Greed is Good: Parallel Algorithms for Bipartite-Graph Partial Coloring on Multicore Architectures

Mustafa Kemal Taş
Computer Science and Engineering, Sabancı University, Istanbul, Turkey
mkemaltas@sabanciuniv.edu

Kamer Kaya
Computer Science and Engineering, Sabancı University, Istanbul, Turkey
Dept. Biomedical Informatics, The Ohio State University, Columbus, USA
kaya@sabanciuniv.edu

Erik Saule
Computer Science, The University of North Carolina at Charlotte, Charlotte, NC, USA
esaule@uncc.edu

***Abstract*—In parallel computing, a valid graph coloring yields a lock-free processing of the colored tasks, data points, etc., without expensive synchronization mechanisms. However, coloring is not free and the overhead can be significant. In particular, for the bipartite-graph partial coloring (BGPC) and distance-2 graph coloring (D2GC) problems, which have various use-cases within the scientific computing and numerical optimization domains, the coloring overhead can be in the order of minutes with a single thread for many real-life graphs.**

In this work, we propose parallel algorithms for bipartite-graph partial coloring on shared-memory architectures. Compared to the existing shared-memory BGPC algorithms, the proposed ones employ greedier and more optimistic techniques that yield a better parallel coloring performance. In particular, on 16 cores, the proposed algorithms are more than $4\times$ faster than their counterparts in the `ColPack` library which is, to the best of our knowledge, the only publicly-available coloring library for multicore architectures. In addition to BGPC, the proposed techniques are employed to devise parallel distance-2 graph coloring algorithms and similar performance improvements have been observed. Finally, we propose two costless balancing heuristics for BGPC that can reduce the skewness and imbalance on the cardinality of color sets (almost) for free. The heuristics can also be used for the D2GC problem and in general, they will probably yield a better color-based parallelization performance especially on many-core architectures.

***Keywords*-Greedy graph coloring; bipartite-graph coloring; distance-2 coloring; shared-memory parallel algorithms.**

I. Introduction

A coloring on a graph $G = (V, E)$ explicitly partitions the vertices in V into a number of disjoint subsets such that two vertices $u, v \in V$ that are in the same color set are independent from each other, i.e., $(u, v) \notin E$. Graphs have been frequently used to model data, e.g., matrices and tensors, as well as computations. In these models, two neighbor vertices usually imply a potential race-condition in a parallel execution. On the other hand, given a valid coloring on V, each color set, formed by independent vertices, can be simultaneously processed in a lock-free manner and without a synchronization overhead. Moreover, in practice, a *good* coloring with a small number of colors will probably yield a better performance compared to a *bad* coloring with a large number of colors since there will be fewer barriers (between the color sets), and hence the parallelization overhead will be reduced. Unfortunately, *the distance-1 graph coloring problem* (D1GC), i.e., coloring a graph with the minimum number of colors such that all adjacent vertices have different colors, is NP-Complete and hard to approximate [1], [2].

The traditional adjacency-based neighborhood is not suitable for numerous applications such as efficient computation of Hessians and Jacobians. Instead, the problem can be modeled as a *bipartite graph partial-coloring* (BGPC) problem. In BGPC, given a bipartite graph $G = (V_A \cup V_B, E)$, one wants to color the vertices in V_A with a minimum number of colors, such that all vertex pairs that are adjacent to at least one V_B vertex have different colors. A similar problem is *distance-2 graph coloring* (D2GC), where a graph is colored in a way that the color of each vertex is different than the colors of the vertices in its distance-2 neighborhood. For more details on the applications of BGPC and D2PC and parallel algorithms to solve these problems on shared-memory and distributed-memory architectures, we refer the reader to [3], [4], [5], [6], [7], [8].

From the parallel computing perspective, another desirable property of a good coloring is the *balance* on the color set cardinalities [9], [10], [11], [12]; a more balanced coloring can improve the convergence speed and the value of the final objective function for some iterative algorithms. However, a tight balance is not required if shared-memory parallelism is the only concern; if all the color set cardinalities are above a certain threshold, that depends on the number of processors/cores available and the task heterogeneities, the parallel performance will not be disrupted by the remaining imbalance since there will be enough work to feed all the available cores/processors.

Good colorings are not free and their generation adds an overhead for parallelization. Furthermore, the impact of this overhead increases if the coloring is performed sequentially and the actual job is executed on a large number of cores. This is why parallelization of graph coloring algorithms have been extensively studied for all the problems above, e.g., [3], [6], [13], [14]. The results in the literature show the execution time of a sequential D1GC algorithm is less than

a second for many real-life graphs. However, for D2GC and BGPC, the overhead can be in the order of minutes.

The contribution of this paper is three-fold: **1)** We propose parallel BGPC algorithms on multicore architectures that employ greedier and more optimistic techniques compared to the existing algorithms. We compared the performance of the proposed algorithms with the one in the `ColPack` library which, to the best of our knowledge, is the only publicly-available coloring library with a parallel BGPC implementation. On the average for eight UFL matrices and with 16 threads/cores, $1.47\times$ speedup is obtained via basic optimizations, another $2.81\times$ speedup is obtained by employing faster and more optimistic techniques without a significant increase on the number of colors. Overall, the proposed algorithm is $4.71\times$ faster than the parallel `ColPack` implementation on 16 threads and uses only 8% more colors. **2)** We applied the same techniques for the D2GC problem and observed similar speedups on the five of eight, square, structurally symmetric matrices in our test-bed. **3)** We integrated two online heuristics to the proposed BGPC algorithms that aim to balance the color set cardinalities during the course of the coloring without a significant overhead: The first heuristic tries not to increase the number of colors, whereas the second one aggressively improves the balance by using more colors (only 11% more on average for the eight graphs in our experiments).

For the experiments, we used `ColPack` as the main baseline: almost all the literature use algorithms which are less optimistic than the ones proposed in this work. On various machines, such as GPUs and distributed-memory systems, these algorithms have been improved with architectural insights. For multicore processors, `ColPack` is considered as a state-of-the-art coloring library and the source is available for modification. Based on this rationale, we put the proposed techniques on top of `ColPack`. We even fine-tuned the performance of existing less-optimistic variants for fairness. This makes all the algorithms use the same codebase and we are sure that the performance difference comes only from the proposed, more optimistic decisions.

The rest of the paper is organized as follows: Section II introduces the notation and background on parallel coloring algorithms. The proposed BGPC algorithms are described in detail in Section III and their adaptation for D2GC is presented in Section IV. The balancing heuristics are described in Section V. Section VI presents the experimental results and Section VII briefly surveys the related coloring literature. Section VIII concludes the paper.

II. Background and Notation

Most of the recent coloring algorithms use a speculative, iterative approach which first colors the vertices optimistically in parallel hoping that a valid coloring will be generated, e.g., [3], [13], [14], [15]. The validity of the coloring is then verified in a conflict removal step; if a conflict, i.e., *a pair of neighbor vertices with the same color*, is detected, one of the vertices is tagged to be colored in the next iteration. Let $G = (V, E)$ be a graph and let $V_{color} \subseteq V$ be the vertices that need to be colored. Let $\texttt{nbor}(v) \subset V_{color}$ define the neighborhood structure of the vertices to be colored. Throughout the text, non-negative integers will be used for coloring and **-1** is used for an uncolored vertex. A pseudocode of the greedy optimistic graph coloring approach is given in Algorithms 1, 2 and 3.

Algorithm 1 GreedyGraphColoring

Input: $G = (V, E)$, $V_{color} \subseteq V$: vertices to be colored, `nbor`(.): the neighborhood function for the vertices in V_{color}.
Output: c[.]: a valid coloring array for V_{color}

```
1: W ← V_color
2: c[v] ← −1, ∀v ∈ V_color
3: while W is not empty do
4:     c ← ColorWorkQueue(G, W, c)
5:     W ← RemoveConflicts(G, W, c)
```

Algorithm 2 ColorWorkQueue

Input: $G = (V, E)$, W: vertices to color, `nbor`(.): the neighborhood function, c[.]: an incomplete coloring with no conflicts.
Output: c[.]: an optimistic coloring.

```
1: for each w ∈ W in parallel do
2:     F ← ∅                          ▷ thread private forbidden color set for w
3:     for each u ∈ nbor(w) do
4:         if c[u] ≠ −1 then
5:             F ← F ∪ {c[u]}
6:     col ← 0                        ▷ first-fit coloring policy
7:     while col ∈ F do
8:         col ← col + 1
9:     c[w] ← col
```

Algorithm 3 RemoveConflicts

Input: $G = (V, E)$: the graph to color, W: vertices to color, `nbor`(.): the neighborhood function, c[.]: an optimistic coloring.
Output: W_{next}: the work queue for next iteration, c[.]: a (probably incomplete) coloring with no conflicts.

```
1: W_next ← ∅                         ▷ a shared queue for the next iter.
2: for each w ∈ W in parallel do
3:     for each u ∈ nbor(w) do
4:         if c[u] = c[w] and w > u then
5:             W_next ← W_next ∪ {w} : atomic
6:             break
```

As the algorithms show, at each iteration, a set of vertices in W are optimistically colored. A conflict removal phase is then performed to check if they are conflicting with the other vertices in V_{color}. When conflicts are detected, the *conflicting vertices* are added to the next iteration's vertex queue and the procedure is repeated. This greedy and optimistic approach can be used for almost all the coloring variants and the definitions of V_{color} and `nbor`(.) change with respect to the problem. For the BGPC problem on a bipartite graph $G = (V, E)$ where $V = V_A \cup V_B$ has two parts, $V_{color} = V_A$ and for each $u \in V_A$, $\texttt{nbor}(u)$ is defined as $\{v \in V_A \setminus \{u\} : \exists w \in V_B \text{ s.t. } (u, w) \in E \text{ and } (v, w) \in E\}$. For D2GC, $V_{color} = V$ and $\texttt{nbor}(u)$ is the set of vertices in V whose shortest-path distances to u are less than or equal to two.

The BGPC problem can be considered as a hypergraph coloring problem [6] where the elements of V_A correspond to the *pins* to be colored, and the ones in V_B correspond to the *nets* in the hypergraph which define the neighborhood. Based on this analogy, for clarity, while describing our BGPC algorithms we will use the terms *vertex* and *net* to denote a V_A and V_B vertex, respectively, in the bipartite graph. Similarly, for a vertex $u \in V_A$ ($v \in V_B$), $\texttt{nets}(u)$ ($\texttt{vtxs}(v)$) will denote the set of V_B (V_A) vertices adjacent to u (v).

For the BGPC problem, the value $\max_{v \in V_B}(|\texttt{vtxs}(v)|)$ is a trivial lower bound on the number of colors required for a valid coloring since all the vertices in each $\texttt{vtxs}(.)$ set need to be colored with distinct colors. The counterpart of this bound for the D2GC variant is $1 + \max_{v \in V}(|\texttt{nbor}(v)|)$.

III. Algorithms for bipartite-graph coloring

In BGPC, both coloring and conflict removal phases can be performed in two ways: *vertex-based* and *net-based*. The existing literature on shared-memory bipartite-graph partial coloring algorithms follow the former approach. However, net-based coloring can be more efficient since the neighborhood single-handedly defines the validity of the coloring. Furthermore, depending on the iteration number and the size of the current work queue W, i.e., the number of remaining vertices to be colored, this approach can be more efficient.

The vertex-based BGPC approach, which is employed by the `ColPack` library, traverses the neighborhood starting from the vertices to be colored both for ColorWorkQueue and RemoveConflicts as shown in Algorithms 4 and 5, respectively.

Algorithm 4 BGPC-ColorWorkQueue-Vertex

Input: $G = (V_A \cup V_B, E)$: a bipartite graph, W: vertices to color, $c[.]$: an incomplete coloring with no conflicts.
Output: $c[.]$: an optimistic coloring.

```
1: for each w ∈ W in parallel do
2:     F ← ∅ : thread private forbidden color set for w
3:     for each v ∈ nets(w) do
4:         for each u ∈ vtxs(v) \ {w} do
5:             if c[u] ≠ −1 then
6:                 F ← F ∪ {c[u]}
7:     ...                          ▷ first-fit coloring (lines 6-9 in Alg. 2)
```

Algorithm 5 BGPC-RemoveConflicts-Vertex

Input: $G = (V_A \cup V_B, E)$, W: vertices to color, $\texttt{nbor}(.)$: the neighborhood function, $c[.]$: an optimistic coloring.
Output: W_{next}: the work queue for next iteration, $c[.]$: a (probably incomplete) coloring with no conflicts.

```
1: W_next ← ∅ : a shared queue for the next iter.
2: for each w ∈ W in parallel do
3:     for each v ∈ nets(w) do
4:         for each u ∈ vtxs(v) \ {w} do
5:             ...                  ▷ detect conflicts (lines 4-6 in Alg. 3)
```

For BGPC-ColorWorkQueue-Vertex, the vertex-based approach needs to go over all the vertices in V_{color} in the first iteration. That is each net $v \in V_B$ will be visited $\texttt{vtxs}(v)$ times and for each visit, all $\texttt{vtxs}(v)$ edges will be processed; hence, the complexity of the neighborhood traversal in the first iteration is $\Theta\left(\sum_{v \in V_B} |\texttt{vtxs}(v)|^2\right)$. The complexity of the conflict removal phase for the first iteration is also $\mathcal{O}\left(\sum_{v \in V_B} |\texttt{vtxs}(v)|^2\right)$. Although there can be early terminations (line 6 of Alg. 3), this worst-case bound is tight; if the optimistic coloring is valid, the neighborhood needs to be traversed for each vertex in $V_A = V_{color}$ in the first conflict removal phase. Unfortunately, for many BGPC use cases, such as numerical optimization, there can be V_B nets having tens of thousands of adjacent vertices. These nets will be problematic while coloring a bipartite graph especially for the first iteration that dominates the overall execution time according to our experience.

In net-based coloring, the vertices are colored by observing the neighborhood from the nets' side; in BGPC, a conflict is created when "two vertices in the same $\texttt{vtxs}$ set are colored with the same color". Hence, the net-based approach sounds more natural for coloring. The coloring and conflict removal phases of the most straightforward and the most optimistic net-based BGPC are given in Algorithms 6 and 7, respectively.

Algorithm 6 BGPC-ColorWorkQueue-Net-v1

Input: $G = (V_A \cup V_B, E)$: a bipartite graph.
Output: $c[.]$: the (most) optimistic coloring array.

```
1: for each v ∈ V_B in parallel do
2:     F ← ∅ : thread private forbidden color set for v
3:     col ← 0 : thread private
4:     for each u ∈ vtxs(v) do
5:         if c[u] = −1 or c[u] ∈ F then
6:             while col ∈ F do
7:                 col ← col + 1
8:             c[u] ← col
9:         F ← F ∪ {c[u]}
```

Algorithm 7 BGPC-RemoveConflicts-Net

Input: $G = (V_A \cup V_B, E)$: a bipartite graph to color, $c[.]$: an optimistic coloring.
Output: $c[.]$: an incomplete coloring.

```
1: for each v ∈ V_B in parallel do
2:     F ← ∅ : thread private forbidden color set for v
3:     for each u ∈ vtxs(v) do
4:         if c[u] ≠ −1 then
5:             if c[u] ∈ F then
6:                 c[u] ← −1
7:             else
8:                 F ← F ∪ {c[u]}
```

The net-based coloring in Algorithm 6 processes the nets in parallel to color the vertices in the adjacency lists. The complexity of each iteration is linear in terms of the size of the graph ($|V_A \cup V_B| + |E|$). However, while coloring, each thread only checks the local conflicts within the neighborhood of the current net's adjacency; this is the optimism. When a vertex u is visited (line 4), the thread first checks the value of $c[u]$. If $c[u]$ is not set yet (or set to **-1** in the previous conflict removal phase), or if $c[u]$ has

Matrix-Graph	$\|V_B\|$	Remaining $\|W_{next}\|$ after the first iteration		
		Alg. 6	Alg. 6 + reverse	Alg. 8
bone010	986,703	863,785	806,264	610,924
coPapersDBLP	540,486	409,621	303,152	133,874

Table I: The number of uncolored (remaining) vertices after the first iteration for two graphs, obtained from matrices bone010 and coPapersDBLP, when Algorithms 6 and 8 are used on 16 threads.

been used for the current net before, u is recolored (line 8). While doing that, Algorithm 6 imitates a net-level first-fit coloring (lines 6–8) for the visited vertices. This is the most optimistic net-based coloring since the threads "hope" that they are using a color in the earlier positions of the adjacency list which will not appear later positions. Unfortunately, our preliminary experiments show that this level of optimism is maleficent due to the large number of conflicts it incurs.

Although the net-based approach is not straightforward to employ for the coloring phase, it suits much better for the conflict removal phase; a net-based traversal given in Algorithm 7 is sufficient to detect all the existing conflicts. Moreover, unlike its vertex-based variant, the complexity of an iteration is linear in terms of the graph size. One drawback is that it may remove more colorings than required compared to vertex-based approach. However, we did not observe a significant performance reduction due to this optimism of net-based conflict detection.

To keep the coloring process in the right track by reducing the number of conflicts, we propose a less optimistic version of BGPC-ColorWorkQueue-Net-v1 as in Algorithm 8. There are two main modifications: first, to reduce the number of re-colorings within the adjacency list of a single net, the algorithm first performs a pass on the adjacency list and marks the forbidden colors (the for loop at line 4). While doing that, it also stores the vertices that need to be colored in a thread private queue W_{local} (line 8). After the first pass, the vertices in W_{local} are visited and colored one-by-one.

The second modification is applied while coloring these vertices; instead of using a first-fit policy that uses the smallest possible color for a vertex, we employ a reverse first-fit policy (lines 9–14) that uses the largest possible color smaller than $|\texttt{vtxs}(v)|$ while coloring the vertices in W_{local}. This policy never uses a negative color since there are at most $|\texttt{vtxs}(v)|$ vertices in W_{local} and $|\texttt{vtxs}(v)|$ colors can still be used for them. Besides, since $|\texttt{vtxs}(v)|$ is a lower-bound on the number of colors used, we do not expect a large increase on the number of colors used. Moreover, reverse first-fit is expected to produce less number of conflicts compared to the first-fit in Algorithm 6, since it does not always use the same small colors but prioritize different colors for each net. To understand the benefits of these modifications better, we refer the reader to Table I where the number of colored (remaining) vertices after the first iteration is presented for two graphs when Algorithms 6 and 8 are employed.

A drawback of the net-based conflict detection is the need of traversing all the nets for all iterations. For the vertex-based approach, it is sufficient to visit only the neighborhood of the vertices colored at the current iteration. However, without an intelligent net-marking technique in the coloring phase, it is not easy to restrict the neighborhood that needs to be traversed to identify all the conflicts. Hence, net-based conflict removal can be much faster than the vertex-based variant for the first few iterations. Although it can make the performance even worse for later iterations, in our experiments, 78% of the runtime is observed to be used on the first iteration. That number goes up to 89% for the first two iterations on average for eight graphs we used. Thus, attacking these first iterations would be enough.

Figure 1 shows the execution times of each iteration of different algorithms while coloring coPapersDBLP with 16 threads. In the figure, an algorithm X-Y applies X-based coloring and Y-based conflict removal where the letter V and N denote vertex- and net-based, respectively. A number n adjacent to the letter N denotes that the algorithm performs net-based approach for the first n iterations and switch to the vertex-based approach. A more detailed explanation of the algorithms is given in Section VI. The figure tells that: 1) most of the time is spent for the coloring, 2) most of the time

Algorithm 8 BGPC-ColorWorkQueue-Net

Input: $G = (V_A \cup V_B, E)$: a bipartite graph, $c[.]$: an incomplete coloring.
Output: $c[.]$: an optimistic coloring array.

```
1:  for each v ∈ V_B in parallel do
2:      F ← ∅ : thread private forbidden color set for v
3:      W_local ← ∅ : thread private vertices to be colored
4:      for each u ∈ vtxs(v) do
5:          if c[u] ≠ −1 and c[u] ∉ F then
6:              F ← F ∪ {c[u]}
7:          else
8:              W_local ← W_local ∪ {u}
9:      col ← |vtxs(v)| − 1                    ▷ reverse first-fit coloring
10:     for each u ∈ W_local do
11:         while col ∈ F do
12:             col ← col − 1
13:         c[u] ← col
14:         col ← col − 1
```

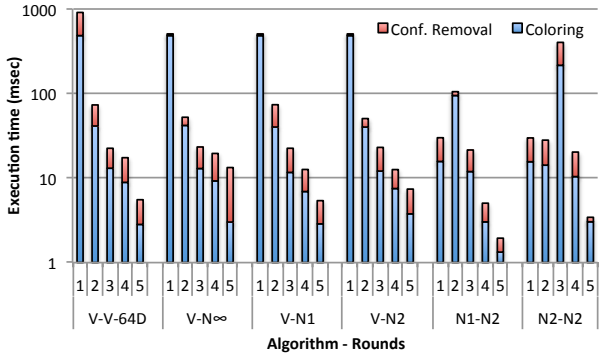


Figure 1: The execution times (in msec.) of each iteration for various algorithms while coloring coPapersDBLP with 16 threads.

is spent in the first iterations, 3) using net-based conflict removal at every iteration can make the performance worse (V-N∞), 4) using net-based coloring is a performance-wise good idea for the first iteration (N1-N2), 5) using an additional net-based coloring at the second iteration is not useful (N2-N2). The last observation can be obsolete if a better net-based (or a hybrid) coloring approach is found.

The impact on the number of colors: The optimistic net-based coloring does not significantly increase the number of colors in the final valid coloring. Although there can be relatively more conflicts during the intermediate steps, and these conflicts tend to increase the number of colors throughout the algorithm, the following lemma shows that net-based approach always uses at most the minimum number of colors for a valid coloring.

Lemma 1: The number of colors used by Algorithm 8 is less than or equal to the minimum number of colors for a valid coloring.

Proof: The value $L = \max_{v \in V_B}(|\texttt{vtxs}(v)|$ is a lower bound on the number of colors required for a valid coloring Each forbidden color set F in the algorithm (line 2) contains the colors of the vertices in $\texttt{vtxs}(v)$ for a $v \in V_B$ and chooses the next unused color id among $\{\texttt{vtxs}(v)-1, \ldots, 0\}$. Hence, each F contains at most L color ids smaller than L and the next color id is always smaller than or equal to L. ∎

Implementation details: For all the algorithms described above, the memories for the forbidden color set F and the local vertex queues W_{local} are allocated only once and simple arrays are used to realize them. Furthermore, these structures are never actually emptied or reset. For each thread, F is repetitively used for different nets/vertices via different markers without any reset operation. Similarly, the local queue W_{local} is emptied by only setting a local pointer to 0.

IV. Algorithms for Distance-2 Graph Coloring

The net-based approach can also be used for the D2GC problem. Due to the similarity between the problem definitions of BGPC and D2GC, the corresponding vertex- and net-based algorithms can be implemented along the lines of the bipartite graph partial coloring algorithms given above with a single difference: distance-1 neighbors must also be considered in the neighborhood as well. Here for completeness, we present the pseudo-codes for net-based D2GC coloring and conflict removal phases in Algorithms 9 and 10, respectively, but skip the vertex-based versions due to the space limitation. Since the input graph $G = (V, E)$ is unipartite, but not bipartite, instead of the $\texttt{nets}(u)$ and $\texttt{vtxs}(u)$, the notation $\texttt{nbor}(u)$ will be used to denote the adjacency list of a vertex $u \in V$. However, for consistency, we will keep naming these greedier versions as net-based.

Unlike the BGPC algorithms, for D2GC, the threads visit actual vertices to be colored; this is why, both of the D2GC coloring and conflict removal algorithms first process the color of the visited vertices (lines 4–7 of Algorithm 9 and lines 3–4 of Algorithm 10). This is necessary to handle the distance-1 neighbors which is the additional requirement for D2GC compared to BGPC. The same reverse first-fit policy is applied while coloring the vertices; the only difference is the candidate color is initialized with $|\texttt{nbor}(v)|$ instead of $|\texttt{nbor}(v)| - 1$ (as in D2GC) since the vertex assigned to a thread will also be colored by the thread requiring at least $|\texttt{nbor}(v)| + 1$ available colors (including color 0).

Algorithm 9 D2GC-ColorWorkQueue-Net

Input: $G = (V, E)$: a graph, $c[.]$: an incomplete coloring.
Output: $c[.]$: the (most) optimistic coloring array.

```
1:  for each v ∈ V in parallel do
2:      F ← ∅ : thread private forbidden color set for v
3:      W_local ← ∅ : thread private vertices to be colored
4:      if c[v] ≠ −1 then
5:          F ← F ∪ {c[v]}
6:      else
7:          W_local ← W_local ∪ {v}
8:      for each u ∈ nbor(v) do
9:          if c[u] ≠ −1 and c[u] ∉ F then
10:             F ← F ∪ {c[u]}
11:         else
12:             W_local ← W_local ∪ {u}
13:     col ← |nbor(v)|                    ▷ reverse first-fit coloring
14:     for each u ∈ W_local do
15:         while col ∈ F do
16:             col ← col − 1
17:         c[u] ← col
18:         col ← col − 1
```

Algorithm 10 D2GC-RemoveConflicts-Net

Input: $G = (V, E)$: a graph to color, $c[.]$: an optimistic coloring.
Output: $c[.]$: an incomplete coloring.

```
1:  for each v ∈ V in parallel do
2:      F ← ∅ : thread private forbidden color set for v
3:      if c[v] ≠ −1 then
4:          F ← F ∪ {c[v]}
5:      for each u ∈ nbor(v) do
6:          if c[u] ≠ −1 then
7:              if c[u] ∈ F then
8:                  c[u] ← −1
9:              else
10:                 F ← F ∪ {c[u]}
```

V. Balancing color set cardinalities

As mentioned before, graph coloring has been frequently used to parallelize a large task with many sub-tasks. In our preliminary experiments, the (reverse) first-fit policy generated a few large color sets (of small colors) and thousands of color sets with less than 2 elements for a real-life optimization problem. This result is in concordant with a comprehensive recent study focusing solely on balancing, parallel balancing heuristics, and their practical impacts on parallel computing [9]. In fact, on a single multicore CPU, the performance reduction (in FLOPS) may not hurt too much since most of the vertices, with small colors, can still be processed in parallel. However,

the impact of the imbalance increases with the number of processors/cores. Furthermore, in most of the iterative algorithms, processing only a few vertices and updating the current solution can be harmful from the optimization perspective since this restricts the dimensions of the moves in the search space performed to reach a better solution.

In this work, we experimented on cost-free and unsupervised balancing heuristics within the BGPC and D2GC algorithms proposed above. The straightforward choice would be keeping color set cardinalities dynamically throughout the execution; but this is expensive especially for large number of cores. Instead, the first proposed heuristic tries to keep the number of colors the same as much as possible and the second one aggressively applies balancing hence increases the number of colors (only around 10% on average). The heuristics are given in Algorithms 11 and 12 for the vertex-based approach. The net-based variants are also similar.

Algorithm 11 COLORWORKQUEUE-B1

Input: $G = (V, E)$, W: vertices to color, `nbor`(.): the neighborhood, c[.]: an incomplete coloring with no conflicts.
Output: c[.]: an optimistic coloring.

1: $col_{max} \leftarrow 0$: **thread private**
2: **for** each $w \in W$ **in parallel do**
3: ... ▷ lines 2-6 of Alg. 2
4: **if** $w \bmod 2 = 0$ **then**
5: $col \leftarrow col_{max}$
6: **while** $col \in F$ **do**
7: $col \leftarrow col - 1$
8: **if** $col = -1$ **then**
9: $col \leftarrow col_{max} + 1$
10: **while** $col \in F$ **do**
11: $col \leftarrow col + 1$
12: **else**
13: $col \leftarrow 0$
14: **while** $col \in F$ **do**
15: $col \leftarrow col + 1$
16: $c[w] \leftarrow col$
17: $col_{max} = max(col_{max}, col)$

In the first balancing heuristic B1, each thread keeps track of the maximum color it uses (col_{max} at line 1). The threads employ the first-fit policy for the odd-numbered vertices (or nets) and otherwise, they employ the reverse first-fit policy starting from col_{max}. Unlike the original BGPC and D2GC algorithms, starting from col_{max}, instead of $|\texttt{nbor}(w)| - 1$, necessitates a safety check (line 8). If this is the case, the heuristic initiates a first-fit starting from $col_{max} + 1$. By performing alternating policies w.r.t. the vertex (or net) id, B1 hopes to distribute the colors evenly in the interval $[0, col_{max}]$. If there is no color between this interval, it extends the size of the interval.

The second heuristic B2, given in Algorithm 12, keeps a variable col_{next} in addition to col_{max} to start from for the color search. The idea is the same: the heuristic wants to distribute the colors in between $[0, col_{max}]$ but increments the color to start by one for each vertex/net. To aggressively favor large color numbers and focus the later colors in the interval more, the minimum color to start is set to $col_{max}/3 + 1$ (the last line of Alg. 12). However, filling these color sets with more vertices increases the probability of them being in a forbidden-color array. Thus, more colors are expected to appear during the course of execution due to the conflicting nature of balancing and using less number of colors.

Algorithm 12 COLORWORKQUEUE-B2

Input: $G = (V, E)$, W: vertices to color, `nbor`(.): the neighborhood, c[.]: an incomplete coloring with no conflicts.
Output: c[.]: an optimistic coloring.

1: $col_{max} \leftarrow 0$: **thread private**
2: $col_{next} \leftarrow 0$: **thread private**
3: **for** each $w \in W$ **in parallel do**
4: ... ▷ lines 2-6 of Alg. 2
5: $col \leftarrow col_{next}$
6: **while** $col \in F$ **do**
7: $col \leftarrow col + 1$
8: **if** $col > col_{max}$ **then**
9: $col \leftarrow 0$
10: **while** $col \in F$ **do**
11: $col \leftarrow col + 1$
12: $c[w] \leftarrow col$
13: $col_{max} = max(col_{max}, col)$
14: $col_{next} = min(col + 1, col_{max}/3 + 1)$

VI. EXPERIMENTS

All the experiments in the paper are performed on a single machine running on 64 bit CentOS 6.5 equipped with 64GB RAM and a dual-socket Intel Xeon E7-4870 v2 clocked at 2.30 GHz where each socket has 15 cores (30 in total). For the multicore implementations, we used OpenMP and all the codes are compiled with `gcc 4.9.2` with the `-O3` optimization flag enabled. For each problem, we experimented on eight different algorithms which are combinations of the heuristics given in Sections III and IV. For fairness, all the algorithms are implemented within the `ColPack` environment using the same data structures as much as possible. All the algorithms are summarized below:

V-V: vertex-based coloring and conflict removal with first-fit policy. This is the default implementation of `ColPack` for BGPC. For D2GC, `ColPack` does not have a parallel implementation but a sequential one exists. We implemented the parallel version based on the BGPC algorithm by adding the corresponding statements for distance-1 neighbors.

V-V-64: Same as V-V but the chunk-size for dynamic scheduling of OpenMP threads are set to 64.

V-V-64D: In `ColPack`'s conflict removal, a conflicting vertex is immediately added to the shared work queue of the next iteration. Unlike V-V64, this algorithm performs a lazy construction by using private queues for each thread that are combined at the end of each iteration.

V-N∞, **V-N1** and **V-N2**: Vertex-based coloring (64D) with net-based conflict removal in all, the first, and the first two iterations, respectively. After that, the algorithms switch to vertex-based (64D) conflict removal.

N1-N2 and **N2-N2**: Similarly, these two algorithms use net-based coloring in the first and first-two iterations, respectively, and both use net-based conflict removal only in the first two iterations. The algorithms switch to the vertex-based (64D) variants after that.

The experiments are performed on eight graphs given in Table II which are generated from their corresponding

Matrix	Properties			Column deg.		Sequential BGPC V-V				Used
						Natural		Smallest Last		
	#rows	#cols	#nnz	max.	Std. dev.	Exec. time	#colors	Exec. time	#colors	
20M_movielens	26,744	138,493	20,000,263	67,310	3,085.81	587.15	70,815	1,236.33	68,077	✓×
af_shell [16]	1,508,065	1,508,065	27,090,195	35	1.00	3.39	50	4.13	45	✓✓
bone010 [16]	986,703	986,703	36,326,514	63	7.61	4.28	132	6.86	110	✓✓
channel [9]	4,802,000	4,802,000	42,681,372	18	1.00	2.57	39	4.75	36	✓✓
coPapersDBLP [9]	540,486	540,486	15,245,729	3,299	66.23	6.73	3,321	9.68	3,300	✓✓
HV15R [17]	2,017,169	2,017,169	283,073,458	484	53.95	66.94	508	87.01	484	✓×
nlpkkt120 [16]	3,542,400	3,542,400	50,194,096	28	3.00	4.22	59	7.88	49	✓✓
uk-2002 [9]	18,520,486	18,520,486	298,113,762	2,450	27.51	32.66	2,450	41.23	2,450	✓×

Table II: Graphs/matrices used in the experiments: columns 2-4 are the numbers of rows, columns, and nonzeros, respectively. The next two columns are the maximum number of nonzeros in a column and the standard deviation of the nonzero distribution. Columns 7-8 show the execution time of the sequential BGPC algorithm and the average number of colors when the natural row order is employed. The next columns do the same for the smallest-last order implemented in `ColPack` to reduce the number of distinct colors. (column 5, which is a lower bound on the number of colors used, can be compared with column 10 to evaluate the coloring quality of `ColPack`). The last column show if the matrix is used in BGPC and D2GC experiments, respectively. The ordering time is not included in the table. Moreover, since the executions are sequential, a conflict detection phase is not performed.

.

UFL matrices [18]. Seven out of the eight graphs have been taken from the coloring and related parallel computing literature [9], [16], [17]. We also included a matrix from MovieLens dataset [19], 20M_movielens, since matrix decomposition, and our preliminary experiments on these matrices, is the application that motivated us for this study. For BGPC, we colored the columns of these matrices where the rows are considered as the nets. For D2GC, we used 5 of 8 structurally symmetric matrices. This is denoted in the last column of the table.

A. Experiments for bipartite graph partial coloring

The execution times of BGPC algorithms for each matrix as well as the number of distinct colors are given in Figure 2 and the results are summarized in Table IV. When the natural vertex order is used, compared to sequential `ColPack` implementation of BGPC, i.e., V-V, one can obtain $6.01\times$ speed-up on 16 threads with 1% increase on the number of colors (V-N2). When the net-based coloring is employed for one iteration (N1-N2), the speedup increases to $11.38\times$ with a small, 8% increase on the number of colors. These algorithms are $2.17\times$ and $4.12\times$, respectively, faster than the parallel BGPC in `ColPack` on 16 threads.

We also compared the results when the smallest-last order in `ColPack` is employed. As Table II shows, this ordering indeed reduces the number of colors for most of the cases. The results of these experiments are summarized in Table IV. Since the sequential `ColPack` execution for this ordering is slower than that of the natural ordering, the speedups increase: compared to sequential V-V, the algorithms V-N2 and N1-N2 are $10.09\times$ and $16.76\times$ faster, respectively, with 16 threads. Compared to parallel V-V, on 16 threads, N1-N2 is $4.43\times$ faster with 9% increase on the number of colors used.

B. Experiments for distance-2 graph coloring

For D2GC, we have experimented on the five of eight matrices in our data set as explained above. We've selected four algorithms which obtained promising results in the BGPC experiments. The results are presented in Table V. Similar

Algorithm	Avg. #colors normalized w.r.t. V-V	Speedup over sequential V-V				Speedup over V-V for $t = 16$
		$t = 2$	$t = 4$	$t = 8$	$t = 16$	
V-V	1.00	0.74	1.24	1.88	2.76	1,00
V-V-64	1.01	0.81	1.40	2.36	4.00	1,45
V-V-64D	1.01	0.85	1.46	2.41	4.05	1,47
V-N∞	1.01	1.47	2.34	3.65	5.84	2,11
V-N1	1.01	1.48	2.35	3.64	5.85	2,11
V-N2	1.01	1.49	2.37	3.71	6.01	**2,17**
N1-N2	1.08	2.39	4.24	7.17	11.38	**4,12**
N2-N2	1.07	1.44	2.63	4.57	7.50	2,71

Table III: The average speedups over sequential and parallel V-V on 16 threads and the increase on the number of colors when the natural ordering of the columns is used. The numbers are the geometric means of the individual results for each matrix.

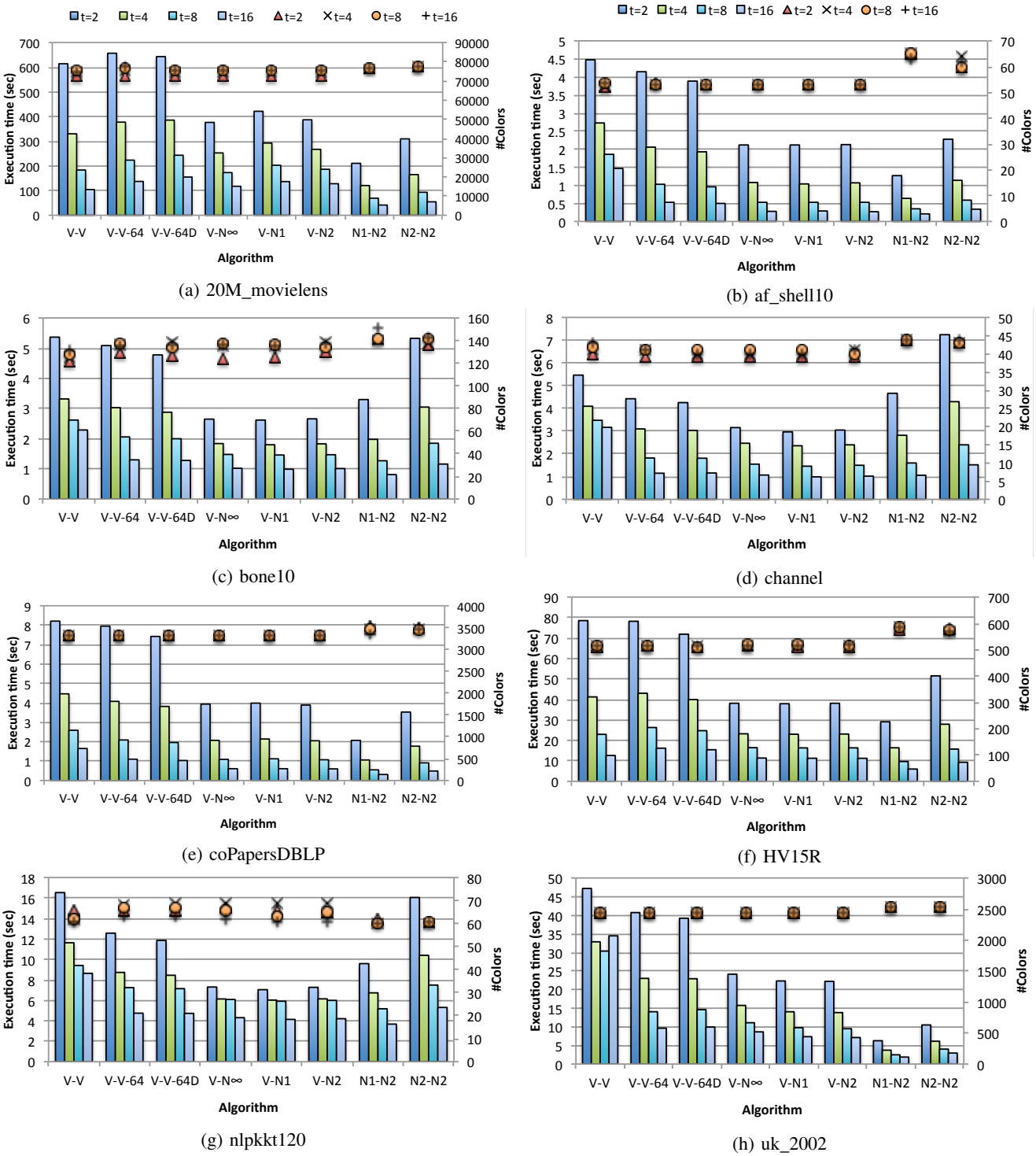
Algorithm	Avg. #colors normalized w.r.t. V-V	Speedup over sequential V-V				Speedup over V-V for $t = 16$
		$t = 2$	$t = 4$	$t = 8$	$t = 16$	
V-V	1.00	0.93	1.65	2.81	3.78	1,00
V-V-64	1.01	0.99	1.89	3.55	6.41	1.70
V-V-64D	0.99	1.04	1.99	3.75	6.86	1.81
V-N∞	1.00	1.62	3.01	5.41	9.20	2.43
V-N1	1.01	1.71	3.19	5.83	10.07	2.66
V-N2	0.99	1.72	3.21	5.87	10.09	**2.67**
N1-N2	1.09	3.47	6.26	10.82	16.76	**4.43**
N2-N2	1.10	2.24	4.04	6.94	11.19	2.96

Table IV: The average speedups over sequential and parallel V-V on 16 threads and the increase on the number of colors when smallest-last order of the columns is used. The numbers are the geometric means of the individual results for each matrix.

to BGPC, 16-thread V-N1 and N1-N2 is $8.97\times$ and $13.2\times$ faster than sequential V-V with only 4% and 9% increase in color counts, respectively. V-V-64D is used to normalize the 16-thread speedups since all the algorithms employ the 64D option. When the improvement of chunk size and lazy work-queue construction is removed, the optimism in N1-N2 obtains $2\times$ performance on 16-threads with only around 5% increase on the number of distinct colors.

C. Experiments on balancing

The impact on balancing heuristics B1 and B2 are presented in Table VI for BGPC experiments. The heuristics are applied to V-N2 and N1-N2 and the results are compared with their original implementation. Experimental results



(a) 20M_movielens

(b) af_shell10

(c) bone10

(d) channel

(e) coPapersDBLP

(f) HV15R

(g) nlpkkt120

(h) uk_2002

Figure 2: The execution times (left axis) on 2, 4, 8, 16 threads, respectively, and the number of colors (right axis) for all the matrices and algorithms.

Algorithm	Color w.r.t. V-V	Speedup over sequential V-V $t=2$	$t=4$	$t=8$	$t=16$	Speedup over V-V-64D for $t=16$
V-V-64D	1.04	1.38	2.18	3.46	6.11	1.00
V-N1	1.04	2.32	3.38	5.22	8.97	1.39
V-N2	1.04	2.27	3.37	5.24	8.87	1.37
N1-N2	1.09	2.49	4.44	7.85	13.20	2.00

Table V: The average speedups over sequential V-V and parallel V-V-64D on 16 threads and the increase on the number of colors (over V-V) when the natural ordering of the columns is used. The numbers are the geometric means of the individual results for each matrix. The results are the averages of 10 experiments for each matrix-algorithm-thread triplet.

Algorithm	Normalized w.r.t. X-N2 Coloring time	#Color sets	Average card.	Std. Dev.
V-N2-U	1.00	1.00	1.00	1.00
V-N2-B1	0.95	1.04	0.96	0.69
V-N2-B2	0.95	1.13	0.89	0.25
N1-N2-U	1.00	1.00	1.00	1.00
N1-N2-B1	0.99	1.04	0.96	0.84
N1-N2-B2	0.99	1.09	0.91	0.62

Table VI: Impact of balancing heuristics, B1 and B2, on the color set cardinalities and the number of color sets for parallel BGPC algorithms V-N2 and N1-N2 on 16 threads. Results are normalized with the original unbalanced algorithms denoted with -U.

show that, applying these heuristics is for free, i.e., there is no computational overhead as expected. For B1, the standard deviation of the color cardinalities decreases $0.69\times$ and $0.84\times$ when applied to V-N2 and N1-N2, respectively, on the expense of 4% color increase. For B2, which aggressively tries to reduce the number of colors, the standard deviation decreases $0.25\times$ and $0.62\times$ with around 9% and 13% increase on the number of colors for V-N2 and N1-N2, respectively. To better visualize the impact of these balancing heuristics, Figure 3 shows the distribution of color set cardinalities for the original and balanced executions of V-N2 and N1-N2 on coPapersDBLP.

VII. Related work

Coloring has mostly been investigated for distance-1 coloring, but most ideas can be ported to other variants. Since graph coloring is NP-Complete [1] and hard to approximate [2] in most of its variants, the vertices are greedily colored one after another, and the lowest available color for a vertex is selected. Such an algorithm produces a coloring with less than $1 + \Delta$ color for the distance-1 variant of the problem. Though to avoid the worst case, it is common to carefully choose the order in which the vertices are processed [7] using either a static [20], [21] or dynamic [22] ordering.

Earlier coloring algorithms [23], [24], [25] are based on generating maximum independent sets in parallel via algorithms such as [26]. Recent techniques optimistically color the vertices in parallel assuming that a valid coloring will be generated and then verify the validity of the coloring. In case of an invalid coloring, one of the neighbor vertices that are of the same color is tagged to be colored again in the next iteration of the algorithm. This technique was successfully applied on distributed memory machine [27], [28], [29], [30], including for BGPC and D2GC [5], [6]. The algorithm was investigated also on shared memory, multicore and manycore architectures [13], [31], [16], [8], [14] and on hybrid MPI + OpenMP systems [32]. One common point of [5], [6] and the proposed work is that the conflict removal phase of D2GC has been performed around middle vertices which is similar to the net-based conflict removal. Nevertheless, the authors studied D2GC in the distributed setting and applied the approach for all iterations.

The balanced graph coloring problem has been studied in the literature from different aspects; from theoretical perspective, the term "equitable" is used for the colorings where the color set cardinalities differ at most one [10], [11]. The most comprehensive study from the parallel computing perspective is recently introduced by Lu et al. [9]. In this work, we follow a similar approach but mostly aim to devise costless and online balancing heuristics that can be applied to the parallel greedy graph coloring algorithms.

VIII. Conclusion and future work

In this work, we proposed novel, greedier and more optimistic parallel algorithms for parallel BGPC and D2GC. We also proposed two costless balancing heuristics that can be applied to both BGPC and D2GC, as well as other coloring variants, to balance the color set cardinalities and improve the impact of the coloring on the real application to be parallelized. The results show that the proposed techniques are useful in practice and improves the performance and the *goodness* of the coloring.

The proposed techniques are suitable for GPUs and Intel Xeon Phi architectures which will be considered in future works. In fact, the task sizes in the vertex-based approach, i.e., the neighborhood sizes, deviates much more compared to that of the net-based approach, i.e., number of vertices adjacent to a net, which can be a comfort while parallelizing the coloring algorithms on manycore architectures. We also believe that a better net-based coloring and a better cost-free, self-balancing heuristic are worth investigating since experimental results indicate that their impact will be significant. Lastly, the optimistic techniques for BGPC and D2GC can be extended to the distance-k graph coloring problem and further performance improvements can be investigated.

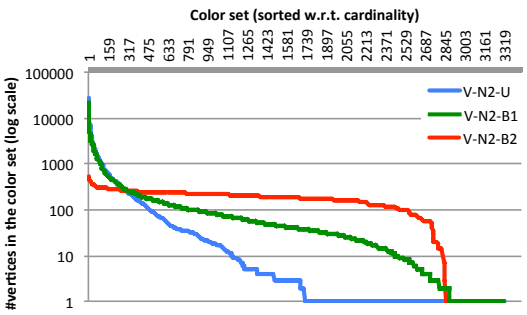
Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1652442.

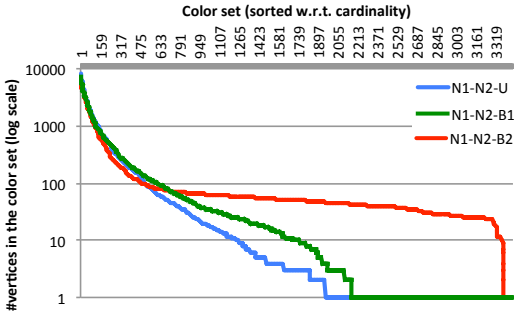
References

[1] D. W. Matula, "A min-max theorem for graphs with application to graph coloring," *SIAM Review*, vol. 10, pp. 481–482, 1968.

[2] D. Zuckerman, "Linear degree extractors and the inapproximability of max clique and chromatic number," *Theory of Computing*, vol. 3, pp. 103–128, 2007.



(a) Balancing coPapersDBLP for V-N2



(b) Balancing coPapersDBLP for N1-N2

Figure 3: Impact of balancing heuristics, B1 and B2, on the color set cardinalities and the number of color sets for BGPC algorithms parallel V-N2 (left) and N1-N2 (right) on 16-threads for coPapersDBLP.

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

[4] T. F. Coleman and J. J. More, "Estimation of sparse Jacobian matrices and graph coloring problems," *SIAM Journal on Numerical Analysis*, vol. 1, no. 20, pp. 187–209, 1983.

[5] D. Bozdağ, Ü. Çatalyürek, A. Gebremedhin, F. Manne, E. Boman, and F. Özgüner, "A parallel distance-2 graph coloring algorithm for distributed memory computers," in *Proc. of 1st Int'l. Conf. on High Performance Computing and Communications*. Springer, Sep 2005, pp. 796–806.

[6] ——, "Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation," *SIAM Journal of Scientific Computing*, vol. 32, no. 4, pp. 2418–2446, 2010.

[7] 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.

[8] ——, "Parallel distance-k coloring algorithms for numerical optimization," in *Euro-Par 2002 Parallel Processing - 8th International Conference*, 2002, pp. 912–921.

[9] H. Lu, M. Halappanavar, D. Chavarra-Miranda, A. Gebremedhin, and A. Kalyanaraman, "Balanced coloring for parallel computing applications," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE*, May 2015, pp. 7–16.

[10] W. Meyer, "Equitable coloring," *Amer. Math. Monthly*, vol. 80, pp. 920–922, 1973.

[11] A. Hajnal and E. Szemeredi, "Proof of a conjecture of p. erdos," *London: North-Holland*, pp. 601–623, 1970.

[12] J. Robert K. Gjertsen, M. T. Jones, and P. E. Plassmann, "Parallel heuristics for improved, balanced graph colorings," *Journal of Parallel and Distributed Computing*, vol. 37, no. 2, pp. 171–186, 1996.

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

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

[15] A. E. Sariyuce, E. Saule, and U. V. Catalyurek, "Scalable hybrid implementation of graph coloring using mpi and openmp," in *Proc. IPDPSW & PhD Forum*, 2012, pp. 1744–1753.

[16] M. Patwary, A. Gebremedhin, and A. Pothen, "New multithreaded ordering and coloring algorithms for multicore architectures," in *Euro-Par 2011 Parallel Processing - 17th International Conference*, 2011, pp. 250–262.

[17] T. Tessem, "Improving parallel sparse matrix-vector multiplication," Master's thesis, Univ. of Bergen, Fac. of Math. and Natural Sciences Dept. of Informatics, December 2013.

[18] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[19] "GroupLens Research: MovieLens Dataset," http://grouplens.org/datasets/movielens/, 2016, [Online; accessed Oct-1-2016].

[20] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *J. ACM*, vol. 30, pp. 417–427, July 1983.

[21] 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 Comp. Journal*, vol. 10, pp. 85–86, 1967.

[22] D. Brélaz, "New methods to color the vertices of a graph," *Commun. ACM*, vol. 22, pp. 251–256, April 1979.

[23] J. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. Martin, "A comparison of parallel graph coloring algorithms," Northeast Parallel Architectures Center at Syracuse University (NPAC), Tech. Rep. SCCS-666, 1994.

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

[25] R. K. Gjertsen Jr., M. T. Jones, and P. Plassmann, "Parallel heuristics for improved, balanced graph colorings," *Journal on Parallel and Dist. Computing*, vol. 37, pp. 171–186, 1996.

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

[27] E. Boman, D. Bozdağ, Ü. Çatalyürek, A. Gebremedhin, and F. Manne, "A scalable parallel graph coloring algorithm for distributed memory computers," in *Proc. of 11th Int'l. Euro-Par Conf. on Parallel Processing*, Aug 2005, pp. 241–251.

[28] D. Bozdağ, A. Gebremedhin, F. Manne, E. Boman, and Ü. Çatalyürek, "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.

[29] A. E. Sarıyüce, E. Saule, and U. V. Catalyurek, "Improving graph coloring on distributed memory parallel computers," in *18th Annual Int. Conf. on High Performance Comp.*, 2011.

[30] A. E. Sarıyüce, E. Saule, and Ü. V. Çatalyürek, "On distributed graph coloring with iterative recoloring," ArXiv, Tech. Rep. arXiv:1407.6745, Jul. 2014.

[31] A. H. Gebremedhin and F. Manne, "Parallel graph coloring algorithms using OpenMP (extended abstract)," in *In First European Workshop on OpenMP*, 1999, pp. 10–18.

[32] A. E. Sarıyüce, E. Saule, and U. V. Catalyurek, "Scalable hybrid implementation of graph coloring using MPI and OpenMP," in *IPDPSW, Workshop on Parallel Computing and Optimization (PCO)*, May 2012.