A Synchronization-Avoiding Distance-1 Grundy Coloring Algorithm for Power-Law Graphs

Jesun Sahariar Firoz, Marcin Zalewski, Andrew Lumsdaine
{jesun.firoz,marcin.zalewski,andrew.lumsdaine}@pnnl.gov
Pacific Northwest National Laboratory
Seattle, Washington, USA

***Abstract*—In this paper, we propose a distributed, unordered, label-correcting distance-1 Grundy (vertex) coloring algorithm, namely, Distributed Control (DC) coloring algorithm. Our algorithm eliminates the need for vertex-centric barriers and global synchronization for color refinement, relying only on atomic operations and local termination detection to update vertex color. DC proceeds optimistically, correcting the colors asynchronously as the algorithm progresses and depends on local ordering of tasks to minimize the execution of sub-optimal work. We implement our DC coloring algorithm and the well-known Jones-Plassmann algorithm and compare their performance with 4 different types of standard RMAT graphs and real-world graphs. We show that the elimination of waiting time of global and vertex-centric barriers and investing this time for local ordering leads to improved scaling for graphs with prominent power-law characteristics and densely interconnected local subgraphs.**

I. Introduction

An interesting problem in graph theory is graph coloring which partitions a set of entities into independent subsets. This problem arises in a wide range of contemporary applications, such as global climate modeling, power flows in electric grids, generating parallel code for GPU computation. Other applications of graph coloring include scheduling, sparse-matrix computation, resource allocation, pattern matching, anomaly detection etc. Designing scalable algorithms to color large-scale graphs with skewed degree distribution ("power-law" graphs) is challenging, since such uneven structure can introduce workload imbalance. In many cases, partitioning graphs to tackle workload imbalance is not useful as good separators may not exist [1], [2].

A *(distance-1) vertex-coloring* of a graph G finds an assignment of colors to every vertex v in a vertex set V such that no two adjacent vertices have the same color. The *graph-coloring* problem asks to find a vertex-coloring which uses as few colors as possible. If each vertex chooses the minimum available color, the resultant coloring is called *Grundy coloring*. The problem of finding an optimal coloring of a graph is NP-complete. However, over the course of time, many heuristic parallel *greedy* algorithms, based on Luby's [3] iterative maximal independent set computation, have been devised that perform well in practice. The most prominent of these algorithms is due to Jones and Plassmann [4].

To make a trade-off between execution time and coloring quality, greedy coloring algorithms apply different vertex ordering criteria when assigning priorities to vertices so as to decide which vertex to color first. In doing so, an implicit predecessor-successor relationship between a vertex and its neighbors is formed. This can be visualized as a *directed acyclic graph* (DAG), termed as *priority DAG* [5], with edges emanating from the predecessor(s) to the successor(s). Once the implicit DAG is created, most algorithms proceed in steps and traverse the DAG in a *DAG-synchronous* fashion: each vertex in a sub-DAG waits until all its predecessors are colored and then color itself with an available color not taken by any of its predecessors. At the level of a single vertex, this resembles Bulk-synchronous-parallel (BSP) execution model, where an algorithm iterates through computation, communication, and synchronization steps.

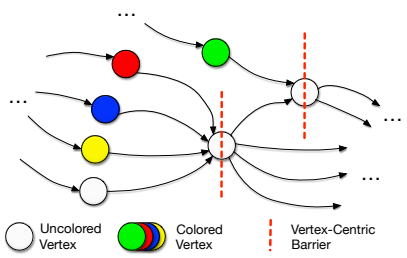


Fig. 1: Vertex-centric barriers.

In greedy coloring algorithms, waiting on a predecessor gives rise to **vertex-centric barrier/synchronization** (Fig. 1). Vertex-centric barriers induce similar ramification as global synchronization barriers in BSP approach, where the whole system must wait for a straggler before moving to the next step. A vertex with a large number of predecessors (straggler) impedes other vertices in the same level from advancing (straggler effect). Although this may not demonstrate itself as a problem in a shared-memory implementation, straggler effect can seriously limit performance of an algorithm in a distributed setting. To avoid such problems, a framework for designing synchronization-avoiding graph algorithms have been proposed in [6].

Previously, a shared-memory algorithm has been proposed [7] that relaxes the constraint of waiting on the predecessors. This algorithm iterates through two steps till convergence. In the first step, based on the local information available during the current iteration, a vertex obtains a color even though all its predecessors have not obtained colors (*speculation step*). After a synchronization barrier, in the second step, the algorithm iterates through all the vertices to fix colors when necessary (*refinement step*). Although this approach relaxes the constraint of waiting on all the predecessors to obtain a color, it fails

to completely eliminate the need for synchronization. In this paper, we show that, instead of splitting vertex-coloring into two steps of speculation and refinement, we can further relax the criteria and don't need to impose any barriers in between coloring and refinement steps.

Based on this observation, we propose a new *label-correcting, unordered, distributed* algorithm for graph coloring, namely Distributed Control (DC) coloring algorithm. An unordered algorithm such as DC does not require a particular execution order and allows independent computations to execute concurrently. Because dependencies are not checked beforehand, computation is performed optimistically, and the results computed previously may need to be corrected (label correction).

These *label-correcting* mechanisms have the advantage of avoiding the straggler effect. Most importantly, in our algorithm, vertex color updates happen based on atomic operations and local termination detection. There is no wasted time involved in waiting on a vertex-centric or global barrier before updating a vertex color. Hence, our algorithm benefits from *optimistic parallelism* [8] by making progress completely asynchronously.

However, if sub-optimal results are calculated in intermediate steps and require updates too often due to speculation, then unordered graph algorithms can perform poorly. To circumvent this problem, the algorithm employs local ordering of tasks with thread-local priority queues. Additionally, we implement application-level message caching and message reduction to handle the propagation of redundant messages from non-monotonic vertex color update function (any color within a range can become available). The combination of these two techniques assists the algorithm in tackling work explosion and minimizes unnecessary updates. We implement our vertex-coloring algorithm in the AM++ [9] asynchronous many-task (AMT) runtime since it supports fine-grained communication and computation, based on active messages. We demonstrate that our algorithm is specially suitable for graphs with dominating power-law degree distribution characteristics and for graphs which need to be executed in large distributed environment. Most of the vertices in such graphs have low degrees but few other vertices, termed as *hubs*, have very high degrees. These high-degree vertices can hamper scalability. Additionally, graphs that have strong power-law distributions also contain densely interconnected subgraphs. Our proposed algorithm demonstrates better scalability with such graphs by exposing latent parallelism. We show that our DC coloring algorithm performs better than the well-known Jones-Plassmann algorithm in AM++ in such cases.

This paper makes the following contributions:

- We introduce an unordered, label-correcting, distributed Grundy coloring algorithm, namely *Distributed Control*, which avoids the need for global and vertex-centric barriers and utilizes the benefits of optimistic (speculative) parallelism.
- We conduct experiments with DC and Jones-Plassmann algorithms with synthetic and real-world graphs that have different degree distributions and clustering coefficients. We conclude that DC based coloring algorithm is suitable for graphs with dominating power-law characteristics and densely interconnected subgraphs. We also report coloring qualities of our implemented algorithms.
- We compare the performance of our vertex-coloring algorithms in AM++ with another implementation in the well-known graph application framework, PowerGraph [10], and report better performance of DC at larger scale.

II. Background

In this section we discuss the baseline Jones-Plassmann algorithm for coloring. Both Jones-Plassmann and our Distributed Control coloring algorithm find a Grundy coloring of a graph G. A vertex v is called a *Grundy vertex* if v is colored with the smallest color not taken by any neighbor. A *Grundy coloring* of G is one in which every vertex is a Grundy vertex. A vertex v is called *properly colored* if for all $i \in neighbor(v), color(i) \neq color(v)$. Grundy coloring is a proper coloring of a graph. The minimum number of colors (*color classes* or *independent subsets*) needed to properly color a graph G is called the *chromatic number* of G, $\chi(G)$ and is an NP-hard problem. Grundy coloring always results in k colors where $\chi(G) \leq k \leq (\Delta+1)$ for graph G with maximum degree of Δ. Our algorithms find a Grundy coloring and use at most $(\Delta + 1)$ colors. Empirical results (Sec. V) show that the maximum no. of colors needed is significantly smaller than Δ.

Different vertex ordering heuristics [11] can be employed to decide which vertex to color first in Jones-Plassmann algorithm and our Distributed Control coloring (Sec. III). For example, the first-fit heuristic [12] colors vertices in the order they appear in the input graph representation. The random ordering heuristic [4] colors vertices in a uniformly random order. The incidence-degree ordering heuristic [13] iteratively colors an uncolored vertex with the largest number of colored neighbors. The saturation-degree ordering heuristic [14] iteratively colors an uncolored vertex whose colored neighbors use the largest number of distinct colors. For both Jones-Plassmann and our Distributed Control coloring(Sec. III) algorithms, we use random ordering heuristic to assign order to vertices for coloring. The resultant ordering is used to decide which vertex to color first. Imposing an ordering, in essence, creates a predecessor-successor relationship between vertices. In the following discussion, we refer to vertices with no predecessor as *roots*.

A. Jones-Plassmann Coloring Algorithm

Jones-Plassmann (JP) algorithm works as follows: each vertex maintains a list of colors taken by the predecessors to keep track of how many predecessors have been colored so far. The algorithm starts by assigning color 0 to the roots and sending out the information to all their successors. When a vertex v finishes obtaining all predecessors' colors, it starts searching for an available color. When it finds an available minimal color value, it assigns the color to itself. If all the colors in the range $0, 1, 2, \cdots, predecessorCount[v]-1$ are taken, the vertex assign $predecessorCount[v]$ as its color. Once colored, the vertex sends its color information to all its successors.

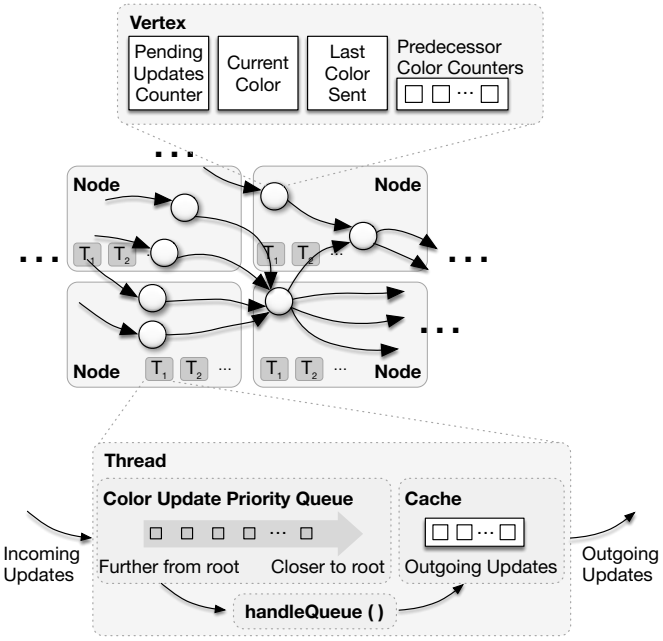


Fig. 2: Overview of the algorithm. The graph is distributed over nodes and processed by threads. Threads receive color updates through active messages, store them in thread-local priority queues, process them, and then send new updates out through a thread-local cache. Every vertex is associated with a count of pending updates for that vertex, its current color, last color update sent, and an array to keep track of predecessor colors. All of the vertex information is shared between threads and updated concurrently.

Distributed Jones-Plassman implements synchronization on vertices with a counter per vertex. As the predecessor information is received, the per-vertex counter is incremented. Whenever the counter value is equal to the predecessor count for a vertex, the termination of the vertex is triggered, and the color for the vertex is calculated. The termination of the algorithm is triggered by global termination detection.

We have chosen Jones-Plassmann Coloring algorithm as our baseline algorithm because it has been proven to be efficient in distributed setting [15], both in terms of execution time and optimality of the result. This algorithm is based on asynchronous push mechanism, where vertices push their states to the successors.

III. Distributed Control Coloring Algorithm

In this section, we present our Distributed Control (*DC*) vertex-coloring algorithm. First we give a brief overview of the algorithm. Next, we divide our discussion by the kind of issues we address, and we divide the algorithm into fragments relevant to addressing such issues.

Overview of the main idea: optimistic execution with work-optimization. Our objective is to design a synchronization-avoiding distributed graph coloring algorithm by eliminating vertex-centric barriers and global synchronization. To do so, the algorithm proceeds with coloring vertices speculatively, and it fixes sub-optimal colors as it

Algorithm 1: Distributed Control coloring algorithm

```
In  : Graph G = ⟨V, E⟩,
      ∀ v ∈ V: owner[v] = rank that owns v
1  procedure Main()
2      initialize()
3      active message epoch
4          parallel foreach v ∈ V do
5              if owner[v] = this rank then
6                  Visit-root(v)
7          handleQueue()
8  procedure Visit-root(Vertex r)
9      if predecessorCount[r] = 0 then
10         color[r] ← 0
11         oldColor ← INVALID_COLOR
12         distance[r] ← 0
13         parallel foreach neighbor v of r do
14             send Visit(v, oldColor, color[r], distance[r])
               to owner(v)
15             activeCount++
16 message handler Visit(Vertex v, Color oldColor, Color newcolor,
     Distance d)
17     tempDistance ← d + 1
18     taskRemainingInLocalColorQs[v]++
19     colorQ[t_id].push(v, oldColor, newColor, tempDistance)
```

Procedure 1: initialize

```
1  foreach thread t_id do
2      Allocate memory for colorQ[t_id];
3      Allocate memory for cacheQ[t_id];
4  foreach v ∈ V do
5      taskRemainingInLocalColorQs[v] ← 0;
6  foreach v ∈ V do
7      if predecessorCount[v] > 0 then
8          Allocate memory for predecessorColors[v] based on
             predecessorCount[v];
9  foreach v ∈ V do
10     color[v] ← INVALID_COLOR;
11 foreach v ∈ V do
12     Allocate memory for lastColorSent[v] ;
```

progresses and better color information from the predecessors becomes available. The basic process, illustrated in Fig. 2, is to optimistically pick the color for a vertex based on the partial information available about its predecessors, which is maintained in an array of predecessor color counters for every vertex. Once a new color is picked, the color change is propagated to all successors, and the process repeats until quiescence. Since colors are fixed when suboptimal updates occur, our algorithm is a *label-correcting* coloring algorithm.

Speculative coloring removes synchronization, but it can lead to suboptimal work where most color updates end up being erased by subsequent fixes. To avoid such work explosion, we introduce mechanisms that favor profitable work: sender-side caching of color updates, receiver-side prioritization of incoming color updates, and local per-vertex termination detection.

Per-Vertex termination detection. Our algorithm maintains a *per-vertex termination counter* (pending updates counter in Fig. 2 and $taskRemainingInLocalColorQs$). For each vertex, this counter keeps track of how many predecessor color updates have been received so far and are waiting to be processed *locally*. Since local work is cheaper than remote updates, a vertex will not send any color updates to its successors until the termination counter reaches the value of 0, indicating that no more local work is available.

Execution-time message ordering via priority queues. As discussed in Sec. I, a greedy coloring algorithm imposes ordering on vertices based on different heuristics such as vertex id, degree etc. to decide which vertex to color first (thus creating predecessor-successor relationships among vertices). This ordering creates a priority DAG with edges emanating from the predecessors to the successors. Because Distributed Control traverses the DAG optimistically (in contrast to Jones-Plassmann) and vertices can be processed in arbitrary order, sub-optimal color updates can trigger work explosion. To minimize such redundant updates, we assign *priorities* to received predecessor messages. We observe that vertices that are relatively close to the roots in the DAG (distance metric) and that have relatively small color values (color metric) should be processed first. The distance metric prioritizes work that is closest to DAG roots and thus most likely to be final, and the color metric favors colors that are more likely to be final colors in Grundy coloring (smaller color wins).

DC vertex-coloring (Proc. 1) maintains two types of priority queues per thread t_{id} on each rank: $colorQ$ and $cacheQ$ (Lns. 2–3 in Proc. 1). The per-thread $colorQ$s orders received messages by the distance and the color metrics. The other priority queue, $cacheQ$, is used to order outgoing messages. We will discuss message caching and reduction later in this section.

Algorithm

The main structure of the algorithm is shown in Alg. 1. The main procedure consists of an initialization step followed by an active message epoch in which initial messages are sent and then handled in the $handleQueue()$ procedure that processes thread-local update queues (described later). The initial messages are sent from roots of the DAG by the $Visit\text{-}root()$ procedure, which first checks if a vertex is a root, and if it is it initializes the color to 0 and sends an update to the root's neighbors. The messages are received by the $Visit$ message handler, and it forwards the incoming color updates to thread-local queues that are then processed in $handleQueue()$. This processing results in color updates and in new messages sent to the $Visit$ handler. These updates continue until there is no more work to be done and the active message epoch terminates. Next, we describe the details of each of these steps.

Initialization. At the beginning, the algorithm initializes the per-vertex termination counters, $taskRemainingInLocalColorQs[v]$ (Ln. 5 in Proc. 1). As discussed earlier, this counter plays the vital role in the algorithm to achieve optimistic parallelism. For a particular vertex v, the $taskRemainingInLocalColorQs[v]$ counter keeps track of how many messages are waiting to be processed in the $colorQ$ priority queues. Whenever $taskRemainingInLocalColorQs[v]$ reaches a value of zero, it triggers an asynchronous color update for v. Each vertex v with non-zero predecessor count maintains a list of colors, $predecessorColors[v]$, that have been taken by its predecessors (Ln. 8 in Proc. 1). When a vertex optimistically acquires a color, it does so by searching through this list to find the minimum color not taken by any predecessor.

Visiting Roots. Alg. 1 starts (Ln. 6) by assigning a color value of zero to all the roots and propagating the color information to the roots' successors asynchronously (Lns. 10–14 in Alg. 1). Each neighbor vertex v, on receiving the message containing its predecessor's color information, executes a message handler $Visit$ (Line 16 in Alg. 1). The handler inserts the received predecessor information in the $colorQ$ priority queue (Ln. 19). Before insertion, we also increment the corresponding local termination detector for v, $taskRemainingInLocalColorQs[v]$, by one (Ln. 18 in Alg. 1).

Procedure 2: handleQueue

```
1  localIter ← threshold;
2  while true do
3      if cacheQ[t_id] not empty and
        (cacheQ[t_id].size < threshold) then
4          v, predOldCol, predNewCol ← cacheQ[t_id].pop() ;
5          tryReductionAndSend(v, predOldCol,
             predNewCol);
6      else
7          localIter--;
8      if localIter = 0 then
9          v, predOldCol, predNewCol ← cacheQ[t_id].pop() ;
10         tryReductionAndSend(v, predOldCol,
             predNewCol);
11         localIter ← threshold;
12     if colorQ[t_id] not empty then
13         v, predOldCol, predNewCol ← colorQ[t_id].pop() ;
14         taskRemainingInLocalColorQs[v]--;
15         colorVertex(v, predOldCol, predNewCol) ;
16         finishCount++;
```

Procedure 3: colorVertex

```
   In : Vertex v, Color predOldColor, Color predNewColor
1  if predOldColor < predecessorCount[v] then
2      predecessorColors[v][predOldColor]-- ;
3  if predNewColor < predecessorCount[v] then
4      predecessorColors[v][predNewColor]++ ;
5  if taskRemainingInLocalColorQs[v] = 0 then
6      newColor ← findMinAvailableColor(v);
7      while true do
8          oldColor ← color[v];
9          if atomicCompareAndSwap(color[v], oldColor,
             newColor) then
10             cacheQ[t_id].push(v, oldColor, newColor);
11             break
```

Processing elements from the priority queues. To process the received predecessor color updates pushed in the $colorQ$,

each thread on each rank executes $handleQueue$ (Proc. 2) until termination is reached (Ln. 7 in Alg. 1). Since our algorithm is an unordered algorithm, it proceeds optimistically as much as possible. But in doing so, the algorithm runs into the risk of executing excessive sub-optimal work. To circumvent this problem, the runtime, at this point, needs to decide which task to execute first. Hence, we choose the best possible locally-available candidate for processing (Ln. 13 in Proc. 2), based on the distance and color metrics we discussed earlier, from the $colorQ$ priority queue. In this way, our algorithm also eliminates some sub-optimal work. Once a vertex v is popped from the priority queue, the $taskRemainingInLocalColorQs[v]$ is decremented (Ln. 14 in Proc. 2) and the algorithm executes $colorVertex$ (Proc. 3).

Coloring Procedure. (Proc. 3) Each message from a predecessor of a vertex v, inserted into the $colorQ$, contains two pieces of information: predecessor's old color and the new color. After popping a message from the queue, the relevant color counters for v containing v's predecessor's color information, $predecessorColors[v]$ are decremented (Ln. 2) and incremented (Ln. 4 in Proc. 3) for the predecessor's old color and new color respectively. Next, a check is performed to see whether the $taskRemainingInLocalColorQs[v]$ has reached a value of zero (Ln. 5), which can trigger a search-and-update for a new color value for v (Ln. 9).

Finding minimum available color. (Proc. 4) Finding the minimum color starts by saving the the current color of vertex v and setting the $availableColor$ to $predecessorCount$ (Ln. 3). Next a search is performed in $predcolor[v]$ to find a minimum color not taken by any predecessor. If such color is found, the vertex color is set to new minimum available color.

Procedure 4: findMinAvailableColor

```
   In  : Vertex v
1  oldColor ← color[v];
2  if atomicCompareAndSwap(color[v], oldColor,
     predecessorCount[v]) then
3      availableColor ← predecessorCount[v];
4      for (i = 0; i < predecessorCount[v]; i++) do
5          if predecessorColors[v][i] == 0 then
6              availableColor ← i;
7              break
8      return availableColor
9  return oldColor;
```

Message Caching. In order to reduce the propagation of sub-optimal work, we cache messages destined for the successors before sending them (Ln. 10 in Proc. 3). For this purpose, we have implemented a customized reduction cache with thread local priority queues, $cacheQ$ (cache in Fig. 2). At the beginning of the $handleQueue$ function (Lns. 3–5 in Proc. 2), before processing any element from the $colorQ$s, the algorithm attempts to send messages to the successors that have been cached in the $cacheQ$s. When the color information of a vertex is popped from the $cacheQ$ (Ln. 4 in Proc. 2), a check is performed to see whether the vertex is already updated with a better color or whether the color has not been changed since last update (Ln. 2 in Proc. 5). If both of the conditions fail, the current vertex color is recorded as the last color sent (Ln. 5 in Proc. 5) and the updated color information is propagated to its successors (Ln. 11 in Proc. 5).

Procedure 5: tryReductionAndSend

```
   In  : Vertex v, Color oldColor, Color newColor
1  lastColorSentV ← lastColorSent[v] ;
2  if lastColorSentV == color[v] then
3      return;
4  while true do
5      if atomicCompareAndSwap(lastColorSent[v],
         lastColorSentV, newColor) then
6          break
7      else
8          lastColorSentV ← lastColorSent[v] ;
9  parallel foreach neighbor w of v do
10     if id[w] > id[v] then
11         send
           Visit(w, lastColorSentV, color[v], distance[v])
           to owner(w);
12         activeCount++;
```

Note that there is no global synchronization barrier or vertex-centric barrier in our algorithm.

Termination. Termination detection is a part of the underlying runtime. Since updates are propagated along the edges of the DAG, at some point no changes are received from the upstream in the DAG, that guarantees termination in conjunction with the termination detection algorithm in the runtime. The runtime termination detection algorithm is based on Sinha-kale-Ramkumar algorithm [16]. Two counters are maintained locally on each compute node: $activeCount$ and $finishCount$. Each time a vertex sends updates to its successors, the local $activeCount$ is incremented (Ln. 15 in Alg. 1 or Ln. 12 in Proc. 5). When a received message from the predecessor is processed from a thread-local priority queue, the local $finishCount$ is incremented by one (Ln. 16 in Proc. 2). Non-blocking global reduction (all-reduce) on these counters accumulates the counts and ensures that the algorithm terminates when their difference is observed to be equal to zero for two consecutive times. The non-blocking reductions are joined only by nodes that have no local tasks (thus they occur rarely).

Note that these counters serve different purpose from $taskRemainingInLocalColorQs[v]$. The latter is the local counter associated with a vertex that triggers a color search as it reaches a value of zero, indicating that no message from predecessor is left to process at a particular instant of time in the thread-local priority queues on the current rank for the current vertex.

Leveraging Approximate Vertex Ordering heuristics as priorities. We have also conducted experiments with incidence-degree (ID) ordering (choosing a vertex with largest number of colored neighbors) and saturation-degree (SD) ordering (choosing a vertex whose colored neighbors need the largest number of distinct colors). In distributed setting, applying these

ordering heuristics in their original forms are overly restrictive. Instead, we apply local approximations of these heuristics to prioritize incoming messages in DC. When applying ID heuristic, we leverage per-vertex termination counter to track how many predecessors have been colored for each vertex and use this information to prioritize messages in such a way that the target vertex with larger per-vertex termination counter value will be given priority while processing. With SD heuristic, whenever a message with the old and the new color information of a predecessor is processed, the corresponding color counters are checked: a check is performed to see whether the 'old' color is becoming available (by transitioning to value 0) or whether the 'new' color is taken for the first time by any neighbor (by transitioning from 0 to 1) respectively. Keeping track of these changes for each vertex with a counter provides us with an approximation of how many distinct colors have been taken by the neighbors of each vertex. This counter can be used to prioritize incoming messages to be processed. When applying ID and SD heuristic separately, we do not employ distance-based priority discussed earlier.

IV. Correctness

In this section, we prove the correctness and the termination guarantee of Distributed Control coloring algorithm.

A. Correctness

Lemma 1. *Distributed Control coloring algorithm eventually converges.*

Proof. Convergence of DC can be proved by structural induction on the levels of the DAG traversed during the algorithm execution. In the base step, roots are on level 0 and since roots have no predecessors, all the root vertices will settle immediately with color 0. Now as an inductive hypothesis, let us assume that all the vertices in the DAG up to level $l-1$ have settled with final colors. Consider the inductive step. All the vertices on level l of the DAG will receive final colors of their predecessors at level $l-1$. Since messages only propagate from predecessors to successors, each vertex will eventually receive all final colors of its predecessor(s) and will settle. As messages propagate in one direction along the edges of the DAG, no cycle forms and vertices at level l will also settle with a final color. Thus DC will eventually converge. □

Lemma 2. *At the convergence step, each vertex (except roots) processes one final message from a predecessor containing its old color and new color.*

Proof. Since roots obtain a color of 0 at the beginning of DC, they only propagate that new color information to their successors. Successors of roots, on receipt of such messages only increment the color counter corresponding to new color (0) (there won't be a matching decrement counter operation for leaving the old color for roots). The predecessors of all other vertices generate a sequence of updates, each message containing the old color as well as the new color. Even if these messages arrive at the successors out-of-order, every decrement to a counter that corresponds to leaving an old color by the predecessor will be matched with an increment of the new color counter at the receiver (successor) end (Lns. 2–4 in Proc. 3). At the receiver vertex v, if the predecessor old color value or the new color value is greater than $predecessorCount[v]$, the value is dropped (since there is no need for keeping track of colors greater than $predecessorCount[v]$). However, if the received value is within the range of 0 and $predecessorCount[v]$, the appropriate counter is adjusted. Convergence of a vertex happens when the last update from the last predecessor is received. At convergence, a vertex has received updates from all its predecessors and all the counter increments have been accounted for by the corresponding decrements except for one. □

Corollary 1. *Processing the last message on convergence will set the local termination detection counter of a vertex to zero, resulting in a final color search.*

Proof. Processing the last update for a vertex from a thread-local priority queue will set the local per-vertex termination counter to zero, thus triggering a color search (Lns. 5–6 in Proc. 3). □

Theorem 1. *Distributed control coloring algorithm converges with correct (Grundy) coloring.*

Proof. To prove the correctness, we show that the final search (as in Proc. 4) for each vertex will ensure the vertex color is correct. Before each search starts (including the final search), the current vertex v's color is saved and then temporarily set to $predecessorCount[v]$. The final search proceeds by scanning through the predecessor color counters to find the minimum one not taken by any the predecessors. After the search is finished, a color update operation is attempted with the new-found minimum color. Since the color was set to $predecessorCount[v]$ before starting the search, the final search ensures that minimum available color will be stored as the final color of a vertex. □

V. Experimental Results

A. Experimental Setup

1) Dataset: We evaluate the performance of Distributed Control and Jones-Plassmann vertex coloring algorithms with synthetic RMAT graphs and real-world graph inputs.

Characteristics of RMAT Graphs. To generate synthetic graphs, we employ the RMAT graph generator [17]. In this paper, we experiment with 4 types of RMAT graphs (Table I): Graph500 [18], Erdős-Rényi (RMAT-ER), RMAT-$\widetilde{G}$ (Good), and RMAT-B (Bad). These graphs differ in degree distribution of vertices and in the density of local subgraphs. A detailed description of the RMAT graph generator and different RMAT graph characteristics can be found in [19]. Both RMAT-$\widetilde{G}$ and RMAT-B contain relatively dense "subcommunities" (dense local subgraphs). The degree distribution of RMAT-ER follows a normal distribution and contains only one local maximum. However, the degree distributions of Graph500, RMAT-B and RMAT-$\widetilde{G}$ are similar and contain several local maxima. The degree distribution characteristics and the local subcommunity

Graph type	a	b	c	d
Graph500	0.57	0.19	0.19	0.05
RMAT-ER	0.25	0.25	0.25	0.25
RMAT-$\widetilde{G}$	0.45	0.25	0.15	0.15
RMAT-B	0.55	0.15	0.15	0.15

TABLE I: Parameters for RMAT generator

Graph type	graph	$\|V\|$	$\|E\|$	d_{avg}	d_{max}	$\tilde{D}$
Social network	Twitter	44M	2.9B	37	750k	36
	Friendster	65M	3.6B	55	5k	32
Webgraph	sk2005	50M	3.8B	38	8M	17
Road network	europe_osm	50M	1B	2	13	$\sim$ 7000

TABLE II: Real-world input graph characteristics. Total number of vertices ($|V|$) and edges ($|E|$) along with the average degree (d_{avg}), maximum degree (d_{max}) and diameter ($\tilde{D}$) for each type of graph input is tabulated here.

structures differentiate these 3 types of RMAT graphs from RMAT-ER. A measure of centrality, local clustering coefficient, which measures how close the neighbors are in forming a clique, also varies significantly for these 3 types of graphs. RMAT-B contains dense local subgraphs. Same property also holds for RMAT-$\widetilde{G}$. Taking all these properties into consideration will help us understand the performance results later in this section. In the constructed synthetic graphs, each vertex of the RMAT graphs has an average degree of 16 (directed). We remove duplicate edges and self-loops from the generated graph. In the plots for our scaling experiments, X axes have a one-to-one correspondence and indicate the scale of the input graph and the corresponding number of compute nodes employed in each experiment. A graph of scale x denotes a graph with 2^x vertices. Each vertex in RMAT graph has an average degree of 16 (directed), for a total of $2 * 16 * 2^x$ edges, considering undirected edges.

Real-world Datasets. For large scale experiments in AM++, Table II summarizes the characteristics of the real-world graph datasets we have experimented with. Twitter and Friendster datasets are generated by crawling two online social networks and are obtained from Laboratory of Web Algorithmics [20] and Stanford Large Network Dataset Collection (SNAP) [21] respectively. Sk2005 is generated by crawling the (.sk) domain for Slovakian researchers in 2005 using UbiCrawler. europe_osm dataset represents European road network extracted from Open Street Map. These two datasets were downloaded from the University of Florida Sparse Matrix Collection [22]. We also compare the performance of our algorithm with PowerGraph on real-world dataset (Table III) collected from [21].

2) Configuration:

a) Hardware: We have conducted our experiments on a 512-node Cray XC30 system. Each compute node on the XC30 system consists of two Intel Xeon E5 12-core x86_64 2.3 GHz CPUs with hyperthreading enabled (up to 48 hardware threads per node) and of 64 GB of DDR3 RAM. All XC30 compute nodes are connected through the Cray Aries interconnect.

b) Compiler Options: We have compiled our code with gcc 7.2.0 and with optimization level '-O3'. Additionally, single node experiments were performed with networking turned on.

c) Graph Distribution: The graph is distributed across compute nodes in a disributed compressed sparse row (CSR) data structure where each node stores a local CSR representation of the local portion of vertices assigned to it (1-D distribution). The vertices are distributed block-cyclically.

B. Scalability Results with RMAT Graphs

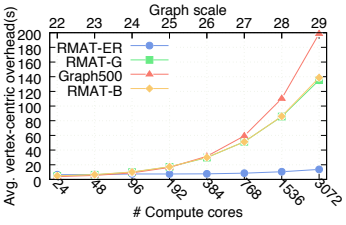
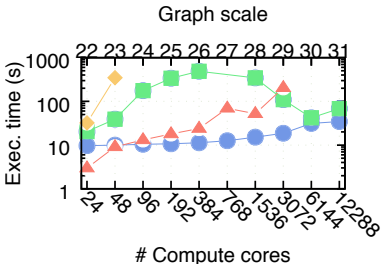
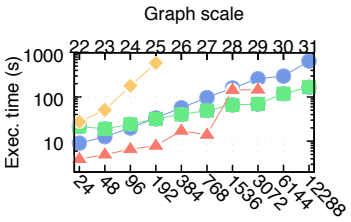


Fig. 4: Average barrier overhead of Jones-Plassmann algorithm.

1) Weak Scaling Results: For weak scaling experiments, we double the number of compute nodes as we double the number of vertices. Figure 3 shows the weak scaling results with different RMAT graphs. As can be seen from the figure, with RMAT-$\widetilde{G}$ (Fig. 3b), Graph500 (Fig. 3c), and RMAT-B (Fig. 3d) graph inputs, Distributed Control coloring algorithm is 2x-3.5x times faster at scale in comparison to Jones-Plassmann algorithm. Although, with RMAT-ER graph input, Distributed Control does not perform well with smaller node count, with larger node count, *DC* performs comparably with Jones-Plassmann algorithm. The color qualities have also been tabulated in each case. With increasing power-law characteristics (Table I), the number of colors required also increases. However, both JP and DC coloring algorithms achieve the same coloring quality. The requirement for more colors with the increase of power-law characteristics can be attributed to the dense local subgraphs (subcommunities). Larger parameter values for a and d, in comparison to b and c values, generate subcommunities within the graph structures. The vertices within these local subgraphs are highly connected and forms almost cliques. The larger and wider range of values for clustering coefficient of RMAT-$\widetilde{G}$ and RMAT-B [19] graphs also validate the existence of dense subcommunities. With a pre-execution ordering heuristic for vertex-coloring, each vertex in such a dense local subgraph has to wait for its predecessors before obtaining a color. As a result, with a larger number of dense local subgraphs, parallelization in Jones-Plassmann becomes limited. On the other hand, *DC* can proceed optimistically, without waiting on a vertex-centric or global barrier. Such execution behavior helps *DC* achieve better performance for Graph500, RMAT-$\widetilde{G}$ and RMAT-B graph inputs at scale. With RMAT-ER graph, however, *DC* can suffer from performance bottleneck if too many sub-optimal updates are performed. This is evident from the workload characteristics of *DC* with RMAT-ER, shown in Fig. 7. *DC*, in this case, is unable to successfully filter out sub-optimal work and suffers from extra work execution. In particular, with smaller compute node count, *DC* suffers from performance bottleneck due to the overhead encountered by execution time ordering and frequent conflicts that arise from optimistic color update. Frequent

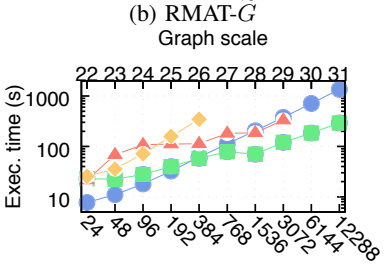


(a) RMAT-ER (color count 16)



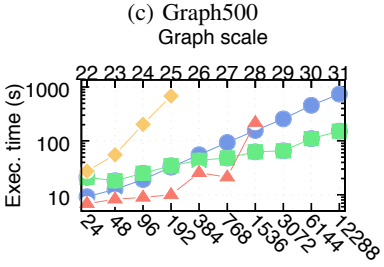
Scale	22	23	24	25	26	27	28	29	30	31
AM++	90	97	106	117	129	140	154	169	185	204
PowerGraph	99	111	122	136	148	164	240	214		

(b) RMAT-$\widetilde{G}$



Scale	22	23	24	25	26	27	28	29	30	31
AM++	347	434	530	636	780	955	1159	1413	1714	2074
PowerGraph	408	614	776	946	1157	1579	1816	2232		

(c) Graph500



Scale	22	23	24	25	26	27	28	29	30	31
AM++	169	192	223	256	295	342	399	454	526	612
PowerGraph	207	242	280	330	395	383	617			

(d) RMAT-B

Fig. 3: Weak Scaling results with different RMAT graphs on Cray XC30 system. Power-law characteristics of the input graphs increases from top to bottom. Color qualities in each case are tabulated under each plot. Algorithms in AM++ achieve the same color qualities. Missing datapoints are due to out-of-memory.

color update makes *DC* a compute-intensive algorithm, rather than a communication-bound algorithm. However, the situation reverses at scale and *DC* catches up with Jones-Plassmann at scale 30. We will discuss more about workload characteristics of the two algorithms in Sec. V-E.

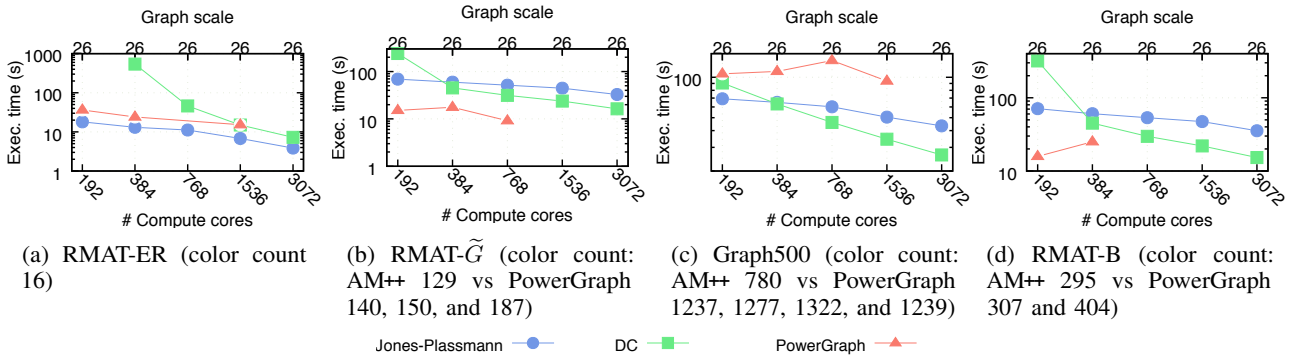
2) Overhead of Vertex-centric Barriers: We measure the overhead of vertex-centric barriers in Jones-Plassmann algorithm as follows: for each vertex, we record the time when the first message from a predecessor is received. Next, we record the time of the receipt of last predecessor update. The difference between these two quantities gives a measure of how much time each vertex wait on a barrier. We compute the average of such time with different RMAT inputs and scales and report the result in Fig. 4. RMAT-ER encounters the smallest barrier overhead that remains constant as the scale of the graph increases. In contrast, other graph inputs have larger barrier overheads at larger scale and present DC with better opportunity to perform by speculation.

3) Effect of Caching: We also show scaling results of *DC* with caching and priority heuristic disabled (DC_{nc_nd}) in Fig. 3. As can be seen from the figure, even at small scale, DC_{nc_nd} does not perform well due to work explosion resulting from aggressive speculation. At or beyond 384 compute cores, the amount of network traffic generated by DC_{nc_nd} causes node failures due to memory exhaustion.

4) Strong Scaling Results: For strong scaling experiments, we double the number of compute nodes and keep the graph size constant. Figure 5 reports strong scaling performance of Distributed Control and Jones-Plassmann algorithms with various RMAT graphs. We also report the color count in each case. As can be seen from Figs. 5b to 5d, at or beyond 384 compute cores, *DC* executes faster than Jones-Plassmann. *DC* also achieves comparable performance with RMAT-ER graph when run on larger number of compute nodes. As we increase the number of node counts, Jones-Plassmann algorithm struggles to scale, since vertex-centric barrier becomes an issue and communication overhead across large number of compute nodes starts affecting its performance. *DC*, on the other hand, enjoys the opportunity of optimistic parallelism with larger resource count. With enough processing units at its disposal, *DC* can support continuous color updates and saturates the computing resources with work. In this way, even though *DC* has to execute more work compared to JP (Sec. V-E), investing the time obtained by eliminating vertex-centric barrier, results in better performance.

C. Experiments With Real-World Datasets

We evaluate coloring algorithms on AM++ with four larger real-world datasets: Friendster, Twitter, sk2005 and europe_osm. These datasets represent social network, follower network, webcrawl graphs and road networks respectively. With Friendster (Fig. 6a) and sk-2005 input (Fig. 6c), *DC* has better scalability compared to JP. In both cases, increasing the number of compute nodes penalizes JP with communication cost as well as vertex-centric barriers. Note that sk2005 requires a large number of colors compared to other input graphs. sk2005



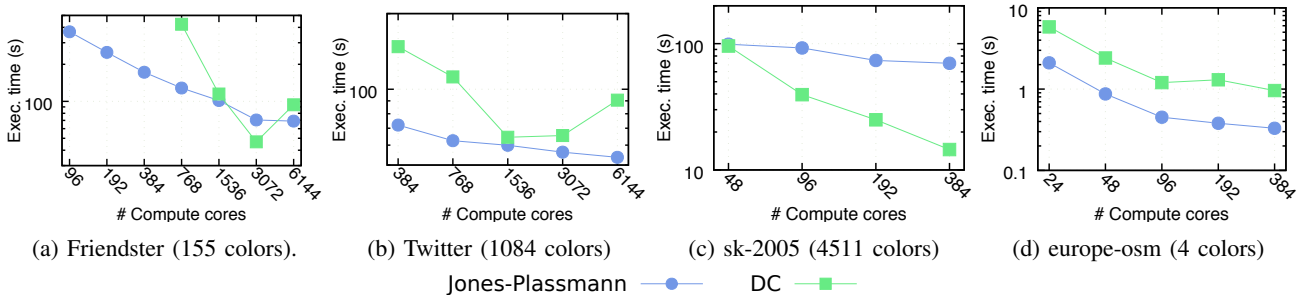
(a) RMAT-ER (color count 16)

(b) RMAT-$\widetilde{G}$ (color count: AM++ 129 vs PowerGraph 140, 150, and 187)

(c) Graph500 (color count: AM++ 780 vs PowerGraph 1237, 1277, 1322, and 1239)

(d) RMAT-B (color count: AM++ 295 vs PowerGraph 307 and 404)

Fig. 5: Strong Scaling results with RMAT synthetic graphs, with degree distribution skew increasing from left to right. RMAT-B has the most skewed degree distribution that provide DC with better opportunity for speculative execution with increased computing resources.



(a) Friendster (155 colors).

(b) Twitter (1084 colors)

(c) sk-2005 (4511 colors)

(d) europe-osm (4 colors)

Fig. 6: Strong scaling results with real-world graph inputs. In each case, we start with the smallest no. of cores that can fit the whole dataset in memory. With DC, Friendster dataset generates more messages during algorithm execution and run out-of-memory with smaller no. cores.

Graphtype	graph	$\|V\|$	$\|E\|$	$\tilde{D}$	cf	$S_{coloring\_pg}$	colorcount
Communication networks	wiki-Talk	2.4M	5M	9	0.0526	1.93	79
	email-EuAll	265k	420k	14	0.0671	4.15	30
Social networks	soc-LiveJ.	4.8M	69M	16	0.2742	1.87	324
	com-orkut	3M	117M	9	0.1666	0.83	115
	com-lj	4M	34M	17	0.28	1.3	333
	com-youtube	1.1M	2.9M	20	0.0808	2.04	38
	com-dblp	317k	1M	21	0.6324	3.43	113
	com-amazon	334k	925k	44	0.3967	1.68	9
Purchase network	amazon0601	403k	3.3M	21	0.4177	1.4	12
Road networks	roadNet-CA	1.9M	5.5M	849	0.0464	22	4
	roadNet-TX	1.3M	3.8M	1054	0.0470	24	4
	roadNet-PA	1M	3M	786	0.0465	19	4
Citation graphs	cit-Patents	3.7M	16.5M	22	0.0757	1.2	14
Web graphs	Web-Google	875k	5.1M	21	0.5143	1.5	43
	Web-BerkStan	685k	7.6M	514	0.5967	6.3	201
	Web-Stanford	281k	2.3M	674	0.5976	2.1	63

TABLE III: Speedup results of DC over PowerGraph ($S_{coloring\_pg}$) with real-world input graphs. Total number of vertices ($|V|$) and edges ($|E|$) along with the diameter ($\tilde{D}$), average clustering coefficient (cf) for each type of graph input is tabulated here.

.

represents a collaboration network of Slovakian researchers with densely connected (hence larger color count). Web-crawl graphs (such as sk2005) have two important topological characteristics: they have low diameters ("small world") and their degree distribution follow power-law ("scale-free"). Road network graphs such as europe_osm, on the other hand, has bounded degree distribution, a smaller average and maximum degree count (cf. Table II) but very high diameter. Since there is not much scope for optimistic parallelism due to low degree-count, *DC* has slower execution time than JP in this case(Fig. 6d). With Twitter input, *DC* achieves comparable performance with 1536 cores (Fig. 6b). The particular Twitter dataset we have experimented with is generated from the follower network. In this social network graph, followers are not connected to each other and hence do not create dense local subgraphs. This is in contrast to the topological structure we observe in RMAT-B, RMAT-$\widetilde{G}$ and Graph500 inputs, which have multiple dense locally connected subgraphs.

D. Comparison to The PowerGraph Framework

We compare the performance of our implementations in AM++ with a similar greedy distributed vertex-coloring algorithm (called *simple coloring*) in PowerGraph [10], a well-known distributed graph processing framework. This helps us to evaluate the efficiency and coloring quality of our algorithms. We performed our experiments with the publicly-available version 2.2 of PowerGraph [23]. PowerGraph processes vertex-centric programs in three phases: *Gather* (gather results from neighbors), *Apply* (compute new updates), and *Scatter (GAS)* (propagate updates to the neighbors). In the *synchronous* execution mode of PowerGraph, each of these micro-steps is separated by a barrier, where all vertices gather and scatter at the same time. The *asynchronous* mode of PowerGraph executes GAS phases without the barrier synchronization. However, before each GAS iteration can proceed, active vertices need to acquire locks on their neighbors to prevent two neighbors from choosing the same value simultaneously. Acquiring a

lock on a high-degree active vertex can limit the scalability of the asynchronous algorithm in PowerGraph for power-law graphs. If the PowerGraph coloring algorithm used a vertex ordering such as the one in *DC* and JP, it would be a pull-based version of these algorithms. However, always locking the whole neighborhood of a vertex for a pull step irrespectively of the source of the triggering change generates a large overhead when compared to a push-based algorithm such as *DC*.

We ran simple coloring algorithm with the asynchronous graph processing mode of the PowerGraph framework. With the synchronous execution mode, PowerGraph coloring algorithm fails to converge [24].

1) RMAT weak scaling: Figure 3 shows the weak scaling results of our implementations and of the simple coloring algorithm in PowerGraph. From Fig. 3, we can see that our implementation of Jones-Plassmann performs better at higher scale with RMAT-ER compared to the PowerGraph implementation. As we increase the problem size, vertex-centric barrier becomes a bottleneck in JP for graphs with skewed degree distribution. Asynchronous execution engine of PowerGraph outperform JP in such cases. Except for RMAT-ER input, *DC* outperforms other two implementations in all cases at higher scale. Also, PowerGraph can not run beyond 3072 cores due to the limitation of the framework. We also tabulated the coloring quality of these implementations in Fig. 3. Both JP and *DC* achieve the same color quality. Compared to PowerGraph, the coloring quality of the AM++ algorithms are better. The slower performance of the coloring algorithm in PowerGraph with power-law graphs can be attributed to the restricted parallelism imposed by the vertex locking mechanism needed for data consistency and updates. We also tried to run two other vertex-coloring algorithms in PowerGraph with different ordering heuristics: saturation-ordered and degree-ordered coloring. Unfortunately these two algorithms fail to complete execution in a reasonable time.

2) RMAT strong scaling: Figure 5 shows strong scaling of PowerGraph coloring algorithm with scale-26 graph. With Graph500 and RMAT-ER, PowerGraph performs worse than JP and DC with increased no. of cores. In many cases, PowerGraph fails with larger cores.

3) Real-world datasets: Table III tabulates speedup of DC over PowerGraph with real-world datasets collected from [21]. Except com-orkut dataset, *DC* outperforms PowerGraph in all cases. Road network graphs require fewest number of colors, yet PowerGraph requires every active vertex update to acquire a lock on all its neighbors. For this reason, in such cases, DC performs almost 20 times better compared to PowerGraph due to no requirement of vertex-centric locking.

E. Workload Characteristics

Figure 7 shows the breakdown of different types of work executed by each vertex-coloring algorithm. We classify workload performed by each algorithm in 3 categories: useful, useless, and rejected work. *Useful* works are the aggregate count of tasks that contain the final predecessor color values and result in successful color updates. Both Jones-Plassmann and Distributed Control execute the same amount of such work over the course of execution. Useful work yields final vertex colors. The other two types of tasks are specific to the *DC* coloring algorithm. In *DC*, whenever a better color value becomes available, it forces an invalidation of the current color and ultimately triggers a correction of the current vertex color value. These updates happen when *DC* processes workitems from the priority queues and tries to update a vertex color with a newly available color. Once updated, the new color information is sent to all the successors. However, instead of sending these updates immediately, we cache these messages for some time. *Useless* work arises when a color update becomes stale in the application-level message cache while waiting to be sent to the successors. *Rejected* work goes over the network but on arrival gets rejected due to containing outdated update. As can be seen in Fig. 7, *DC* performs more work compared to JP. However, only rejected work results in network messages. Compared to other inputs, RMAT-ER results in the largest amount of this type of work, hence *DC* has worst performance with RMAT-ER input. Nonetheless, the elimination of vertex-centric barriers in *DC* results in significant performance benefit on graphs with densely connected subcommunities and strong power-law.

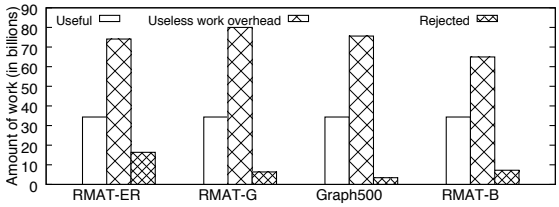


Fig. 7: Workload statistics (512 compute nodes, scale 31 graph).

Ordering	No. colors	Ordering time (s)	Coloring time (s)	Color ratio
Random	500	0.21	11.68	1.06
Largest first	388	0.2	10.91	1.36
Incidence degree	365	30.49	11.47	1.452
Smallest last	366	32.21	11.36	1.448

TABLE IV: Color quality, sequential ordering time, and coloring time of a Scale 24 Graph500 graph in ColPack [25]. The color ratio column reports ratio of no. of colors obtained by our algorithm to the sequential version with a particular ordering. The ordering is tabulated from the least restrictive ordering to the most restrictive one.

F. Coloring Quality

In Table IV, we compare the color quality of our algorithm (with random ordering heuristic for pre-execution order) to those in a well-known coloring software package ColPack [25]. ColPack provides sequential and parallel versions of algorithms for a range of graph coloring problems. As can be seen from Table IV, in the worst case, our algorithm uses 1.45 times more colors than the sequential version with incidence-degree ordering. However, in sequential setting, even with small scale 24 graph, it takes 30.49s to obtain the incidence degree order. Smallest last, the most restrictive ordering among the ordering schemes, takes about 32.21s to order the vertices. On the other hand, random ordering does not require any ordering time, and it uses few extra colors. Imposing other orderings in distributed

JP	DC	DC_SD	DC_ID
155.24	75.66	65.39	63.30

TABLE V: Execution time (in seconds) of DC algorithms (DC_ID: incidence degree heuristic, DC_SD: saturation degree heuristic) and the JP algorithm with Graph500 scale-28 graph on 64 compute nodes.

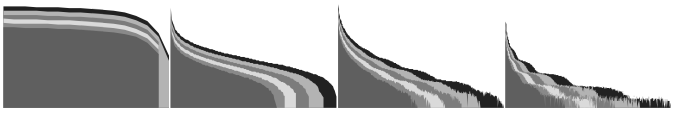


Fig. 8: From left-to-right: Distribution of color bins for RMAT-ER, RMAT-$\widetilde{G}$, RMAT-B, and Graph500 (color numbers on the x-axis and color counts on the y-axis). The plot combines scales 25-30, indicated by shades of gray.

setting is more involved. Finally, random ordering is necessary for the performance guarantees in DAG-based algorithms [5].

G. Impact Of Approximate Ordering Heuristics on DC As Priority Metrics

As discussed in Sec. III, we experiment with approximate ID and SD heuristics for prioritizing incoming messages for target vertices. We present our results in Table V. DC with the ID heuristic outperforms other algorithms on the test input. It is important to note that these heuristics only affect performance and not the total no of colors since the total number of colors is decided by the pre-ordering of vertices in the DAG (the heuristics are used during execution to avoid wasted work).

H. Skewness of Sizes of Color Classes

In Fig. 8, we show the sizes of different color classes (independent subsets) for 4 types of RMAT graphs. Sizes of color classes can assist to choose between JP and DC. If the sizes of the color classes are such that most of the vertices are concentrated in the lower-numbered color bins (for example in RMAT-B, RMAT-$\widetilde{G}$ and Graph500 in Fig. 8), it can be a good predictor of better performance of DC over JP with such inputs. This is because, due to the speculative nature of DC, it tends to choose first the minimum available color in a search and propagates such speculated color choice as soon as possible. The larger the lower color bins are, right predictions will be made earlier in the computation of DC. Moreover, amount of rejected work is highest (Fig. 7) with graph that has the most equal color distribution (RMAT-ER).

VI. Related work

Greedy Ordering Heuristics. The first greedy coloring algorithm was proposed by [26]. Different vertex ordering heuristics for Jones-Plassmann algorithm have been proposed and evaluated in [11].

Empirically Evaluated Shared-Memory Algorithms. A two-step distance-1 coloring algorithm was proposed in [7]. They evaluated the algorithm on shared memory system with small-size inputs. Authors in [19] studied the interplay between architectures and algorithms in the context of vertex-coloring algorithms. They evaluated the algorithms on synthetic RMAT graphs. The approach proposed by Deveci et al. [27] introduced a set of refinements and optimizations over the two-stage (assignment and correction) coloring procedure by Gebremedhin and Manne for many-core architectures and iterative graph coloring algorithms. In contrast, our distributed algorithm eliminates the need to separate the conflict detection phase from the assignment phase with optimistic execution, thus eliminating the bottlenecks associated with synchronization that separates these stages.

Theoretical Distributed Algorithms. Based on the message passing model, many theoretical algorithms have been proposed for distributed graph coloring problem [28], [29], [30], [31], [32]. In this model, message communication happens within a set of synchronized steps. For example, [29] present a deterministic $(\Delta + 1)$-coloring distributed algorithm with running time of $O(\Delta) + \frac{1}{2} \log^* n$. We refer to [33], [34] for further discussion.

Empirically Evaluated Distributed-Memory Algorithms. [35] proposed a framework for parallelizing greedy coloring algorithm for static graphs. The framework partitions a graph among several processors and *speculatively* colors the vertices greedily and resolves conflicts in a set of synchronized *supersteps*. However, they experimented with small-size graphs. [36] reported performance of graph coloring algorithm based on Luby's parallel maximal independent set algorithm [3] on a Pregel-like system. However, their implementation is not scalable (for example sk2005 took about 26 minutes on 90 nodes). Both of the previous approaches are based on synchronous supersteps. [15] evaluated Jones-Plassmann algorithm in distributed settings for static and dynamic graphs. Our optimistic parallelism and label-correction approach is related to progressive reads semantics in [37], however no consideration was given about prioritizing work.

VII. Conclusion

In this paper, we have presented a label-correcting, completely asynchronous vertex-coloring algorithm that is based on optimistic parallelization. The algorithm eliminates vertex-centric barriers and global synchronization altogether and relies only on atomic operations to update vertex colors. The time obtained by elimination of such barriers is invested in ordering tasks to minimize the effect of executing sub-optimal work. We have demonstrated the potential of our algorithm with power-law graphs containing densely connected subcommunities – both in terms of scalability and performance.

VIII. Acknowledgement

This work is supported by the NSF grant no. 1716828, Lilly Endowment, Inc, and by the Defense Advanced Research Projects Agency's (DARPA) Hierarchical Identify Verify Exploit Program (HIVE) at the DOE Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. We also thank the anonymous reviewers and the shepherd for their comments and help in improving the paper.

REFERENCES

[1] K. Lang, "Fixing two weaknesses of the spectral method," in *Advances in Neural Information Processing Systems*, 2006, pp. 715–722.

[2] P. Erdos, R. L. Graham, and E. Szemeredi, "On sparse graphs with dense long paths," *Comp. and Math. with Appl*, vol. 1, pp. 145–161, 1975.

[3] M. Luby, "A Simple Parallel Algorithm for the Maximal Independent Set Problem," in *Proc. 17th Annual ACM Symposium on Theory of Computing*, ser. STOC '85. New York, NY, USA: ACM, 1985, pp. 1–10.

[4] M. T. Jones and P. E. Plassmann, "A Parallel Graph Coloring Heuristic," *SIAM J. Sci. Comput.*, vol. 14, no. 3, pp. 654–669, May 1993.

[5] G. E. Blelloch, J. T. Fineman, and J. Shun, "Greedy Sequential Maximal Independent Set and Matching Are Parallel on Average," in *Proc. 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12. ACM, 2012, pp. 308–317.

[6] J. S. Firoz, M. Zalewski, T. A. Kanewala, and A. Lumsdaine, "Synchronization-avoiding graph algorithms," in *25th International Conference on High Performance Computing (HiPC)*. IEEE, 2018.

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

[8] M. Kulkarni and K. Pingali, "Scheduling issues in optimistic parallelization," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–7.

[9] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A Generalized Active Message Framework," in *Proce. 19th Int. Conf. on Parallel Architectures and Compilation Techniques*. ACM, 2010, pp. 401–410.

[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[11] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '14. New York, NY, USA: ACM, 2014, pp. 166–177.

[12] L. Lovśz, M. Saks, and W. T. Trotter, "An on-line graph coloring algorithm with sublinear performance ratio," *Discrete Math.*, vol. 75, no. 1-3, pp. 319–325, Sep. 1989.

[13] T. F. Coleman and J. J. Moré, "Estimation of sparse jacobian matrices and graph coloring blems," *SIAM journal on Numerical Analysis*, vol. 20, no. 1, pp. 187–209, 1983.

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

[15] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 30:1–30:12.

[16] A. B. Sinha, L. V. Kale, and B. Ramkumar, "A dynamic and adaptive quiescence detection algorithm," *University of Illinois at Urbana-Champaign, Urbana-Champaign*, 1993.

[17] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining." in *SDM*, M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, Eds. SIAM, 2004, pp. 442–446.

[18] http://www.graph500.org/, accessed: 2017-12-31.

[19] 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, pp. 576–594, 2012.

[20] "Laboratory of web algorithmics," http://law.di.unimi.it/datasets.php, accessed: January 2018.

[21] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[22] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25.

[23] "Graphlab git repository," https://github.com/jegonzal/PowerGraph, accessed: January 2018.

[24] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation," in *20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2015.

[25] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, "Colpack: Software for graph coloring and related problems in scientific computing," *ACM Trans. Math. Softw.*, vol. 40, no. 1, pp. 1:1–1:31, Oct. 2013.

[26] 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.

[27] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring for manycore architectures," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 892–901.

[28] N. Alon, L. Babai, and A. Itai, "A fast and simple randomized parallel algorithm for the maximal independent set problem," *J. Algorithms*, vol. 7, no. 4, pp. 567–583, Dec. 1986.

[29] L. Barenboim and M. Elkin, "Distributed (Δ+1)-coloring in linear (in Δ) time," in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: ACM, 2009, pp. 111–120.

[30] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon, "Parallel symmetry-breaking in sparse graphs," *SIAM J. Discret. Math.*, vol. 1, no. 4, pp. 434–446, Oct. 1988.

[31] M. Goldberg and T. Spencer, "A new parallel algorithm for the maximal independent set problem," *SIAM J. Comput.*, vol. 18, no. 2, pp. 419–427, Apr. 1989.

[32] N. Linial, "Locality in distributed graph algorithms," *SIAM J. Comput.*, vol. 21, no. 1, pp. 193–201, Feb. 1992.

[33] J. Schneider and R. Wattenhofer, "A new technique for distributed symmetry breaking," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, 2010, pp. 257–266.

[34] L. Barenboim and M. Elkin, "Deterministic distributed vertex coloring in polylogarithmic time," *Journal of the ACM (JACM)*, vol. 58, no. 5, p. 23, 2011.

[35] 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.

[36] S. Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *Proc. VLDB Endow.*, vol. 7, no. 7, pp. 577–588, Mar. 2014.

[37] K. Vora, "Exploiting asynchrony for performance and fault tolerance in distributed graph processing," Ph.D. dissertation, UC Riverside, 2017.