High-Performance Parallel Graph Coloring with Strong Guarantees on Work, Depth, and Quality

Maciej Besta¹, Armon Carigiet¹, Zur Vonarburg-Shmaria¹, Kacper Janda², Lukas Gianinazzi¹, Torsten Hoefler¹
¹Department of Computer Science, ETH Zurich
²Faculty of Computer Science, Electronics and Telecommunications, AGH-UST

Abstract—We develop the first parallel graph coloring heuristics with strong theoretical guarantees on work and depth and coloring quality. The key idea is to design a relaxation of the vertex degeneracy order, a well-known graph theory concept, and to color vertices in the order dictated by this relaxation. This introduces a tunable amount of parallelism into the degeneracy ordering that is otherwise hard to parallelize. This simple idea enables significant benefits in several key aspects of graph coloring. For example, one of our algorithms ensures polylogarithmic depth and a bound on the number of used colors that is superior to all other parallelizable schemes, while maintaining work-efficiency. In addition to provable guarantees, the developed algorithms have competitive run-times for several real-world graphs, while almost always providing superior coloring quality. Our degeneracy ordering relaxation is of separate interest for algorithms outside the context of coloring.

This is the full version of a paper published at ACM/IEEE Supercomputing’20 under the same title

I. INTRODUCTION

Graph coloring, more specifically vertex coloring, is a well-studied problem in computer science, with many practical applications in domains such as sparse linear algebra computations [1]–[7], conflicting task scheduling [8]–[11], networking and routing [12]–[20], register allocation [21], [22], and many others [23]. A vertex coloring of a graph $G$ is an assignment of colors to vertices, such that no two neighboring vertices share the same color. A $k$-coloring is a vertex coloring of $G$ which uses $k$ distinct colors. The minimal amount of colors $k$ for which a $k$-coloring can be found for $G$ is referred to as the chromatic number $\chi(G)$. An optimal coloring, also sometimes referred to as the coloring problem or a $\chi$-coloring, is the problem of coloring $G$ with $\chi(G)$ colors. Finding such and optimal coloring was shown to be NP-complete [24].

Nonetheless, colorings with a reasonably low number of colors can in practice be computed quite efficiently in the sequential setting using heuristics. One of the most important is the Greedy heuristic [25], which sequentially colors vertices by choosing, for each selected vertex $v$, the smallest color not already taken by $v$’s neighbors. This gives a guarantee for a coloring of $G$ with at most $\Delta + 1$ colors, where $\Delta$ is the maximum degree in $G$. To further improve the coloring quality (i.e., #colors used), Greedy is in practice often used with a certain vertex ordering heuristic, which decides the order in which Greedy colors the vertices. Example heuristics are first-fit (FF) [25] which uses the default order of the vertices in $G$, largest-degree-first (LF) [25] which orders vertices according to their degrees, random (R) [26] which chooses vertices uniformly at random, incidence-degree (ID) [1] which picks vertices with the largest number of uncrowded neighbors first, saturation-degree (SD) [27], where a vertex whose neighbors use the largest number of distinct colors is chosen first, and smallest-degree-last (SL) [28] that removes lowest degree recursively colors the resulting graph, and then colors the removed vertices. All these ordering heuristics, combined with Greedy, have the inherent problem of no parallelism.

Jones and Plassmann combined this line of work with earlier parallel schemes for deriving maximum independent sets [29], [30] and obtained a parallel graph coloring algorithm (JP) that colors a vertex $v$ once all of $v$’s neighbors that come later in the provided ordering have been colored. They showed that JP, combined with a random vertex ordering (JP-R), runs in expected depth $O(\log n/\log \log n)$ and $O(n + m)$ work for constant-degree graphs ($n$ and $m$ are #vertices and #edges in $G$, respectively). Recently, Hasenplaugh et al. [31] extended JP with the largest-log-degree first (LLF) and smallest-log-degree-last (SLL) orderings with better bounds on depth; these orderings approximate the LF and SL orderings, respectively. There is also another (earlier) work [32] that – similarly to JP-SLL – approximates SL with the “ASL” ordering. The resulting coloring combines JP with ASL, we denote it as JP-ASL. [32]. However, it offers no bounds for work or depth.

Overall, there is no parallel algorithm with strong theoretical guarantees on work and depth and quality. Whilst having a reasonable theoretical run-time, JP-R may offer colorings of poor quality [31], [33]. On the other hand, JP-LF and JP-SL, which provide a better coloring quality, run in $\Omega(n)$ or $\Omega(\Delta^2)$ for some graphs [31]. This was addressed by the recent JP-LLF and JP-SLL algorithms [31] that produce colorings of similarly good quality to their counterparts JP-LF and JP-SL, and run in an expected depth that is within a logarithmic factor of JP-R. However, no guaranteed upper bounds on the coloring quality (#colors), better than the trivial $\Delta + 1$ bound from Greedy, exist for JP-LF, JP-SLL, or JP-ASL.

To alleviate these issues, we present the first graph coloring algorithms with provably good bounds on work and depth and quality, simultaneously ensuring high performance and competitive quality in practice. The key idea is to use a novel vertex ordering, the provably approximate degeneracy ordering (ADG, contribution #1) when selecting which vertex is the next to be colored. The exact degeneracy ordering is – intuitively – an ordering obtained by iteratively removing ver-
tics of smallest degrees. Using the degeneracy ordering with JP leads to the best possible Greedy coloring quality [28]. Still, computing the exact degeneracy ordering is hard to parallelize: for some graphs, it leads to $\Omega(n)$ coloring run-time [31]. To tackle this, we (provably) relax the strict degeneracy order by assigning the same rank (in the ADG ordering) to a batch of vertices that – intuitively – have similarly small degrees. This approach also results in provably higher parallelization because each batch of vertices can be processed in parallel.

This simple idea, when applied to graph coloring, gives surprisingly rich outcome. We use it to develop three novel graph coloring algorithms that enhance two relevant lines of research. We first combine ADG with JP, obtaining JP-ADG (contribution #2), a coloring algorithm that is parallelizable: vertices with the same ADG rank are colored in parallel. It has the expected worst-case depth of $O(\log^2 n + \log \Delta + \log^2 \log n)$, where $\Delta$ is the degeneracy of a graph $G$: an upper bound on the minimal degree of every induced subgraph of $G$ (detailed in $\S$ II-B) [28]. JP-ADG is also work-efficient ($O(n+m)$ work) and has good coloring quality: it uses at most $2(1+\varepsilon)d+1$ colors \(1\) for any $\varepsilon > 0$. Moreover, we also combine ADG with another important line of graph coloring algorithms that are not based on JP but instead use speculation [32], [34]–[46]. Here, vertices are colored independently ("speculative coloring"). Potential coloring conflicts (adjacent vertices assigned the same colors) are resolved by repeating coloring attempts. Combining ADG with this design gives DEC-ADG (contribution #3), the first scheme based on speculative coloring with provable strong guarantees on all key aspects of parallel graph coloring: work $O(n+m)$, depth $O(\log \Delta)$, and quality $(2+\varepsilon)d$. Finally, we combine key design ideas in DEC-ADG with an existing recent algorithm [40] (referred to as ITR) also based on speculative coloring. We derive an algorithm called DEC-ADG-ITR that improves coloring quality of ITR both in theory and practice.

We conduct the most extensive theoretical analysis of graph coloring algorithms so far, considering 20 parallel graph coloring routines with provable guarantees (contribution #5). All our algorithms offer substantially better bounds than past work. Compared to the most recent JP-SLL and JP-LLF colorings [31], JP-ADG gives a strong theoretic coloring guarantee that is much better than $\Delta+1$ because $d \ll \Delta$ for many classes of sparse graphs, such as scale-free networks [47] and planar graphs [48]. It also provides an interesting novel tradeoff in the depth. On one hand, it depends on $\log n$ while JP-SLL and JP-LLF depend on $\log \Delta$. However, while JP-SLL and JP-LLF depend linearly on $\Delta$ or $\sqrt{m}$ (which are usually large in today’s graphs), the depth of JP-ADG depends linearly on degeneracy $d$, and, as we also show, $d \ll \sqrt{m}$ and $d \ll \Delta$. We also perform a broad empirical evaluation, illustrating that our algorithms (1) are competitive in run-times for several real-world graphs, while (2) offering superior coloring quality. Only JP-SL and JP-SLL use comparably few colors, but they are at least $1.5\times$ slower. Our routines are of interest for coloring both small and large graphs, for example in online execution scheduling and offline data analytics, respectively. We conclude that our algorithms offer the best coloring quality at the smallest required runtime overhead.

In a brief summary, we offer the following:

- The first parallel algorithm for deriving the (approximate) graph degeneracy ordering (ADG).
- The first parallel graph coloring algorithm (JP-ADG), in a line of heuristics based on Jones and Plassman’s scheme, with strong bounds on work, depth, and coloring quality.
- The first parallel graph coloring algorithm (DEC-ADG), in a line of heuristics based on speculative coloring, with strong bounds on work, depth, and coloring quality.
- A use case of how ADG can seamlessly enhance an existing state-of-the-art graph coloring scheme (DEC-ADG-ITR).
- The most extensive (so far) theoretical analysis of parallel graph coloring algorithms, showing advantages of our algorithms over state-of-the-art in several dimensions.
- Superior coloring quality offered by our algorithms over tuned modern schemes for many real-world graphs.

We note that degeneracy ordering is used beyond graph coloring [49]–[52]; thus, our ADG scheme is of separate interest.

II. FUNDAMENTAL CONCEPTS

We start with background: Table I lists key symbols. Vertex coloring was already described in $\S$ I.

A. Graph Model and Representation

We model a graph $G$ as a tuple $(V,E)$; $V$ is a set of vertices and $E \subseteq V \times V$ is a set of edges; $|V| = n$ and $|E| = m$. We focus on graph coloring problems where edge directions are not relevant. Thus, $G$ is undirected. The maximum, minimum, and average degree of a given graph $G$ are $\Delta$, $\delta$, and $\bar{\delta}$, respectively. The neighbors and the degree of a given vertex $v$ are $N(v)$ and $\deg(v)$, respectively. $G[U] = (U,E[U])$ denotes an induced subgraph of $G$: a graph where $U \subseteq V$ and $E[U] = \{(v,u) \mid v \in U \land u \in U\}$, i.e., $E[U]$ contains edges with both endpoints in $U$. $N_U(v)$ and $\deg_U(v)$ are the neighborhood and the degree of $v \in V$ in $G[U]$. The vertices are identified by integer IDs that define a total order: $V = \{1, \ldots, n\}$. These IDs define a total order $\succ$ on the vertices that is used to sort the neighborhoods. We store $G$ using CSR, the standard graph representation that consists of $n$ sorted arrays with neighbors of each vertex (2$m$ words) and offsets to each array ($n$ words).

B. Degeneracy, Coreness, and Related Concepts

A graph $G$ is $s$-degenerate [53] if, in each of its induced subgraphs, there is a vertex with a degree of at most $s$. The degeneracy $d$ of $G$ [54]–[57] is the smallest $s$, such that $G$ is still $s$-degenerate. The degeneracy ordering of $G$ [28] is an ordering, where each vertex $v$ has at most $d$ neighbors that are ordered higher than $v$. Then, a $k$-approximate degeneracy...
A graph \( G = (V, E); V \) and \( E \) are sets of vertices and edges. \( G[U] = (U, E[U]) \) is a subgraph of \( G \) induced on \( U \subseteq V \). Numbers of vertices and edges in \( G \); \(|V| = n, |E| = m\).

Maximum degree, minimum degree, and average degree of \( G \).

The degree of \( v \) and the neighborhood of a vertex \( v \in V \).

The degree of \( v \) in a subgraph induced by the vertex set \( U \subseteq V \).

The neighborhood of \( v \) in a subgraph induced by \( U \subseteq V \).

A priority function \( V \to \mathbb{R} \) associated with vertex ordering \( X \).

The number of processors in a given PRAM machine.

A list of selected symbols used in the paper. When we use a symbol in the context of a specific loop iteration \( \ell \), we add \( \ell \) in brackets or as subscript (e.g., \( \delta \ell \) is \( \delta \) in iteration \( \ell \)).

### C. Models for Algorithm Analysis

As a compute model, we use the DAG model of dynamic multitithreading [59], [60]. In this model, a specific computation (resulting from running some parallel program) is modeled as a directed acyclic graph (DAG). Each node in a DAG corresponds to a constant time operation. In-edges of a node model the data used for the operation. As operations run in constant time, there are \( O(1) \) in-edges per node. The out-edges of a node correspond to the computed output. A node can be executed as soon as all predecessors finish executing.

Following related work [31], [61], we assume that a parallel computation (modeled as a DAG) runs on the ideal parallel computer (machine model). Each instruction executes in unit time and there is support for concurrent reads, writes, and read-modify-write atomics (any number of such instructions finish in \( O(1) \) time). These are standard assumptions used in all recent parallel graph coloring algorithms [31], [61]. We develop algorithms based on these assumptions but we also provide algorithms that use weaker assumptions (algorithms that only rely on concurrent reads).

We use the work-depth (W–D) analysis for bounding runtimes of parallel algorithms in the DAG model. The work of an algorithm is the total number of nodes and the depth is defined as the longest directed path in the DAG [62], [63].

Our Analyses vs. PRAM In our W–D analysis, two used machine model variants (1) only need concurrent reads and (2) may also need concurrent writes. These variants are analogous to those of the well-known PRAM model [63–66]: CREW and CRCW, respectively. Thus, when describing a W–D algorithm that only relies on concurrent reads, we use a term “the CREW setting”. Similarly, for a W–D algorithm that needs concurrent writes, we use a term “the CRCW setting”.

The machine model used in this work is similar to PRAM. One key difference is that we do not rely on the (unrealistic) PRAM assumption of the synchronicity of all steps of the algorithm (across all processors). The well-known Brent’s result states that any deterministic algorithm with work \( W \) and depth \( D \) can be executed on \( P \) processors in time \( T \) such that \( \max\{W/P, D\} \leq T \leq W/P + D \) [67]. Thus, all our results are applicable to a PRAM setting.

### D. Compute Primitives

We use a Reduce operation. It takes as input a set \( S = \{s_1, ..., s_n\} \) implemented as an array (or a bitmap). It uses a function \( f: S \to \mathbb{N} \) called the operator; \( f(s) \) must be defined for any \( s \in S \). Reduce calculates the sum of elements in \( S \) with respect to \( f: f(s_1) + ... + f(s_n) \). This takes \( O(\log n) \) depth and \( O(n) \) work in the CREW setting [68], [69], where \( n \) is the array size. We use Reduce to implement Count \((S)\), which computes the size \( |S| \). For this, the associated operator \( f \) is defined as \( f(s) = 1 \) if \( s \in S \), and \( f(s) = 0 \) otherwise. Reduce, as well as the more general PrefixSum operation, are well studied parallel primitives, which are used in numerous parallel computations and algorithms [69]. We also assume a DecrementAndFetch (DAF) to be available; it atomically decrements its operand and returns a new value [31]. We use DAF to implement Join to synchronize processors (Join decrements its operand, returns the new value, and releases a processor under a specified condition). For details, we refer to the paper of Hasenplugh et al. [31].

### E. Randomization

Randomization has been used in numerous algorithms [30], [70]–[75]. We distinguish between Monte Carlo algorithms, which return a correct result w.h.p., and Las Vegas algorithms, which always return the correct result but have probabilistic run-time bounds. The JP algorithm, with an ordering heuristic that employs randomness like ADG, is a Las Vegas algorithm. For a large part of the analysis we use Random Variables to describe various random events. To prove that statements hold w.h.p. we mainly use simple Markov and Chernoff bounds.

Randomization is an important tool for proving bounds on the performance of parallel algorithms. It allows us to prove that certain events are likely to occur with high probability, even if they are not guaranteed to happen.

A statement holds with high probability (w.h.p.) if it holds with a probability greater than \( 1 - \frac{1}{n^c} \) for all \( c \).
III. PARALLEL APPROXIMATE DEGENERACY ORDERING

We first describe ADG, a parallel algorithm for computing a partial approximate degeneracy ordering. ADG outputs vertex priorities $p_{ADG}$, which are then used by our coloring algorithms (Section IV). Specifically, these priorities produce an order in which to color the vertices (ties are broken randomly).

ADG is shown in Algorithm 1. ADG is similar to SL [28], which iteratively removes vertices of the smallest degree to construct the exact degeneracy ordering. The key difference and our core idea is to repeatedly remove in parallel all vertices with degrees smaller than $(1 + \varepsilon)\delta$. The parameter $\varepsilon \geq 0$ controls the approximation accuracy. We multiply $1 + \varepsilon$ by the average degree $\delta$ as it enables good bounds on quality and run-time, as we show in Lemma 1 and 4. Compared to SL (which has depth $O(n)$), ADG has depth $O(\log^2 n)$ and obtains a partial $2(1 + \varepsilon)$-approximate degeneracy ordering.

```
/* Input: A graph G(V, E).
   * Output: A priority (ordering) function $\rho: V \to \mathbb{R}$.*/
D = [deg(v1) deg(v2) ... deg(vn)] //An array with vertex degrees.
ℓ = 1; U = V //U is the induced subgraph used in each iteration ℓ.

while U ≠ ∅ do
  |U| = Count(U); //Derive |U| using a primitive Count
  cnt = Reduce(U); //Derive the sum of degrees in |U|: $\sum_{v \in U} D[v]
  \delta = \frac{\text{cnt}}{|U|} //Derive the average degree for vertices in U

  if (|U| ≥ (1 + \varepsilon)\delta) then
    δ = \frac{\text{cnt}}{|U|} //Derive the average degree for vertices in U

  R = \{v ∈ U | D[v] ≤ (1 + \varepsilon)\delta \}
  UPDATE(U, R, D); //Update D to reflect removing R from U
  U = U \ R //Remove selected low-degree vertices (that are in R).

  for all v ∈ R do in parallel:
    //Set the priority of vertices
    $p_{ADG}(v) = \delta$ //The priority is the current iteration number ℓ

  ℓ += 1

  UPDATE (U, R, D);
  for all v ∈ R do in parallel:
    DecrementAndFetch(D[v])

Algorithm 1: ADG, our algorithm for computing the $2(1 + \varepsilon)$-approximate degeneracy ordering; it runs in the CRCW setting.
```

In ADG, we maintain a set $U \subseteq V$ of active vertices that starts as $V$ (Line 5). In each step (Lines 8–18), we use $\varepsilon$ and $\delta$ to select vertices with small enough degrees (Line 13); The average degree $\delta$ is computed in Lines 8–10. The selected vertices form a set $R$ and receive a priority $p_{ADG}$ equal to the step counter $\ell$. We then remove them from the set $U$ (Line 15) and update the degrees $D$ accordingly (Line 14). We continue until the set $U$ is empty. When used with JP, ties between vertices with the same $p_{ADG}$ are broken with another priority function $\rho'$ (e.g., random ordering).

**Design Details**

For the theoretical analysis we use the following design assumptions. We implement $D$ as an array and use $n$-bit dense bitmaps for $U$ and $R$. This enables updating vertex degrees in $O(1)$ and resolving $v \in U$ and $v \in R$ in $O(1)$ time. Constructing $R$ in each step can be implemented in $O(1)$ depth and $O(|U|)$ work. The operation $U = U \setminus R$ takes $O(1)$ depth and $O(|R|)$ work by overwriting the bitmap $U$. To calculate the average degree on Line 10, we derive $|U|$ and sum all degrees of vertices in $U$. The former is done with a Count over $U$. The latter uses Reduce with the associated operator $f(v) = D[v]$. As both Reduce operations run in $O(\log n)$ depth and $O(|U|)$ work, the same holds for the average degree calculation.

**Depth**

First, note that each line in the while loop runs in $O(\log n)$ depth, as discussed above. We will now prove that the while loop iterates $O(\log n)$ many times, giving the total ADG depth of $O(\log^2 n)$. The key notion is that, in each iteration, we remove a constant fraction of vertices due to the way that we construct $R$ (based on the average degree $\delta$). We will further use the following fact.

**Lemma 1.** For a constant $\varepsilon > 0$, ADG does $O(\log n)$ iterations and has $O(\log^2 n)$ depth in the CRCW setting.

**Proof.** At each step $\ell$ of the algorithm we can have at most $\frac{n}{1 + \varepsilon}$ vertices with a degree larger than $(1 + \varepsilon)\delta$. This can be seen from the fact, that the sum of degrees in the current subgraph can be at most $n$ times the average degree $\delta$. For vertices with a degree exactly $(1 + \varepsilon)\delta$ we get $\frac{n}{1 + \varepsilon} \cdot (1 + \varepsilon)\delta = n\delta$, which would result in a contradiction if we had more than $\frac{n}{1 + \varepsilon}$ vertices with larger degree. Thus, if we remove all vertices with degree $\leq (1 + \varepsilon)\delta$, we remove a constant fraction of vertices in each iteration (at least $\frac{n}{1 + \varepsilon}$ vertices), which implies that ADG performs $O(\log n)$ iterations in the worst case, immediately giving the $O(\log^2 n)$ depth. To see this explicitly, one can define a simple recurrence relation for the number of iterations $T(n) ≤ 1 + T\left(\frac{n}{1 + \varepsilon}\right)$, $T(1) = 1$; solving it gives $T(n) ≤ \left[\frac{\log \frac{n}{\log(1 + \varepsilon)} + 1}{\log(1 + \varepsilon)}\right] ∈ O(\log n)$.

**Work**

The proof of work is similar; it also uses the fact that a constant fraction of vertices is removed in each iteration. Intuitively, (1) we show that each while loop iteration performs $O\left(\sum_{v \in R} deg(v) + |U|\right)$ work (where $U$ is the set $U$ in iteration $i$), and (2) we bound $\sum_{i=1}^{k} |U_i|$ by a geometric series, implying that it is still in $O(n)$.

**Lemma 2.** For a constant $\varepsilon > 0$, ADG does $O(n + m)$ work in the CRCW setting.

**Proof.** Let $k$ be the number of iterations we perform and let $U_i$ be the set $U$ in iteration $i$. To calculate the total work performed by ADG, we first consider the work in one iteration. As explained in “Design Details”, deriving the average degree takes $O(|U_i|)$ work in one iteration. Initializing $R$ takes $O(|U_i|)$ and removing $R$ from $U_i$ takes $O(|R|)$. UPDATE takes $O\left(\sum_{v \in R} deg(v)\right)$ work. Thus, the total work in one iteration is in $O\left(\sum_{v \in R} deg(v) + |U_i|\right)$. As each vertex becomes included in $R$ in a single unique iteration, this gives $\sum_{i=1}^{k} \sum_{v \in R} deg(v) ∈ O(m)$. Moreover, since we remove a constant number of vertices in each iteration from $U$ (at least $\frac{n}{1 + \varepsilon}$ as shown above in the proof of the depth of ADG), we can bound $\sum_{i=1}^{k} |U_i|$ by a geometric series, implying that it is still in $O(n)$. This can be seen from the fact that $\sum_{i=1}^{k} |U_i| ≤ \sum_{i=0}^{k} \left(\frac{1}{1 + \varepsilon}\right) n ≤ \sum_{i=0}^{\infty} \left(\frac{1}{1 + \varepsilon}\right)^i n = \frac{n}{1 + \varepsilon}$ if $\frac{1}{1 + \varepsilon} < 1$ (which holds as $\varepsilon > 0$). Ultimately, we have $O\left(\sum_{i=0}^{k} \left(\sum_{v \in R} deg(v) + |U_i|\right)\right) ∈ O(m) + O(n) ∈ O(m + n)$.
Approximation ratio. We now prove that the approximation ratio of ADG on the degeneracy order is $2(1 + \varepsilon)$. First, we give a small lemma used throughout the analysis.

**Lemma 3.** Every induced subgraph of a graph $G$ with degeneracy $d$, has an average degree of at most $2d$.

**Proof.** By the definition of a $d$-degenerate graph, in every induced subgraph $G[U]$, there is a vertex $v$ with $deg_U(v) \leq d$. If we remove $v$ from $G[U]$, at most $d$ edges are removed. Thus, if we iteratively remove such vertices from $G[U]$, until only one vertex is left, we remove at most $d \cdot |U| - 1$ edges. We conclude that $\delta(G[U]) = \frac{1}{|U|} \sum_{v \in U} deg_U(v) \leq 2d$.

**Lemma 4.** ADG computes a partial $(2(1 + \varepsilon))$-approximate degeneracy ordering of $G$.

**Proof.** By the definition of $R$ (Line 13), all vertices removed in step $\ell$ have a degree of at most $(1 + \varepsilon)\delta_{i}$, where $\delta_{i}$ is the average degree of vertices in subgraph $U$ in step $\ell$. From Lemma 3, we know that $\delta_{i} \leq 2d$. Thus, each vertex has a degree of at most $2(1 + \varepsilon)d$ in the subgraph $G[U]$ (in the current step). Hence, each vertex has at most $2(1 + \varepsilon)d$ neighbors that are ranked equal or higher. The result follows by the definition of a partial $(2(1 + \varepsilon))$-approximate degeneracy order.

A. Comparison to Other Vertex Orderings

We analyze orderings in Table II. While SLL and ASL heuristically approximate SL, which computes a degeneracy ordering, they do not offer guaranteed approximation factors. Only ADG comes with provable bounds on the accuracy of the degeneracy order while being (provably) parallelizable.

**Lemma 5.** A variant of ADG as specified in Algorithm 2 and in § III-B does $O(m + nd)$ work in the CREW setting.

**Proof.** The proof is identical to that for ADG in the CRCW setting, the difference is in analyzing the impact of UPDATE on total work. To compute $Count(N_{U}(v) \cap R_{i})$ (cf. Algorithm 2), we use a Reduce over $N_{U}(v)$. Since $R_{i} \subseteq U_{i}$, we can define the operator $f$ as $f(v) = 1$ if $v \in R_{i}$, and $f(v) = 0$ otherwise. Thus, when computing total work in iteration $i$, instead of $\sum_{v \in R_{i}} deg_{U}(v)$, we must consider $\sum_{v \in U_{i}} deg_{U}(v)$. Consequently, we have $\sum_{i=1}^{k} \sum_{v \in U_{i}} deg_{U}(v) \leq \sum_{i=1}^{k} (2d \cdot |U_{i}|) \leq 2d \cdot \sum_{i=1}^{k} |U_{i}| \in O(nd)$ (the first inequality by Lemma 3, the second one by the observation that we remove a constant fraction of vertices in each iteration, cf. the proof of Lemma 1).

IV. PARALLEL GRAPH COLORING

We now use our approximate degeneracy ordering to develop new parallel graph coloring algorithms. We directly enhance the recent line of works based on scheduling colors, i.e., assigning colors to vertices without generating coloring conflicts (§ IV-A). In two other algorithms, we allow conflicts but we also provably resolve them fast (§ IV-B, § IV-C).

A. Graph Coloring by Color Scheduling (JP-ADG)

We directly enhance recent works of Hasenplaugh et al. [31] by combining their Jones-Plassmann (JP) version of coloring with our ADG, obtaining JP-ADG. For this, we first overview JP and definitions used in JP. The JP algorithm uses the notion of a computation DAG $G_{\rho}(V, E_{\rho})$, which is a directed version of the input graph $G$. Specifically, the DAG $G_{\rho}$ is used by JP to schedule the coloring of the vertices: The position of a vertex $v$ in the DAG $G_{\rho}$ determines the moment the vertex $v$ is colored. The DAG $G_{\rho}$ contains the edges of $G$ directed from the higher priority to lower priority vertices according to the priority function $\rho$, i.e., $E_{\rho} = \{(u, v) \in E \mid \rho(u) > \rho(v)\}$.

JP is described in Algorithm 3. As input, besides $G$, it takes a priority function $\rho: V \rightarrow \mathbb{R}$ which defines a total order on vertices $V$. First, JP uses $\rho$ to compute a DAG $G_{\rho}(V, E_{\rho})$, where edges always go from vertices with higher $\rho$ to ones with lower $\rho$ (Alg. 3, Lines 6–9). Vertices can then be safely assigned a color if all neighbors of higher $\rho$ (predecessors in the DAG) have been colored. The algorithm does this by first calling JPColor with the set of vertices that have no predecessors (Alg. 3, Lines 13–15). JPColor then colors $v$ by calling GetColor, which chooses the smallest color not already taken by $v$’s predecessors. Afterwards, JPColor checks if any of $v$’s successors can be colored, and if yes, it calls again JPColor on them. As shown by Hasenplaugh et al. [31], this algorithm can run in $O(|P| \log \Delta + \log n)$ depth and $O(n + m)$ work, where $|P|$ is the size of the longest path in $G_{\rho}$.

Now, in JP-ADG, we first call ADG to derive $\rho_{ADG}$. Then, we run JP using $\rho_{ADG}$. More precisely, we use $\rho = \langle \rho_{ADG}, \rho_{R} \rangle$ where $\rho_{R}$ randomly breaks ties of vertices that were removed in the same iteration in Algorithm 1, and thus have the same rank in $\rho_{ADG}$. The obtained JP-ADG algorithm is similar to
Part 1: compute the DAG $G_\rho$ based on $\rho$.

5 $C = \{0, 0, ... 0\}$ //Initialize colors

6 for all $v \in V$ do in parallel:

7 //Predecessors and successors of each vertex $u$ in $G_\rho$;

8 $\text{pred}[v] = \{u \in N(v) \mid |u| \geq |v|\}$

9 $\text{succ}[v] = \{u \in N(v) \mid |u| > |v|\}$

10 //Number of uncolored predecessors of each vertex $u$ in $G_\rho$:

11 count[$v$] = $|\text{pred}[v]|$

12

13 //Part 2: color vertices using $G_\rho$

14 for $v$ all $v \in V$ do in parallel:

15 //Start by coloring all vertices without predecessors:

16 if $\text{pred}[v] = \emptyset$ do $\text{JPColor}(v)$

17 $\text{JPColor}(v)$ //JPColor, a routine used in JP

18 $C[v] = \text{GetColor}(v)$

19 for all $u \in \text{succ}[v]$ do in parallel:

20 if $\text{Join}(\text{count}[u]) = 0$

21 $\text{GetColor}(v)$ //Color $v$ if it has no uncolored predecessors

22 $\text{JPColor}(u)$ //Color $u$ with the smallest color available.

23 return min ($C$) //Output the smallest color available.

Algorithm 3: JP, the Jones-Plassman coloring heuristic. With $\rho = \langle \rho_{\text{ADG}}, \rho_R \rangle$, it gives JP-ADG that provides $2(1+\varepsilon)d+1$-coloring.

past work based on JP in that it follows the same “skeleton” in which coloring of vertices is guided by the pre-computed order, on our case $\rho_{\text{ADG}}$. However, as we prove later in this section, using ADG is key to our novel bounds on depth, work, and coloring quality. Intuitively, ADG gives an ordering of vertices in which each vertex has a bounded number of predecessors (by definition of $s$-degenerate graphs and graph degeneracy $d$). We use this to bound coloring quality and sizes of subgraphs in $G_\rho$. The latter enables bounding the maximum path in $G_\rho$, which in turn gives depth and work bounds.

As for different combinations of JP and orderings, we define them similarly to past work. JP-R is JP with a random priority function $\rho_R$. JP-FF uses the natural vertex order. JP-LF uses $\rho(v) = \langle \text{deg}(v), |v| \rangle$ with a lexicographic order. JP-SL is defined by $\rho(v) = \langle \rho_{\text{SL}}, |v| \rangle$, with a degeneracy ordering $\rho_{\text{SL}}$. JP-LLF is defined by $\rho = \langle |\text{deg}(v)|, |v| \rangle$ and JP-SLL by $\rho = \langle \rho_{\text{SL}}, |v| \rangle$, where $\rho_{\text{SL}}$ is the order computed by the SLL algorithm from Hasenplaugh et al. [31].

We first prove a general property of JP-ADG, which we will use to derive bounds on coloring quality, depth, and work.

**Lemma 6.** JP, using a priority function $\rho$ that defines a $k$-approximate degeneracy ordering, colors a graph $G$ with at most $kd+1$ colors, for $\varepsilon > 0$.

**Proof.** Since $\rho$ defines a $k$-approximate degeneracy ordering, any $v \in V$ has at most $kd$ neighbors $v'$ with $\rho(v') \geq |v|$ and thus at most $kd$ predecessors in the DAG. Now, we can choose the smallest color available from $\{1, \ldots, kd+1\}$ to color $v$, when all of its predecessors have been colored. □

**Coloring Quality.** The coloring quality now follows from the properties of the priority function obtained with ADG.

**Corollary 1.** With priorities $\rho = \langle \rho_{\text{ADG}}, \rho_R \rangle$, JP-ADG colors a graph with at most $2(1+\varepsilon)d+1$ colors, for $\varepsilon > 0$.

**Depth.** To bound the depth of JP-ADG, we follow the approach by Hasenplaugh et al. [31]. We analyze the expected length of the longest path in a DAG induced by JP-ADG to bound its expected depth. We first provide some additional definitions. The induced subgraph of $G_\rho$ is a (directed) subgraph of $G_\rho$ induced by a given vertex set. Note that as $\rho$ is a total order on $V$, the DAG $G_\rho$ is strongly connected. Therefore, we define $\overrightarrow{\rho} = \max_{v \in V} \{\rho_{\text{ADG}}(v)\}$.

**Lemma 7.** For a priority function $\rho = \langle \rho_{\text{ADG}}, \rho_R \rangle$, where $\rho_{\text{ADG}}$ is a partial $k$-approximate degeneracy ordering for a constant $k > 1$, $\rho_R$ is a random priority function, the expected length of the longest path in the DAG $G_\rho$ is $O \left( d \log n + \frac{\log d \log^2 n}{\log \log n} \right)$.

**Proof.** Let $G_\rho(\ell)$ be the subgraph of $G_\rho$ induced by the vertex set $V(\ell) = \{v \in V \mid \rho_{\text{ADG}} = \ell\}$. Let $\ell$ be the maximal degree and $\delta_\ell$ be the average degree of the subgraph $G_\rho(\ell)$.

Since, by the definition of $G_\rho$, there can be no edges in $G_\rho$ that go from one subgraph $G_\rho(\ell)$ to another $G_\rho(\ell')$ with $\ell' > \ell$, we can see that a longest (directed) path $P$ in $G_\rho$ will always go through the subgraph $G_\rho(\ell)$ in a monotonically decreasing order with regards to $\ell$. Therefore, we can split $P$ into a sequence of (directed) sub-paths $P_1, \ldots, P_\ell$, where $P_\ell$ is a path in $G(\ell)$. We have $|P| = \sum_{\ell \in \{\rho_{\text{ADG}}(v)\} \in V} |P_\ell|$ and by Corollary 6 from past work [31], the expected length of a longest sub-path $P_\ell$ is in $O(\Delta_\ell + \log \Delta_\ell \log n / \log \log n)$, because $G_\rho(\ell)$ is induced by a random priority function. By linearity of expectation, we have for the whole path $P$:

$$E[|P|] = O \left( \sum_{\ell=1}^{\Delta_\ell} \Delta_\ell + \log \Delta_\ell \cdot \frac{\log n}{\log \log n} \right)$$

Next, since $\rho_{\text{ADG}}$ is a partial $k$-approximate degeneracy ordering, all vertices in $G(\ell)$ have at most $kd$ neighbors in $G(\ell)$. Thus, $\Delta_\ell \leq kd$ holds. This and the fact that $\overrightarrow{\rho} \in O(\log n)$ gives:

$$\sum_{i=1}^{\overrightarrow{\rho}} \Delta_i \leq \sum_{i=1}^{\overrightarrow{\rho}} d \cdot k \in O(d \log n)$$

$$\sum_{i=1}^{\overrightarrow{\rho}} \log \Delta_i \in O(\log d \log n)$$

Thus, for the expected length of a longest path in $G$:

$$E[|P|] = O \left( d \log n + \frac{\log d \log^2 n}{\log \log n} \right)$$

Our main result follows by combining our bounds on the longest path $P$ in the DAG $G_\rho$ and a result by Hasenplaugh et al. [31], which shows that JP has $O(\log n + \log \Delta_\cdot |P|)$ depth.
Theorem 1. JP-ADG colors a graph \( G \) with degeneracy \( d \) in depth \( O(\log^2 n + \log \Delta \cdot (d \log n + \log d \log n)) \) and \( O(n + m) \) work in the CRCW setting.

Proof. Since \( \rho_{ADG} \) is a partial \((1 + \varepsilon)\)-approximate degeneracy ordering (Lemma 4) and since ADG performs at most \( O(\log n) \) iterations (Lemma 1), the depth follows from Lemma 7, Lemma 1 and past work [31], which shows that JP runs in \( O(\log n + \log \Delta \cdot |P|) \) depth. As both JP and ADG perform \( O(n + m) \) work, so does JP-ADG. \( \square \)

B. Graph Coloring by Silent Conflict Resolution (DEC-ADG)

Our second coloring algorithm takes a radical step to move away from the long line of heuristics based on JP. The key idea is to use ADG to decompose the input graph into low-degree partitions (thus “DEC-ADG”), shown in Algorithm 4. Here, ADG is again crucial to our bounds. Specifically, vertices with the same ADG rank form a partition that is “low-degree”: it has a bounded number of edges to any other such partitions (by the definition of ADG). Each such partition is then colored separately, with a simple randomized scheme in Algorithm 5. (by the definition of ADG). Each such partition is then colored separately, with a simple randomized scheme in Algorithm 5.

We first detail Algorithm 4. A single low-degree partition \( G(\ell) \) produced by the iteration \( \ell \) of ADG is the induced subgraph of \( G \) over the vertex set \( R \) removed in this iteration (Line 13, Alg. 1). Formally, \( G(i) = G[R(i)] \) where \( R(i) = \{ v \in V \mid \rho(v) = i \} \) and \( \rho \) is the partial \( k \)-approximate degeneracy order produced by ADG (cf. § II-B). Thus, in DEC-ADG, we first run ADG to derive the ordering \( \rho \) and also the number \( p \) of low-degree partitions \( (p \in O(\log n)) \). Here, we use ADG*, a slightly modified ADG that also records – as an array \( G = [G(1) \ldots G(p)] \) – each low-degree partition. Then, we iterate over these partitions (starting from \( p \) and color each with SIM-COL (“SIMple coloring”, Alg. 5). We discuss SIM-COL in more detail later in this section, its semantics are that it colors a given arbitrary graph \( G \) in our context \( G \) is the \( \ell \)-th partition \( G(\ell) \) using \((1 + \mu)\Delta \) colors, where \( \mu > 0 \) is an arbitrary value. To keep the coloring consistent with respect to already colored partitions, we maintain bitmaps \( B_v \) that indicate colors already taken by \( v \)’s neighbors in already colored partitions: If \( v \) cannot use a color \( c \), the \( c \)-th bit in \( B_v \) is set to 1. These bitmaps are updated before each call of SIM-COL (Lines 16–18).

How large should \( B_v \) be to minimize storage overheads but also ensure that each vertex has enough colors to choose from? We observe that a single bitmap \( B_v \) should be able to contain at most as many colors as neighbors of \( v \) in a partition currently being colored \( (G(\ell)) \), and in all partitions that have already been colored \( (G(\ell'), \ell' > \ell) \). We denote this neighbor count with \( \text{deg}_v(\ell) \). Observe that any \( \text{deg}_v(\ell) \) is at most \( kd = \lceil (1 + \varepsilon)d \rceil \), as partitions are created according to a partial \( k \)-approximate degeneracy order where \( k = 2(1 + \varepsilon) \) (note that, in DEC-ADG, we use factors \( \varepsilon/4 \) and \( \varepsilon/12 \) instead of \( \varepsilon \) for more straightforward proofs (this is possible as \( \varepsilon \) can be an arbitrary non-negative value).

In SIM-COL, we color a single low-degree partition \( G(\ell) = (V(\ell), E(\ell)) \). SIM-COL takes two arguments: (1) the partition to be colored (it can be an arbitrary graph \( G = (V, E) \) but for clarity we explicitly use \( G(\ell) = (V(\ell), E(\ell)) \) that denotes a partition from a given iteration \( \ell \) in DEC-ADG) and (2) bitmaps associated with vertices in a given partition \( R(\ell) \). By design, SIM-COL delivers a \((1 + \mu)\Delta\)-coloring; \( \mu > 0 \) can be an arbitrary value. To be able to derive the final bounds for DEC-ADG we set \( \mu = \varepsilon/4 \). \( U \) is vertices still to be colored, initialized as \( U = V(\ell) \). In each iteration, vertices in \( U \) are first colored randomly. Then, each vertex \( v \) compares its color \( C[v] \) to the colors of its active (not yet colored) neighbors in \( N_U \) and checks if \( C[v] \) is not already taken by other neighbors inside and outside of \( V(\ell) \) (by checking \( B_v \), see Lines 9–13. The goal is to identify whether at least one such neighbor has the same color as \( v \). For this, we use Reduce over \( N_U(v) \) with the operator \( f \) defined as \( f_{eq}(u) = (C[v] \iff C[u]) \) (the “\( \iff \)” operator works analogously to the same-name operator in C++) and a simple lookup in \( B_v \). If \( v \) and \( u \) have the same color, \( f_{eq}(u) \) equals 1. Thus, if any of \( v \)’s neighbors in \( U \) have the same color as \( v \), Reduce\(N_U(v), f_{eq} > 0 \). This enables us to attempt to re-color \( v \) by setting \( C[v] = 0 \). If a vertex gets colored, we remove it from \( U \) (Line 19) and update the bitmaps of its neighbors (Line 18). We iterate until \( U \) is empty.

**Depth, Work** We now prove the time complexity of DEC-ADG. The key observation is that the probability that a particular vertex becomes inactive (permanently colored) is constant regardless of the coloring status of its neighbors. The key proof technique is to use Markov and Chernoff Bounds.
Algorithm 5: SIM-COL, our simple coloring routine used by DEC-ADG. It delivers a \((1 + \mu)\Delta\)-coloring, where \(\mu > 0\) is an arbitrary value. When using SIM-COL as a subroutine in DEC-ADG, we instantiate \(\mu = \varepsilon/4\); we use this value in the listing above for concreteness.

Before we proceed with the analysis, we provide some definitions. For each round \(\ell\) of SIM-COL (Algorithm 5), we define an indicator random variable \(X_v\) to refer to the event in which a vertex \(v\) gets removed from \(U\) (i.e., becomes colored and thus inactive) in this specific round \(\ell\). The vertex \(v\) is removed if and only if the color \(C[v]\), which is selected on Line 7, is not used by some neighbor of \(v\) (i.e., this color is not in \(B_v\)) and no active neighbor chose \(C[v]\) in this round.

The random variable \(X_v\) indicates the complement of event \(X_v\) (i.e., a vertex is not removed from \(U\) in a given round). Next, let \(Z\) be a Bernoulli random variable with probability \(Pr[Z = 1] = p = 1 - \frac{1}{1+\mu}\), and let \(\overline{Z}\) be the complement of \(Z\). Finally, we use a concept of stochastic dominance: For two random variables \(A\) and \(B\) that are defined over the same set of possible outcomes, we say that \(A\) stochastically dominates \(B\) if and only if \(Pr[A \geq z] \geq Pr[B \geq z]\), for all \(z\).

In the following, we show that the event of an arbitrary vertex \(v\) becoming deactivated \((X_v = 1)\) is at least as probable as \(Z = 1\). This will enable us to use these mutually independent variables \(Z\) to analyze the time complexity of SIM-COL and DEC-ADG.

**Claim 1.** In every iteration, for every vertex \(v\), the probability that the vertex \(v\) becomes inactive is at least \(1 - \frac{1}{1+\mu}\).

**Proof.** The probability that \(v\) becomes inactive in any iteration \((Pr[X_v = 1])\) is at least \(1 - \frac{1}{\text{deg}(v)}\), where \(i\) is the number of distinct colors in \(B_v\) and received from neighbors in this round. This is because, in each iteration, while \(v\) connects to vertices with a total of \(i\) distinct colors, the total number of colors to be selected from is \((1 + \mu)\text{deg}(v)\). Now, as \(\mu\) can have at most \(\text{deg}(v)\) colored neighbors, we get \(1 - \frac{1}{\text{deg}(v)}\), which shows that \(Pr[X_v = 1] \geq Pr[Z = 1]\) holds for all active \(v\).

Thus, in expectation, a constant fraction of the vertices becomes inactive in every iteration. Now, in the next step, we will apply Markov and Chernoff bounds to an appropriately chosen binomial random variable, showing that the number of vertices that are removed is concentrated around its expectation. Hence, the algorithm terminates after \(O(\log n)\) iterations.

First, to proof the intuitive fact that \(Z\) can be used to approximate the number of vertices removed in each round, we use the technique of coupling (See § II-E) together with a handy equivalence between stochastic dominance and the coupling of two random variables [76].

**Lemma 8.** The random variable \(X = \sum_{v \in U} X_v\) stochastically dominates \(Y = \sum_{i = 1}^{\lceil |U|/\mu \rceil} Z\).

**Proof.** Without loss of generality let us assume that all vertices in \(U\) are numerated from 1 to \(|U|\), such that we can write \(X = \sum_{i = 1}^{\lceil |U|/\mu \rceil} X_i\). Further note, that we have shown in Claim 1 that the event \(X_v = 1\) happens independently with a probability of at least \(1 - \frac{1}{1+\mu}\) and therefore \(Pr[X_v = 1] \geq Pr[Z = 1]\) holds. We want to construct a coupling \((\tilde{X}, \tilde{Y})\) with \(Pr[\tilde{X} \geq \tilde{Y}] = 1\) to show the stochastic dominance of \(X\) [76].

First let \(\tilde{Y}_i\) be equal to \(Y_i\) for all \(i = 1..|U|\) and define \(\tilde{Y}\) to be \(\sum_{i = 1}^{\lceil |U|/\mu \rceil} \tilde{Y}_i\). Next define \(\tilde{X}_i\) to be 0 whenever \(\tilde{Y}_i\) = 0. If \(\tilde{Y}_i\) is 1 we define the probability of \(\tilde{X}_i\) to be \(Pr[X_i = 1]\) conditioned under \(X_i \geq 1\).

We can see that by design \(\tilde{X}_i \geq \tilde{Y}_i\) holds for all \(i = 1..|U|\) and therefore, with Theorem 4.23 of other work [76], conclude that \(X\) dominates \(Y\).

**Lemma 9.** A random variable \(\overline{X} = \sum_{v \in U} \overline{X_v}\) is stochastically dominated by a random variable \(\overline{Y} = \sum_{i = 1}^{\lceil |U|/\mu \rceil} \overline{Z}\).

**Proof.** From Claim 1 we already know that \(\forall v \in U. Pr[X_v = 1] \geq Pr[Z = 1]\). Since \(X_v\) and \(Z\) are Bernoulli random variables, \(Pr[Z = 1] \geq Pr[X_v = 1]\) holds for their complements. Now with a similar argument as shown in Lemma 8 we can again conclude the proof.

**Lemma 10.** SIM-COL performs \(O(\log n)\) iterations w.h.p. (for constant \(\mu > 1\)).

**Proof.** The number of vertices that are not removed (i.e., not permanently colored) in one round can be expressed as \(\overline{X} = \sum_{v \in U} \overline{X_v}\) (\(\overline{X_v}\) is 1 means that \(v\) is not removed from \(U\) in a given iteration). Now, we apply the Markov inequality to \(\overline{Y}\):

\[
Pr[\overline{Y} \leq \frac{\mu |U|}{1+\mu}] \leq \frac{\mathbb{E}[\overline{Y}]}{\frac{\mu |U|}{1+\mu}} = \frac{\frac{\mu |U|}{1+\mu}}{\frac{\mu |U|}{1+\mu}} = 1.
\]

As \(\overline{Y}\) dominates \(\overline{X}\) (Lemma 9), we have

\[
Pr[\overline{\overline{X}} \geq \frac{\mu |U|}{1+\mu}] \geq Pr[\overline{\overline{X}} \geq \frac{\mu |U|}{1+\mu}] = \frac{\mu |U|}{1+\mu}.
\]

Consequently, \(Pr[\overline{\overline{X}} \geq \frac{\mu |U|}{1+\mu}] \leq \frac{1}{\mu}\). Hence,

\[
Pr[\overline{\overline{X}} \geq \frac{\mu |U|}{1+\mu}] \leq 1 - \frac{1}{\mu}.
\]

Note that \(0 < 1 - \frac{1}{\mu} \leq 1\) for \(\mu > 1\). Thus, with probability at least \(1 - \frac{1}{\mu}\), at least \(\frac{|U|}{1+\mu}\) vertices are removed from \(U\) (i.e., permanently colored) in each round.
For some constants $c_1, c_2$, let now $c_1 \cdot \log n + c_2$ be the number of iterations of SIM-COL performed. We want to derive a number of iterations where we deactivated at least $\frac{\mu}{\mu + \eta}$ vertices. This can be expressed as a random variable $Q = \sum_{i=1}^{c_1 \cdot \log n + c_2} Q_i$, where $Q_i$ is the Bernoulli random variable that indicates if in iteration $i$ sufficiently many vertices were deactivated. This happens independently, as shown above, with probability $P_r[Q_i = 1] \geq \frac{\mu}{\mu + \eta}$. Now we apply a Chernoff bound to $X$ and get:

$$P_r \left[ Q < (1 + \eta) \frac{\mu}{\mu + \eta} (c_1 \log n + c_2) \right] \geq 1 - e^{-\frac{\mu}{\mu + \eta} \left( c_1 \log n + c_2 \right) \left( (2 + \eta) \frac{\mu}{\mu + \eta} - 1 \right)}$$

Thus, if we choose $0 < \eta < 1$ and $c_1$ appropriately, it follows that we perform $O(\log n)$ iterations w.h.p.

To bound the work of SIM-COL, we can observe that similarly to the number of vertices, the number of edges incident to at least one active vertex also decreases by a constant factor in each iteration with high probability. The work in an iteration is bounded by this number of edges and every iteration has depth $O(\Delta)$ (in the CREW setting) and $O(\log \Delta)$ (in the CRCW setting). Hence, we conclude:

**Lemma 11.** SIM-COL takes $O(\Delta \log n)$ depth (in the CREW setting) or $O(\log \Delta \log n)$ depth (in the CRCW setting), and it has $O(n + m)$ work w.h.p. in the CREW setting.

Proof. Since the reduction on Line 13 can be implemented in $O(\log (\deg(v)))$ depth for each $v \in U$ and since updating the bitmaps in Part 3 takes $O(\Delta)$ (assuming CREW) and $O(\log \Delta)$ (assuming CRCW) we get, together with Lemma 10, an overall depth of $O(\Delta \log n)$ (assuming CREW) and $O(\log \Delta \log n)$ (assuming CRCW) w.h.p.

Second, to analyze the work performed by the algorithm we will first look at the work performed in one outer-loop
iteration. Let $U_i$ be the set $U$ in iteration $i$. Let $n_i = |U_i|$ be the number of active vertices and $m_i$ be the number of active edges at the start of iteration $i$. Choosing colors for all vertices $v \in U_i$ (Line 7) takes $O(|U_i|)$ work in each iteration. The reduction for all $v \in U_i$ (Line 13) takes $O\left(\sum_{v \in U_i} \deg(v)\right)$ work. Updating the bitmaps (Line 18) also takes $O\left(\sum_{v \in U_i} \deg(v)\right)$ work. Removing vertices that have been colored in the current iteration from $U_i$ can be done in $O(|U_i|)$ work. Therefore the work for one iteration is bound by $O\left(\sum_{v \in U_i} \deg(v) + |U_i|\right) = O(m_i + n_i).

To argue further we want to make a case distinction over the number of active vertices.

**Case 1** Iterations where $n_i = O(\log n)$ but $n_i \not\in \Theta(n)$:
We can model the number of vertices that get removed in each iteration as $X = \sum_{v \in U_i} X_v$. This random variable, as seen from Lemma 8, stochastically dominates $Y = \sum_{v \in U_i} Z$, i.e., for all valid $x$ we have $\Pr[X \geq x] \geq \Pr[Y \geq x]$. Note that $\mathbb{E}[Y] = \frac{n_i}{1 + n_i}$ and that because of our assumption we have $n_i > c_1 \log n + c_2$ for any constants $c_1 > 0$ and $c_2$. As $Y$ is defined over independent random variables $Z$, we can apply a Chernoff bound and, as $X$ stochastically dominates $Y$:

$$
\Pr\left[X \geq (1 - \eta) \mathbb{E}[Y]\right] \leq e^{-\frac{\eta^2 \mathbb{E}[Y]}{2}}
$$

This implies, for appropriately chosen constants $0 < \eta < 1$ and $c_1$, that in each iteration, where we have active vertices in order of $\log n$, at least a constant fraction of them will be deactivated w.h.p. Now, by using a union bound over all the iterations, which are in $O(\log n)$ w.h.p. (Lemma 10), we can also conclude that w.h.p. we will always deactivate a constant fraction of vertices. Further, we know that an edge is deactivated with at least the same probability as one of its incident vertices. Hence, w.h.p., we will also deactivate a constant fraction of our active edges in each iteration. Thus, the overall work performed is in $O(n + m)$ w.h.p.

**Case 2** Iterations where $n_i \in O(\log n)$: As there are at most $O(\log n)$ active vertices left, we can bound the work in one iteration by $O(\log n + \log^2 n)$. With Lemma 10, we get overall work of $O(\log^2 n)$ w.h.p. for these iterations.

Therefore, we get $O(n + m)$ work w.h.p. over all iterations.

Now, we turn our attention back to DEC-ADG. As DEC-ADG decomposes the edges of the input graph into $O(\log n)$ disjoint subgraphs of maximum degree $O(d)$, we have:

**Lemma 12.** DEC-ADG takes $O(\log d \log^2 n)$ depth and $O(n + m)$ work w.h.p. in the CRCW setting.

**Proof.** Since the graph partitions $G(\ell)$ are induced by a partial $kd$-approximate degeneracy order, we know that each $G(\ell)$ has a maximal degree of at most $kd$. Therefore, we can conclude together with Lemma 11, that SIM-COL (Algorithm 5), if called on $G(\ell)$, induces $O(\log d \log n)$ depth w.h.p. (assuming CRCW) and performs $O(\mathcal{R}(\ell) + |E|\mathcal{R}(\ell))$ work in expectation (i.e., work proportional to the number of vertices and edges in $G(\ell)$). Computing ADG takes $O(\log^2 n)$ depth and $O(n + m)$ work. Updating bitmaps takes $O(\log d)$ depth (with a simple Reduce), since each vertex in $G(\ell)$ has at most $kd$ neighbors in $Q$, and $O(\sum_{v \in \mathcal{R}(\ell)} \deg(v))$ work.

Now, since DEC-ADG (Algorithm 4) performs $O(\log n)$ iterations, we get an overall depth of $O(\log d \log^2 n)$ w.h.p.. More precisely, this can be seen from Lemma 10. Since we use a Chernoff bound to bound the number of iterations performed by SIM-COL, we can also guarantee that each of the $O(\log n)$ instances of SIM-COL will perform $O(\log n)$ iterations w.h.p.

The work in one loop iteration can be bound by $O(\mathcal{R}(\ell) + |E|\mathcal{R}(\ell) + \sum_{v \in \mathcal{R}(\ell)} \deg(v))$ in expectation (as seen above). Thus, since each vertex is in exactly one partition $\mathcal{R}(\ell)$ and each edge is in at most one (i.e., in at most one set $E(\mathcal{R}(\ell))$ of edges that belong to a subgraph $G(\ell)$), we can conclude that Algorithm 4 performs $O(n + m)$ work in expectation.

**Coloring Quality.** Finally, we prove the coloring quality.

**Claim 2.** DEC-ADG produces a $(2 + \varepsilon)d$ coloring of the graph for $0 < \varepsilon \leq 8$.

**Proof.** Since we use ADG to partition the graph into $(\rho, G)$ ($G \equiv G(1) \ldots G(\pi)$) on Line 8, we know that $\rho$ is a partial $(2 + \varepsilon/12)$-approximate degeneracy ordering. Therefore, we also know that each vertex $v \in G(i)$ has at most $2(1 + \varepsilon/12)d$ neighbors in partitions $G(i')$ with $i' \geq i$. This implies, that if we run SIM-COL on each partition $G(i)$, we will color each partition with at most $(1 + \varepsilon/42)(1 + \varepsilon/12)d$ colors. This is by the design of SIM-COL, which delivers a $(1 + \Delta \varepsilon)$-coloring for any graph $G$; in our case, we have $G \equiv G(i)$, $\Delta = 2(1 + \varepsilon/12)$, and $\mu = \varepsilon/4$. Now, $(1 + \varepsilon/42)(1 + \varepsilon/12)d$ is smaller or equal to $(2 + \varepsilon)d$ for $\varepsilon \leq 8$, as $(1 + \varepsilon/42)(1 + \varepsilon/12) = \left(2 + \frac{168 + \varepsilon^2}{24}\right) \leq (2 + \varepsilon)$, for $\varepsilon \leq 8$.

Now we have already seen that the runtime bounds of DEC-ADG hold for $\mu > 1$ (Lemma 11). Since we defined $\mu$ to be $\varepsilon/4$ they hold for $\varepsilon > 4$. Thus, together with Claim 2 we can see that the algorithm attains it's runtime and color bounds for $4 < \varepsilon \leq 8$.

**C. Enhancing Existing Coloring Algorithms.**

Finally, we illustrate that ADG does not only provide new provably efficient algorithms, but also can be used to enhance existing ones. For this, we seamlessly replace our default SIM-COL routine with a recent speculative coloring heuristic, ITR, by Çatalyurek et al. [40]. The result, DEC-ADG-ITR, is similar to DEC-ADG, except that the used SIM-COL differs in Line 7 from the default Algorithm 5: colors are not picked by the design of SIM-COL, which delivers a $(1 + \Delta \varepsilon)$-coloring for any graph $G$; in our case, we have $G \equiv G(i)$, $\Delta = \varepsilon/2$, and $\mu = \varepsilon/4$. Now, $(1 + \varepsilon/4)(1 + \varepsilon/2)d$ is smaller or equal to $(2 + \varepsilon)d$ for $\varepsilon \leq 8$, as $(1 + \varepsilon/4)(1 + \varepsilon/2) = \left(2 + \frac{168 + \varepsilon^2}{24}\right) \leq (2 + \varepsilon)$, for $\varepsilon \leq 8$.

Using ADG enables deriving similar bounds on coloring quality $\left(2 + \varepsilon\right)d$ as before. Since each vertex in partition $G(\ell)$ can have at most $kd$ colored neighbors when it gets colored, as shown before, we always choose the smallest color
form \{1, ..., kd + 1\}. However, deriving good bounds on work and depth is hard because the lack of randomization (when picking colors) prevents us from using techniques such as Chernoff bounds. We were still able to provide new results.

Selecting colors can be done in \(O(\Delta)\) and \(O(\Delta \cdot |U_i|)\) work, where \(U_i\) is the set of \(U\) in iteration \(i\) (of the while loop in SIM-COL) and \(\Delta = \max_{v \in V(\ell)} \text{deg}(v)\). For the modified SIM-COL, we get, since all other operations are equal, \(O(\Delta I)\) depth (\(I\) is the number of iterations of ITR); the work is

\[
O \left( \sum_{i=1}^{I} \left[ \Delta \cdot |U_i| + \sum_{v \in U_i} \text{deg}(v) \right] \right).
\]

Depth and work in DEC-ADG-ITER are, respectively

\[
O(Id \log n)
\]

and (as a simple sum over all iterations \(I\))

\[
O \left( \sum_{i=1}^{I} \left[ \sum_{v \in U_i} \text{deg}(v) \right] + \sum_{v \in R(\ell)} \text{deg}(v) \right)
\]

Note that these bounds are valid in the CREW setting.

D. Using Concurrent Reads

Similarly to past work [31], our algorithms rely on concurrent reads. However, a small modification to ADG gives variants that only need concurrent reads (a weaker assumption that is more realistic in architectures that use caches). Specifically, one can implement UPDATE from Algorithm 1 by iterating over all \(v \in U\) in parallel, and for each \(v\), appropriately modifying its degree: \(D[v] = D[v] - \text{Count}(N_U(v) \cap R)\). This makes both ADG and DEC-ADG rely only on concurrent reads but it adds a small factor of \(d\) to work (\(O(nd + m)\)).

E. Comparison to Other Coloring Algorithms

We exhaustively compare JP-ADG and DEC-ADG to other algorithms in Table III. We consider: non-JP parallel schemes (Class 1), the best sequential greedy schemes (Class 2), and parallel algorithms based on JP (Class 3). We consider depth (time), work, used model, quality, generality, randomized design, work-efficiency, and scalability. We also use past empirical analyses [31], [33], [37] and our results (§ VI) to summarize run-times and coloring qualities of algorithms used in practice, focusing on modern real-world graphs as input. Details are in the caption of Table III.

As explained in Section I, only our algorithms work for arbitrary graphs, deliver strong bounds for depth and work quality, and are often competitive in practice. Now, JP-SL may deliver a little higher quality colorings, as it uses the exact degeneracy ordering (although without explicitly naming it) and its quality is (provably) \(d + 1\). However, JP-SL comes with much lower performance. On the other hand, most recent JP-LLF and JP-SLL only provide the straightforward \(\Delta + 1\) bound for coloring quality. These two are however inherently parallel as they depend linearly on \(\log n\), while JP-ADG depends on \(\log^2 n\). Yet, JP-ADG has a different advantage in depth: JP-SLL and JP-LLF depend linearly on \(\sqrt{m}\) or \(\Delta\) while JP-ADG on \(d\). In today’s graphs, \(d\) is usually much (i.e., orders of magnitude) smaller than \(\sqrt{m}\) and \(\Delta\) [61]. To further investigate this, we now provide a small lemma.

Lemma 13. For any \(d\)-degenerate graph, we have \(\sqrt{m} \geq d/2\).

Proof. First we can see, from the definition of degeneracy, that \(G\) is \(d\)-degenerate (i.e. every induced subgraph has at least one vertex with degree \(\leq d\), but not \((d - 1)\)-degenerate. Therefore, since \(G\) is not \((d - 1)\)-degenerate, we will have at least one subgraph \(G(V, E)\) where each vertex has a degree larger than \(d - 1\). This then also implies, that there are at least \(d\) vertices in \(G\). Now we get for the edges of \(G\) that \(|E| = \frac{1}{2} \sum_{v \in V} \text{deg}(v) \geq \frac{1}{2} \sum_{i=1}^{d} d = \frac{d^2}{2}\). Thus, \(G\) has at least \(d^2/2\) edges and therefore we get \(\sqrt{m} \geq d/2\) \(\square\)

Lemma 13 and the fact that \(d \leq \Delta\) illustrate that the expected depth of JP-ADG is up to a logarithmic factor comparable to JP-R, JP-LLF, and JP-SLL. This further illustrates that our bounds on depth in JP-ADG offer an interesting tradeoff compared to JP-LF and JP-LLF.

We finally observe that the design of ADG combined with the parametrization using \(\varepsilon\) enables a tunable parallelism-quality tradeoff. When \(\varepsilon \to 0\), coloring quality in JP-ADG approaches \(2d + 1\), only \(\approx 2\times\) more than JP-SL. On the other hand, for \(\varepsilon \to \infty\), \(\rho_{\text{ADG}}\) becomes irrelevant and the derived final ordering \(\rho = \langle \rho_{\text{ADG}}, \rho_X \rangle\) converges to \(\rho_X\). Now, \(X\) could be the random order \(R\) but also the low-depth LF and LLLF orders based on largest degrees. This enables JP-ADG to increase parallelism tunably, depending on user’s needs.

We also compare JP-ADG to works based on speculation and conflict resolution [38], [40], [43], [45], based on an early scheme by Gebremedhin [37]. A direct comparison is difficult because these schemes do no offer detailed theoretical investigations. Simple bounds on coloring quality, depth, and work are – respectively – \(\Delta + 1\), \(O(\Delta I)\), and \(O(\Delta I P)\), where \(I\) is \#iterations and \(P\) is \#processors. Here, we illustrate that using ADG in combination with these works simplifies deriving better bounds for such algorithms, as seen by the example of DEC-ADG-ITR, see § IV-B.

V. OPTIMIZATIONS AND IMPLEMENTATION

We explored different design choices and optimizations for more performance. Some relevant design choices were already discussed in Section III; these were key choices for obtaining our bounds without additional asymptotic overheads. Here, the main driving question that we followed was how to maximize the practical performance of the proposed coloring algorithms? For brevity, we discuss optimizations by extending Algorithm 1 and 3. We also conduct their theoretical analysis.

A. Representation of \(U\) and \(R\)

The first key optimization (ADG, Alg. 1) is to maintain set \(U\) (vertices still to be assigned the rank \(i\)) and sets \(R(\cdot)\) (vertices removed from \(U\), i.e., \(R(i)\) contains vertices removed in iteration \(i\)) together in the same contiguous array such that all elements in \(R(\cdot)\) precede all elements in \(U\). In iteration \(i\),
this gives an array \([R(1) \ldots R(i)]\) index \(U\), where index points to the first element of \(U\). Initially, in iteration 0, index is 0. Then, in iteration \(i+1\), one extracts \(R(i+1)\) from \(U\) and substitutes \([R(0) R(1) \ldots R(i)]\) index \(U\) with \([R(0) R(1) \ldots R(i)]\) index \(R(i+1) U \setminus R(i+1)\). Formally, we keep an invariant that, if any vertex \(v\) has a degree \((1+\varepsilon)\delta\), then \(v \in R\) (i.e., \(v \in R(0) \cup \ldots \cup R(i+1)\)). Prior to removing \(R(i+1)\) from \(U\), we partition \(U\) into \([R(i+1) U \setminus R(i+1)]\) beforehand, which can be done in time \(O(|U|)\). We do this by iterating over \(U\), comparing the degree of each vertex to \((1+\varepsilon)\delta\), and placing this vertex either in \(R(i+1)\) or in \(U \setminus R(i+1)\), depending on its degree. Then, the actual removal of \(R(i+1)\) from \(U\) takes \(O(1)\) time by simply moving the index pointer by \([R(i+1)\]) positions “to the right”, giving as at the end of iteration \(i+1\) a representation \([R(0) R(1) \ldots R(i) R(i+1) \text{ index } U]\).

B. Explicit Ordering in \(R(\cdot)\)

We advocate to sort each \(R(\cdot)\) by the increasing count of neighbors within the vertex set \(U \cup R(\cdot)\) (i.e., for any two vertices \(v, u \in R(\cdot)\), we place \(v\) before \(u\) in the array representation of \(R(\cdot)\) if and only if \(u\) has more neighbors in \(U \cup R(\cdot)\) than \(v\)). This gives an explicit ordering within the set of vertices that are removed in the same iteration of ADG. Further, this induces a total order on \(\rho_{ADG}\) (i.e., random tie breaking becomes not necessary). Our evaluation indicates that such sorting often enhances the accuracy of the obtained approximate degeneracy ordering, which in turn consistently improves the ultimate coloring accuracy. Sorting can be performed with linear time integer sort and the relative neighbor count of each vertex can be obtained using array \(D\). We also observe that, for some graphs, this additional vertex sorting improves the overall runtime. We additionally explored parallel integer sort schemes used to maintain the above-described representation of \(U \cup R\). We tried different algorithms (radix sort [83], counting sort [84], and quicksort [85]).

C. Combining JP and ADG

We observe that Part 1 of JP-ADG (Lines 6–9, Algorithm 3), where one derives predecessors and successors in a given ordering to construct the DAG \(G_p\), can also be implemented as a part of UPDATE in ADG, in Algorithm 1. To maintain the output semantics of ADG, we use an auxiliary priority function \(\text{rank} : V \rightarrow \mathbb{R}\) that simultaneously specifies the needed DAG structure. For each \(v \in V\), \(\text{rank}[v]\) is defined as the number of neighbors \(u \in N(v)\), for which \(\rho_{ADG}(u) > \rho_{ADG}(v)\) holds. Analogously, \(\text{rank}[v]\) resembles the number of neighbors of \(v\) that were removed from \(U\) after the removal of \(v\) itself. This optimization does not change the theoretical results.

D. Degree Median Instead of Degree Average

We also use degree median instead of degree average in ADG, to derive \(R : R = \{u \in U \mid \delta[u] \leq (1 + \varepsilon)\delta_m\}\), where \(\delta_m\) is median of degrees of vertices in \(U\). We developed variants of ADG (“ADG-M”), JP-ADG (“JP-ADG-M”), and DEC-ADG (“DEC-ADG-M”) that use \(\delta_m\), and analyzed them extensively; they enable speedups for some graphs.

One advantage of ADG-M is that deriving median takes \(O(1)\) time in a sorted array. However, the whole \(U\) has to be sorted in each pass. We incorporate linear-time integer sorting, which was shown to be fast in the context of sorting vertex IDs [86] ADG-M only differs from Algorithm 1 in that, (1) instead of \(\delta\), we select the median degree \(\delta_m\) (of the vertices in \(U\)), and (2) \(R\), the set of vertices removed from \(U\) in a given iteration, is now defined as the vertices \(v \in U\), which satisfy \(\deg_U(v) \leq \delta_m\). Additionally we limit \(R\) to half the size of \(U\) (+1 if \(|U|\) is odd). We refer to the priority function produced by ADG-M as \(\rho_{ADG-M}\).

E. Push vs. Pull

In JP-ADG, computing the updates to \(D\) can be implemented either in the push or the pull style (pushing updates to a shared state or pulling updates to a private state) [36]. More precisely, one can either iterate through the vertices in \(R(\cdot)\) and decrement counters in \(D\) accordingly for each neighbor in \(U\). This is pushing as the accesses to \(D\) modify the shared state. On the other hand, in pulling, one iterates through the remaining vertices in \(U \setminus R(\cdot)\) and counts the number of neighbors in \(R(\cdot)\) for each vertex separately. This number is then subtracted from the relative counter in \(D\). Here, there are no concurrent write accesses to \(D\). We analyzed both variants and found that, while pushing needs atomics, pulling incurs more work. Both options ultimately give similar performance.

F. Caching Sums of Degrees

In each iteration, the average degree of vertices in \(U\) is computed and stored in \(\hat{\delta}\). Here, instead of computing \(\hat{\delta}\) in each iteration by explicitly summing respective degrees, one can maintain the sum of degrees in \(U\), \(\Sigma_U\) (in addition to \(\hat{\delta}\), and update \(\Sigma_U\) accordingly by subtracting the number of edges in the cut \((R(\cdot), U \setminus R(\cdot))\) in each iteration. We omit this enhancement from the listings for clarity; our evaluation indicates that it slightly improves the runtime (by up to \(\approx 1\%\)).

G. Infrastructure Details

We integrated our algorithms with both the GAP benchmark suite [87] and with GBBS [61], a recent platform for testing parallel graph algorithms. We use OpenMP [88] for parallelization. We settle on GBBS.

H. Detailed Algorithm Specifications

We now provide detailed specifications of our algorithms with the optimizations described earlier in this section. We refer to them as ADG-O and ADG-M-O. The former is our fundamental ADG algorithm (described in Section III) that uses the average degree to select vertices for removal in a given iteration. The latter is the ADG variant that uses the median degree instead of the average degree (described in § V-D). The respective listing (of ADG-O) is in Algorithm 6. We omit the listing of ADG-M-O because it is identical to that of ADG-O. The only difference is in the
way the PARTITION subroutine works. Specifically, it results in \( U = \{ R, \delta_m, u_i | R_i + 2 \ldots u_i | U, \} \) such that \( \forall u \in U : D[u] \leq D[\delta_m] \Rightarrow u \in R, \). \( \delta_m \) is the median of the degrees of vertices in \( U \) based on the increasing degree ordering.

Note that in the UPDATEandPRIORITYIZE subroutine, when we refer to \( N(v) \), this technically is \( N_{\{U,R\}}(v) \). However, as enforcing the induced neighbor set entails performance overheads, and as using \( N(v) \) is not incorrect, we use \( N(v) \).

1) Analysis of ADG-M: We first investigate ADG-M.

Lemma 14. For a constant \( \varepsilon > 0 \), ADG-M can be implemented such that it has \( O(\log n) \) iterations, \( O(\log^2 n) \) depth, and \( O(n + m) \) work in the CRCW setting.

Proof. Since the algorithm removes at least half the vertices of \( U \) in each iteration \((\pm 1) \) vertex) we can conclude, as for the default version of ADG, that ADG-M performs \( O(\log n) \) iterations in the worst case. As in each iteration all operations can run in \( O(\log n) \) depth, ADG-M’s depth is \( O(\log^2 n) \).

Let \( k \) be the number of performed iterations, let \( U_i \) be the set \( U \) in iteration \( i \). To calculate work performed by ADG-M, we first consider the work in one iteration. Finding the median takes \( O(|U_i|) \) work. Since all other operations are the same as for ADG, we get again \( O((\sum_{v \in U_i} \deg(v)) + |U_i|) \) work in one iteration. As we also remove a constant number of vertices in each iteration from \( U \), we conclude, as with ADG, that ADG-M performs \( O(n + m) \) work over all iterations.

□

Lemma 15. ADG-M computes a partial 4-approximate degeneracy ordering of \( G \).

Proof. In each step \( \ell \) of ADG-M, only less than then half of the vertices in the induced subgraph \( G[U] \) can have a degree that is strictly larger than \( 2\delta_\ell \), where \( \delta_\ell \) is the average degree of vertices in this subgraph. Now, as we always remove the vertices with the lowest degree and as we remove half of the vertices \((\pm 1) \) from \( U \), all vertices that get removed have a degree of at most \( 2\delta_\ell \). From Lemma 3, we know that \( \delta \leq 2d \). Thus, each vertex has a degree of at most \( 4d \) in the subgraph \( G[U] \) (in the current step). Consequently, each vertex has at most \( 4d \) neighbors that are ranked equal or higher.

□

2) Analysis of JP-ADG-M: We now derive the bounds for JP-ADG-M. We proceed similarly as for JP-ADG. The partial approximate degeneracy ordering delivered by ADG-M is referred to as \( \rho_{\text{ADG-M}} \).

Corollary 2. JP-ADG-M colors a graph with at most \( 4d + 1 \) colors using the priority function \( \rho = (\rho_{\text{ADG-M}}, \rho_R) \).

Proof. Since \( (\rho_{\text{ADG-M}}, \rho_R) \) is a 4-approximate degeneracy ordering, this result follows from Lemma 6 and Lemma 15.

□

Corollary 3. JP-ADG-M colors a graph \( G \) with degeneracy \( d \) in expected depth \( O(\log^2 n + \log \Delta \cdot (d \log n + \log d \log^2 n)) \) and \( O(n + m) \) work in the CRCW setting.

Proof. Since \( \rho_{\text{ADG-M}} \) is a partial 4-approximate degeneracy ordering (Lemma 15) and since ADG-M performs at most \( O(\log n) \) iterations (Lemma 14), the depth follows from Lemma 7, Lemma 14 and past work [31], which shows that JP runs in \( O(\log n + \log \Delta \cdot |P|) \) depth. Since both JP and ADG-M perform \( O(n + m) \) work, the same holds for JP-ADG-M.

□

3) Analysis of DEC-ADG-M: The analysis of DEC-ADG-M is analogous to that of DEC-ADG, with the difference in that we use the ADG-M routine instead of ADG. The work and depth bounds remain the same as in DEC-ADG \((O(n + m) \)}
w.h.p. and $O(d \log^2 n)$), and the coloring bound – similarly to JP-ADG-M – increases from $(2 + \varepsilon)d$ to $(4 + \varepsilon)d$.

4) Impact of Optimizations: We also investigate the impact of optimizations described in § V-A, § V-B, and § V-C on the bounds of the associated routines (ADG-O, ADG-M-O, JP-ADG-O, JP-ADG-M-O). The key part is to illustrate that (1) moving “Part 1” of JP-ADG into ADG, and (2) the modified combined representation for $R$ and $U$ increase neither depth or work. This can be shown straightforwardly.

VI. EVALUATION

In evaluation, we found that empirical results follow theoretical predictions, already scrutinized in Table III and Section IV. Thus, for brevity, we now summarize the most important observations. A comprehensive comparison of run-times and coloring qualities of different algorithms is in Table III (together with abbreviations of used comparison baselines).

A. Methodology, Architectures, Parameters

We first provide the details of the evaluation methodology, used architectures, and considered parameters, to facilitate interpretability and reproducibility of experiments [89].

Used Architectures In the first place, we use Einstein, an in-house Dell PowerEdge R910 server with an Intel Xeon X7550 CPUs @ 2.00GHz with 18MB L3 cache, 1TiB RAM, and 32 cores per CPU (grouped in four sockets). We also conducted experiments on Ault (a CSCS server with Intel Xeon Gold 6140 CPU @ 2.30GHz, 768 GiB RAM, 18 cores, and 24.75MB L3) and Fulin (a CSCS server with Intel Skylake @ 2GHz, 1.8 TiB RAM, 52 cores, and 16MB L3).

Methodology We provide absolute runtimes when reporting speedups. In our measurements, we exclude the first measured 1% of performance data as warmup. We derive enough data to obtain the mean and 95% non-parametric confidence intervals. Data is summarized with arithmetic means.

Algorithms & Comparison Baselines We focus on modern heuristics from Table III. For each scheme, we always pick the most competitive implementation (i.e., fewest colors used need many re-coloring attempts, giving long tail run-times). For each scheme, we always pick the best configuration, consistently with past heuristics from Table III. For each scheme, we always pick the original heuristics from Table III. For each scheme, we always pick the best implementation (i.e., fewest colors used need many re-coloring attempts, giving long tail run-times). For each scheme, we always pick the best configuration, consistently with past heuristics from Table III. For each scheme, we always pick the best configuration, consistently with past heuristics from Table III. For each scheme, we always pick the best configuration, consistently with past heuristics from Table III. For each scheme, we always pick the best configuration, consistently with past heuristics from Table III. For each scheme, we always pick the best configuration, consistently with past heuristics from Table III. For each scheme, we always pick the best configuration, consistently with past heuristics from Table III.

The results are in Figure 1. Following past analyses [31], we consider separately two distinctive families of algorithms: those based on speculative coloring (SC), and the ones with the Jones and Plassman structure (color scheduling). These two classes of algorithms – especially for larger datasets – are often complementary, i.e., whenever one class achieves lower performance, the other thrives, and vice versa. This is especially visible for larger graphs, such as h-dsk, h-wb, or s-gmc. The reason is that the structure of some graphs (e.g., with dense clusters) entails many coloring conflicts which may need many re-coloring attempts, giving long tail run-times.

B. Summary of Insights

Our algorithms almost always offer superior coloring quality. Only JP-SL, JP-SLL (HP), and sometimes ITRB by Boman et al. [38] (Zoltan) use comparably few colors, but they are at least $1.5 \times$ and $2 \times$ slower, respectively. Simultaneously, run-times of our algorithms are comparable or marginally higher than the competition in the class of algorithms with speculative
Smaller graphs (used in online execution scheduling, ...)

Larger graphs (used in offline data analytics, ...)

Fig. 1: Run-times (1st and 3rd columns) and coloring quality (2nd and 4th columns). Two plots next to each other correspond to the same graph. Graphs are representative (other results follow similar patterns). Parametrization: 32 cores (all available), $\varepsilon = 0.01$, sorting: Radix sort, direction-optimization: push, JP-ADG variant based on average degrees $\delta$. SL and SLL are excluded from run-times in the right column (for larger graphs) because they performed consistently worse than others. We exclude DEC-ADG for similar reasons and because it is mostly of theoretical interest; instead, we focus on DEC-ADG-ITR, which is based on core design ideas in DEC-ADG. Numbers in bars for color counts are numbers of used colors. “SC”: results for the class of algorithms based on speculative coloring (ITR, DEC-ADG-ITR). “JP”: results for the class of algorithms based on the Jones and Plassman approach (color scheduling, JP). A vertical line in each plot helps to separate these two classes of algorithms. DEC-ADG-ITR uses dynamic scheduling. JP-ADG uses linear time sorting of $R$. Any schemes that are always outperformed in a given respect (e.g., Zoltan in runtimes or ColPack in qualities) are excluded from the plots.
tive coloring) and within at most 1.3-1.4 × of the competition (in the class of JP baselines). Thus, we offer the best coloring quality at the smallest required run-time overhead. Finally, our routines are the only ones with theoretical guarantees on work, depth, and quality.

C. Analysis of Run-Times with Full Parallelism

We analyze run-times using all the available cores. Whenever applicable, we show fractions due to reordering (pre-processing, e.g., the “ADG” phase in JP-ADG) and the actual coloring (e.g., the “JP” phase in JP-ADG). JP-SL, JP-SLL (HP), and JP-ASL (ColPack) are the slowest as they offer least parallelism. JP-LF, JP-LLF, and JP-R (GBBS/Ligra) are very fast, as their depth is in $O(\log n)$. We also analyze speculative coloring from ColPack and Zoltan; we summarize the most competitive variants. ITR does not come with clear bounds on depth, but its simple and parallelizable structure makes it very fast. ITRB schemes are $>2 \times$ slower than other baselines and are thus excluded from run-time plots. We also consider an additional variant of ITR based on ASL [82], ITR-ASL. In several cases, it approaches the performance of ITR.

The coloring run-times of JP-ADG are comparable to JP-LF, JP-LLF, and others. This is not surprising, as this phase is dominated by the common JP skeleton (with some minor differences from varying schedules of vertex coloring). However, the reordering run-time in JP-ADG comes with certain overheads because it depends on $\log^2 n$. This is expected, as JP-ADG – by its design – performs several sequential iterations, the count of which is determined by $\varepsilon$ (i.e., how well the degeneracy order is approximated). Importantly, JP-ADG is consistently faster (by more than $1.5 \times$) than JP-SL and JP-SLL that also focus on coloring quality.

DEC-ADG-ITR – similarly to JP-ADG – entails ordering overheads because it precomputes the ADG low-degree decomposition. However, total run-times are only marginally higher, and in several cases lower than those in ITR. This is because the low-degree decomposition that we employ, despite enforcing some sequential steps in preprocessing, reduces counts of coloring conflicts, translating to performance gains.

D. Analysis of Coloring Quality

Coloring quality also follows the theoretical predictions: JP-SL outperforms JP-SLL, JP-LF, and JP-LLF (by up to 15%), as it strictly follows the degeneracy order. Overall, all four schemes (GBBS/Ligra, HP) are competitive. As expected, JP-FF and JP-R come with much worse coloring qualities because they do not focus on minimizing color counts. As observed before [31], ITR (ColPack) outperforms JP-FF and JP-R but falls behind JP-LF, JP-LLF, JP-SL, and JP-SL. JP-ASL and ITR-ASL (ColPack) offer low (often the lowest) quality. ITRB (Zoltan) sometimes approaches the quality of JP-SL, JP-SLL, DEC-ADG-ITR, and JP-ADG.

The coloring quality of our schemes outperforms others in almost all cases. Only JP-SL, JP-SLL (GBBS/Ligra, HP), and sometimes ITRB (Zoltan) are competitive, but they are always much slower. In some cases, JP-ADG (e.g., in s-ork) and DEC-ADG-ITR (e.g., in s-gmc) are better than JP-SL and JP-SLL (by 3-10%). Hence, while the strict degeneracy order is in general beneficial when scheduling vertex coloring, it does not always give best qualities. JP-ADG consistently outperforms others, reducing used color counts by even up to 23% compared to JP-LLF (for m-wta). Finally, DEC-ADG-ITR always ensures much better quality than ITR, up to 40% (for s-lib). Both DEC-ADG-ITR and JP-ADG offer similarly high coloring qualities across all comparison targets.

E. Analysis of Strong Scaling

We also investigate strong scaling (i.e., run-times for the increasing thread counts). Relative performance differences between baselines do not change significantly, except for SLL that becomes more competitive when the thread count approaches 1, due to the tuned sequential implementation that we used [31]. Representative results are in Figure 2; all other graphs result in analogous performance patterns. Most variants from ColPack, Zoltan, GBBS/Ligra, and HP scale well (we still exclude Zoltan due to high runtimes). Importantly, scaling of our baselines is also advantageous and comparable to others. This follows theoretical predictions, as the $\log^2 n$
factor in our depth bounds is alleviated by the presence of the degeneracy $d$ (or $\log d$) instead of $\Delta$, as opposed to the competition; see § IV-E on page 7 for details.

F. Analysis of Weak Scaling

Weak scaling is also shown in Figure 2. We use Kronecker graphs [101] of the increasing sizes by varying the number of edges/vertex; this fixes the used graph model. JP-ADG scales comparably to other JP baselines; DEC-ADG-ITR scales comparably or better than ITR or ITR-ASL.

G. Analysis of Impact from $\varepsilon$

Representative results of the impact of $\varepsilon$ are in Fig. 3. As expected, larger $\varepsilon$ offers more parallelism and thus lower runtimes, but coloring qualities might decrease. Importantly, the decrease is minor, and the qualities remain the highest or competitive across almost the whole spectrum of $\varepsilon$.

H. Memory Pressure and Idle Cycles

We also investigate the pressure on the memory bus, see Figure 4. For this, we use PAPI [102] to gather data about idle CPU cycles and L3 cache misses. Low ratios of L3 misses or idle cycles indicate high locality and low pressure on the memory bus. Overall, our routines have comparable or best ratios of active cycles and L3 hits.

I. Performance Profiles

We also summarize the results from Figure 1 using performance profiles [103], see Figure 5 for a representative profile for coloring quality. Details on using performance profiles are in the extended report due to space constraints; intuitively, such a profile shows cumulative distributions for a selected performance metric (e.g., a color count). The summary in Figure 5 confirms the previous insights: DEC-ADG-ITR, JP-ADG, and JP-SL offer the best colorings.

J. Additional Analyses of Design Choices

We also analyze variants of JP-ADG and DEC-ADG-ITR, considering sorting of set $R$, using static vs. dynamic schedules and other aspects from Section V (push vs. pull, median vs. average degree, different sorting algorithms). In Figure 1, we use JP-ADG with counting sorting of $R$ and DEC-ADG-ITR with dynamic scheduling. All these design choices have a certain (usually up to 10% of relative difference) impact on performance, but (1) it strongly depends on the input graph, and (2) does not change fundamental performance patterns.

VII. Related Work

One of the first methods for computing a $(\Delta + 1)$-coloring in parallel for general graphs arose from the parallel polylogarithmic-depth maximal independent set (MIS) algorithm by Karp and Wigderson [29], further improved to $O(\log n)$ depth by a simpler MIS algorithm in the influential paper by Luby [30] The key idea in this simple coloring strategy is to (1) find a MIS $S$, (2) apply steps of Greedy in parallel for all vertices in $S$ (which is possible as, by definition, no two vertices in a MIS are adjacent), coloring all vertices in $S$ with a new color, (3) remove the colored vertices from $G$, and (4) repeat the above steps until all vertices are colored.

Karp and Wigderson’s, and Luby’s algorithms started a large body of parallel graph coloring heuristics [1]–[3], [9]–[11], [25]–[31], [33], [37], [77]–[81], [104]. We already exhaustively analyzed them in Section I and in Table III. Almost all of them have theoretical guarantees based on the work-depth [105] or the PRAM model [106]. We build on and improve on these works in several dimensions, as explained in detail in Section I.

Many works exist in the theory of distributed graph coloring based on models such as LOCAL or CONGESTED-CLIQUE [107]–[121]. These algorithms are highly theoretical and do not come with any implementations. Moreover, they come with assumptions that are unrealistic in HPC settings, for example distributed LOCAL and CONGEST algorithms do not initially know the interconnection graph, or the message size in LOCAL algorithms can be unbounded. Finally, they cannot be directly compared to Work-Depth or PRAM algorithms. Thus, they are of little relevance to our work.

Many practical parallel and distributed approaches were proposed. They often use different speculative schemes [34]–[46], where vertices are colored speculatively and potential conflicts are resolved in a second pass. Some of these schemes...
were implemented within frameworks or libraries [82], [122]–[124]. Another line of schemes incorporates GPUs and vectorization [41], [44], [45], [125]–[129]. Other schemes use recoloring [130], [131] in which one improves an already existing coloring. Patidar and Chakrabarti use Hadoop to implement graph coloring [132]. Alabandi et al. [133] illustrate how to increase parallelism of coloring heuristics. These works are orthogonal to this paper: they do not provide theoretical analyses, but they usually offer numerous architectural and design optimizations that can be combined with our algorithms for further performance benefits. As we focused on theoretical guarantees and its impact on performance, and not on architecture-related optimizations, we leave integrating these optimizations with our algorithms as future work.

There are works on coloring specific graph classes, such as planar graphs [134]–[137]. Some works impose additional restrictions, for example coloring balance, which limits differences between numbers of vertices with different colors [138]–[140]. Other lines of related work also exist, for example on edge coloring [141], dynamic or streaming coloring [142]–[148], k-distance-coloring and other generalizations [149]–[151], and sequential exact coloring [152]–[154]. There are even works on solving graph coloring with evolutionary and genetic algorithms [155]–[157] and with machine learning methods [158]–[163]. All these works are unrelated as we focus on unrestricted, parallel, and 1-distance vertex coloring with provable guarantees on performance and quality, targeting general, static, and simple graphs.

The general structure of our ADG algorithm, based on iteratively removing vertices with degrees in certain ranges defined by the approximation parameter $\varepsilon$, was also used to solve other problems, for example the $(2 + \varepsilon)$-approximate maximal densest subgraph algorithms by Dhulipala et al. [61]. Finding more applications of ADG is left for future work.

We note that, while our ADG scheme is the first parallel algorithm for deriving approximate degeneracy ordering with a provable approximation factor, two algorithms in the streaming setting exist [164], [165].

Graph coloring has been targeted in several recent works related to broad graph processing paradigms, abstractions, and frameworks [36], [166]–[173]. Several HPC works [174]–[176] consider distributed graph coloring in the context of high-performance RDMA networks and RMA programming [177]–[182]. Different coloring properties of graphs were also analyzed in the context of graph compression and summarization [183], [184].

VIII. CONCLUSION

We develop graph coloring algorithms with strong theoretical guarantees on all three key aspects of parallel graph coloring: work, depth, and coloring quality. No other existing algorithm provides such guarantees.

One algorithm, JP-ADG, is often superior in coloring quality to all other baselines, even including the tuned SL and SLL algorithms specifically designed to reduce counts of used colors [31]. It also offers low run-times for different input graphs. As we focus on algorithm design and analysis, one could combine JP-ADG with many orthogonal optimizations, for example in the GPU landscape, to achieve more performance without sacrificing quality. Another algorithm, DEC-ADG, is of theoretical interest as it is the first routine – in a line of works based on speculative coloring – with strong theoretical bounds. While being less advantageous in practice, we use its underlying design to enhance a recent coloring heuristic [40] obtaining DEC-ADG-ITR, an algorithm with (1) strong quality bounds and (2) competitive performance, for example up to 40% fewer colors used then compared to the base design [40].

Our algorithms use a very simple (but rich in outcome) idea of provably relaxing the strict vertex degeneracy order, to maximize parallelism when deriving this order. This idea, and our corresponding parallel ADG algorithm, are of separate interest, and could enhance other algorithms that rely on vertex ordering, for example in mining maximal cliques [49], [50].
We provide the most extensive theoretical study of parallel graph coloring algorithms. This analysis can be used by other researchers as help in identifying future work.

Acknowledgements: We thank Hussein Harake, Colin McMurtie, Mark Klein, Angelo Mangili, and the whole CSCS team granting access to the Ault and Daint machines, and for their excellent technical support. We thank Timo Schneider for his immense help with computing infrastructure at SPCL.

REFERENCES

[1] T. F. Coleman and J. J. Moré, “Estimation of sparse jacobian matrices and graph coloring problems,” SIAM Journal on Numerical Analysis, vol. 20, no. 1, pp. 187–209, 1983.

[2] M. T. Jones and P. E. Plassmann, “Scalable iterative solution of sparse linear systems,” Parallel Computing, vol. 20, no. 3, pp. 753–773, 1994.

[3] A. H. Gebremedhin, F. Manne, and A. Pothen, “What color is your jacobian? graph coloring for computing derivatives,” SIAM Review, vol. 47, no. 4, pp. 629–705, 2005.

[4] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoeﬄer, and E. Solomonik, “Communication-eﬃcient jaccard similarity for high-performance distributed genome comparisons,” IEEE IPDPS, 2020.

[5] G. J. Besta, F. Marending, E. Solomonik, and T. Hoeﬄer, “SlimSell: A Vectorized Graph Representation for Breadth-First Search,” in Proceedings of the 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS’17). IEEE, May 2017.

[6] G. Kwasniewski, M. Kabici, M. Besta, J. VandeVondele, R. Solcà, and T. Hoeﬄer, “Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–2.

[7] E. Solomonik, M. Besta, F. Vella, and T. Hoeﬄer, “Scaling between centrality using communication-efﬁcient sparse matrix multiplication,” in ACM/IEEE Supercomputing, 2017, p. 47.

[8] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson, “Executing dynamic data-graph computations deterministically using chromatistic scheduling,” ACM Transactions on Parallel Computing (TOPC), vol. 3, no. 1, pp. 1–31, 2016.

[9] E. M. Arkin and E. B. Silverberg, “Scheduling jobs with ﬁxed start and end times,” Discrete Applied Mathematics, vol. 18, no. 1, pp. 1–8, 1987.

[10] D. Marx, “Graph colouring problems and their applications in scheduling,” Periodica Polytechnica Electrical Engineering (Archives), vol. 48, no. 1-2, pp. 11–16, 2004.

[11] R. Ramaswami and K. Parhi, “Distributed scheduling of broadcasts in a radio network,” in Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies, ser. IEEE INFOCOM ’89. IEEE, 1989.

[12] D. Ghrab, B. Derbel, I. Jemili, A. Dhraief, A. Belghith, and E.-G. Talbi, “Coloring based hierarchical routing approach,” 2013.

[13] G. Li and R. Simha, “The partition coloring problem and its application to wavelength routing and assignment,” in Proceedings of the First Workshop on Optical Networks. Citeseer, 2000, p. 1.

[14] M. Besta, S. M. Hassan, S. Yalamanchi, R. Ausavarungnirun, O. Mutlu, and T. Hoeﬄer, “Slim noc: A low-diameter on-chip network topology for high energy eﬃciency and scalability,” in ACM SIGPLAN Notices, 2018.

[15] M. Besta and T. Hoeﬄer, “Slim Fly: A Cost Eﬀective Low-Diameter Network Topology.” Nov. 2014, aCM/IEEE Supercomputing.

[16] S. Di Girolamo, K. Tararan, A. Kurth, M. Schaffner, T. Schneider, J. Beránek, M. Besta, L. Benini, D. Roweth, and T. Hoeﬄer, “Network-accelerated non-contiguous memory transfers,” arXiv preprint arXiv:1908.05850, 2019.

[17] M. Besta, M. Schneider, K. Cynk, M. Konieczny, E. Henriksson, S. Di Girolamo, A. Singla, and T. Hoeﬄer, “Fatpaths: Routing in supercomputers and data centers that shortest paths fall short,” ACM/IEEE Supercomputing, 2020.

[18] M. Besta, J. Domke, M. Schneider, M. Konieczny, S. Di Girolamo, T. Schneider, A. Singla, and T. Hoeﬄer, “High-performance routing with multipathing and path diversity in supercomputers and data centers,” arXiv preprint arXiv:2007.03776, 2020.

[19] M. Javedankherad, Z. Zeinalpour-Yazdi, and F. Ashitani, “Content placement in cache networks using graph coloring,” IEEE Systems Journal, 2020.

[20] A. Dey and A. Pal, “Fuzzy graph coloring technique to classify the accidental zone of a traffic control,” Annals of Pure and Applied Mathematics, vol. 3, no. 2, pp. 169–178, 2013.

[21] G. J. Chatin, “Register allocation & splicing via graph coloring,” ACM Sigplan Notices, vol. 17, no. 6, pp. 98–101, 1982.

[22] J. de Fine Licht et al., “Transformations of high-level synthesis codes for high-performance computing,” arXiv:1805.08288, 2018.

[23] R. Lewis, A guide to graph colouring. Springer, 2015, vol. 7.

[24] M. R. Garey, D. S. Johnson, and L. Stockmeyer, “Some simplified NP-complete problems,” in Proceedings of the sixth annual ACM symposium on Theory of computing, ser. STOC’74. ACM, 1974, pp. 47–63.

[25] D. J. A. 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.

[26] M. T. Jones and P. E. Plassmann, “A parallel graph coloring heuristic,” SIAM Journal on Scientific Computing, vol. 14, no. 3, pp. 654–669, 1993.

[27] D. Brézaz, “New methods to color the vertices of a graph,” Communications of the ACM, vol. 22, no. 4, 1979.

[28] D. W. Matula and L. L. Beck, “Smallest-last ordering and clustering and graph coloring algorithms,” Journal of the ACM, vol. 30, no. 3, pp. 417–427, 1983.

[29] R. M. Karp and W. Avi, “A fast parallel algorithm for the maximal independent set problem,” JACM, vol. 32, no. 4, pp. 762–773, 1985.

[30] M. Luby, “A simple parallel algorithm for the maximal independent set problem,” SIAM journal on computing, vol. 15, no. 4, p. 1036–1053, 1986.

[31] 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. ACM, 2014, pp. 166–177.

[32] M. M. A. Patwary, A. H. Gebremedhin, and A. Pothen, “New multi-threaded ordering and coloring algorithms for multicore architectures,” in Euro-Par 2011 Parallel Processing. E. Jeannot, R. Namyst, and J. Roman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 250–262.

[33] J. Allwright, R. Bordawekar, D. Coddington, K. Dincer, and C. L. Martin, “A comparison of parallel Graph coloring algorithms,” 1995.

[34] U. V. Çatalyürek, F. Dobrian, A. Gebremedhin, M. Halappanavar, and A. Pothen, “Distributed-memory parallel algorithms for matching and coloring,” in 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. IEEE, 2011, pp. 1971–1980.

[35] D. Bozdag, 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] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoeﬄer, “To push or to pull: On reducing communication and synchronization in graph computations,” in Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, 2017, pp. 93–104.

[37] A. H. Gebremedhin and F. Manne, “Scalable parallel graph coloring algorithms,” Concurrerncy: Practice and Experience, vol. 12, no. 12, pp. 85–120, 2000.

[38] E. G. Boman, D. Bozdag, U. Catalyurek, A. H. Gebremedhin, and F. Manne, “A scalable parallel graph coloring algorithm for distributed memory computers,” in European Conference on Parallel Processing. Springer, 2005, pp. 241–251.

[39] A. H. Gebremedhin, I. G. Lassous, J. Gusted, and J. A. Telle, “Graph coloring on a coarse grained multiprocessor,” in International Workshop on Graph-Theoretic Concepts in Computer Science. Springer, 2000, pp. 184–195.

[40] U. 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.

[41] E. Saule and U. V. Çatalyürek, “An early evaluation of the scalability of graph algorithms on the intel mic architecture,” in 2012 IEEE
[184] M. Besta and T. Hoefer, “Survey and taxonomy of lossless graph compression and space-efficient graph representations,” arXiv preprint arXiv:1806.01799, 2018.