coloring framework [19] to present MPI+X implementations of distance-1 and distance-2 coloring. Specifically, this algorithm uses 2-hop ghosts to minimize the communication cost. Nevertheless, the randomized candidate selection procedure used for resolving conflict may still lead to multiple rounds of inter-partition communications.

B. Dynamic Graph Coloring

As applying static coloring algorithms to dynamic graphs can be costly for small changes, numerous methods have been proposed to update the colors for dynamic graphs.

An agent-based algorithm, is proposed in [20] for coloring dynamic graphs, where the candidate vertex between two conflicting adjacent vertices is selected for recoloring by choosing the vertex with the lowest degree of saturation. The algorithm limits recoloring the endpoints of a changed edge and their immediate neighbors. Another greedy online coloring algorithm proposed in [30] accepts vertices one by one with their corresponding edges, and colors the vertices using a color selection rule. However, this approach leads to suboptimal color quality for not adjusting the color of neighbor vertices.

To evaluate the success of an online coloring algorithm, a performance metric is introduced in [31] along with theoretical bounds. To maintain color consistency between the online approach and that of a static graph, an incremental algorithm for maintaining vertex color in dynamic graphs is proposed in [21]. However, this algorithm recolors a large number of vertices to achieve the same color output as the target static graph algorithm [9] after a single edge insertion or deletion, resulting in a relatively high overhead and low efficiency.

The authors in [2] claim to present the first generic distributed, online graph coloring algorithm. They use a hashing on global vertex ID to resolve color conflict, reducing the

initial communication cost. Although the algorithm adjusts color for only endpoints of an inserted or deleted edge, it misses the opportunity to use fewer colors because it does not adjust the color of neighbors for edge deletion.

Unlike the existing approaches, our proposed distributed LVCU algorithm follows the speculate and iterate approach in each partition and colors the affected vertices by processing the changed edges in parallel. The color conflicts are resolved iteratively. Instead of using an additional parameter such as degree, saturation degree, or hashed value of a vertex to determine which vertex to recolor, our algorithm uses the minimum available information such as vertex ID and vertex color. The LVCU algorithm maintains a proper $\delta + 1$ coloring without attempting to find an optimal solution, although it aims to reduce the number of colors used in case of edge deletion.

VIII. Conclusion

We first presented a loosely maintained vertex color update algorithm, which uses a partially stored saturation color set to recolor vertices efficiently using less memory. Next, we proposed a novel priority assignment technique followed by a distributed color update algorithm. Extensive experimental evaluations on real-world and synthetic networks show that our GPU-based implementation updates vertex color up to 20X faster than the baseline method using comparable colors.

Although producing a partial saturation color set is sequential in our current approach, it is amenable to dynamic parallelism in GPU. Furthermore, a CPU-GPU hybrid architecture can improve the performance executing heavy operations at the host and light repetitive works at the kernel. We plan to investigate these two approaches as future work.

Acknowledgment

This work was partially supported by the NSF projects SANDY (Award # OAC-1725755) and CANDY (Award # OAC-2104078).

References

[1] A. Khanda, S. Srinivasan, S. Bhowmick, B. Norris, and S. K. Das, "A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 929–940, 2022.

[2] S. Sallinen, K. Iwabuchi, S. Poudel, M. Gokhale, M. Ripeanu, and R. Pearce, "Graph colouring as a challenge problem for dynamic graph processing on distributed systems," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 347–358.

[3] A. Khanda, F. Corò, F. B. Sorbelli, C. M. Pinotti, and S. K. Das, "Efficient route selection for drone-based delivery under time-varying dynamics," in *IEEE 18th International Conference on Mobile Ad Hoc and Smart Systems (MASS)*, 2021, pp. 437–445.

[4] A. Khanda, F. Corò, and S. K. Das, "Drone-truck cooperated delivery under time varying dynamics," in *Proceedings of the 2022 Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*, 2022, pp. 24–29.

[5] S. Srinivasan, S. Pollard, S. K. Das, B. Norris, and S. Bhowmick, "A shared-memory algorithm for updating tree-based properties of large dynamic networks," *IEEE Transactions on Big Data*, vol. 8, no. 2, pp. 302–317, 2022.

[6] A. Mishra, S. Banerjee, and W. Arbaugh, "Weighted coloring based channel assignment for wlans," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 9, no. 3, pp. 19–31, 2005.

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

[8] M. R. Garey and D. S. Johnson, *Computers and intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[9] D. J. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

[10] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *Journal of the ACM (JACM)*, vol. 30, no. 3, pp. 417–427, 1983.

[11] D. Brélaz, "New methods to color the vertices of a graph," *Communications of the ACM*, vol. 22, no. 4, pp. 251–256, 1979.

[12] G. M. Slota, K. Madduri, and S. Rajamanickam, "PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks," in *IEEE International Conference on Big Data*, 2014, pp. 481–490.

[13] D. Bozdağ, A. H. Gebremedhin, F. Manne, E. G. Boman, and U. V. Catalyurek, "A framework for scalable greedy coloring on distributed-memory parallel computers," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 515–535, 2008.

[14] I. Bogle, E. G. Boman, K. Devine, S. Rajamanickam, and G. M. Slota, "Distributed memory graph coloring algorithms for multiple gpus," in *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2020, pp. 54–62.

[15] (2017). [Online]. Available: https://portal.tacc.utexas.edu/user-guides/longhorn

[16] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

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

[18] S. Rajamanickam, S. Acer, L. Berger-Vergiat, V. Dang, N. Ellingwood, E. Harvey, B. Kelley, C. R. Trott, J. Wilke, and I. Yamazaki, "Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels," *arXiv preprint arXiv:2103.11991*, 2021.

[19] (2017) Kokkos kernels. [Online]. Available: https://github.com/kokkos/kokkos-kernels

[20] D. Preuveneers and Y. Berbers, "Acodygra: an agent algorithm for coloring dynamic graphs," *Symbolic and Numeric Algorithms for Scientific Computing (September 2004)*, vol. 6, pp. 381–390, 2004.

[21] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang, "Effective and efficient dynamic graph coloring," *Proceedings of the VLDB Endowment*, vol. 11, no. 3, pp. 338–351, 2017.

[22] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "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*, 2016, pp. 1–12.

[23] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM J. on computing*, vol. 15, no. 4, pp. 1036–1053, 1986.

[24] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM J. on Scientific Computing*, vol. 14, no. 3, pp. 654–669, 1993.

[25] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *Proc. of 26th ACM symposium on Parallelism in algorithms and architectures*, 2014, pp. 166–177.

[26] G. Alabandi, E. Powers, and M. Burtscher, "Increasing the parallelism of graph coloring via shortcutting," in *Proc. of 25th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 2020, pp. 262–275.

[27] M. Osama, M. Truong, C. Yang, A. Buluç, and J. Owens, "Graph coloring on the gpu," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 231–240.

[28] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency: Practice and Experience*, vol. 12, no. 12, pp. 1131–1146, 2000.

[29] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multi-core and massively multithreaded architectures," *Parallel Computing*, vol. 38, no. 10-11, pp. 576–594, 2012.

[30] L. Ouerfelli and H. Bouziri, "Greedy algorithms for dynamic graph coloring," in *IEEE International Conference on Communications, Computing and Control Applications (CCCA)*, 2011, pp. 1–5.

[31] A. Miller, "Online graph colouring," in *Canadian Undergraduate Mathematics Conference*. Citeseer, 2004.