

# Parallel Graph Coloring for Manycore Architectures

Mehmet Deveci, Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam  
Sandia National Laboratories, Albuquerque, NM  
{mndevec, egboman, kddevin, srajama}@sandia.gov

**Abstract**—Graph algorithms are challenging to parallelize on manycore architectures due to complex data dependencies and irregular memory access. We consider the well studied problem of coloring the vertices of a graph. In many applications it is important to compute a coloring with few colors in near-linear time. In parallel, the optimistic (speculative) coloring method by Gebremedhin and Manne [1] is the preferred approach but it needs to be modified for manycore architectures. We discuss a range of implementation issues for this vertex-based optimistic approach. We also propose a novel edge-based optimistic approach that has more parallelism and is better suited to GPUs. We study the performance empirically on two architectures (Xeon Phi and GPU) and across many data sets (from finite element problems to social networks). Our implementation uses the Kokkos library, so it is portable across platforms. We show that on GPUs, we significantly reduce the number of colors (geometric mean 4X, but up to 48X) as compared to the widely used cuSPARSE library. In addition, our edge-based algorithm is 1.5 times faster on average than cuSPARSE, where it has speedups up to 139X on a circuit problem. We also show the effect of the coloring on a conjugate gradient solver using multicolored Symmetric Gauss-Seidel method as preconditioner; the higher coloring quality found by the proposed methods reduces the overall solve time up to 33% compared to cuSPARSE.

**Keywords**—graph coloring, combinatorial scientific computing, GPU, Xeon Phi, manycore

## I. INTRODUCTION

Graph problems arise in many applications, such as data analysis and computational science and engineering. Graph algorithms serve as useful tool by themselves for solving real problems of interest and as good benchmarks for kernel performance in emerging architectures. They are characterized by irregular data access, dynamic data structures, and non-traditional parallelization patterns with synchronization bottlenecks dictated by the structure of the graph or by the algorithm. In this paper, we consider an archetypal graph problem: *graph coloring*. Graph coloring is often used to find independent tasks that can be executed simultaneously. As minimizing the number of colors is NP-hard, we consider fast *almost-linear* time algorithms. Parallel graph coloring presents an interesting challenge for algorithm developers both in terms of the performance of the graph kernel itself and in terms of the impact of the coloring on real problems. While the former relies on new algorithms and implementations for improving the performance of the graph algorithm, the latter relies on the quality of the parallel graph algorithm, typically the number of colors for graph coloring. In this paper, we are concerned with both of these aspects of parallel graph coloring.

As architectures evolve from multicore to manycore, the

number of the threads or computing units that parallel algorithms are designed for has increased rapidly. Thus, *thread scalability* in parallel algorithms is crucial. The concept of thread scalability ideally applies to both the performance and quality aspects of the parallel algorithm, which means the trade-off of parallel performance and the quality of the result has to be taken into account in algorithm design. In the context of graph coloring, we contend that most of the past work, even work geared toward thread parallelism, is not focused on thread scalability for hundreds or thousands of threads. We show that it is possible to achieve both good performance and high quality with massive parallelism.

Different manycore architectures such as Xeon Phis and GPUs introduce more complexity in both algorithm design and implementation. New programming paradigms, such as Kokkos, OpenACC, and OpenCL, allow us to write implementations that are portable between different architectures. These paradigms can form the foundation for performance portability, enabling one algorithm/code to perform well on different architectures. We use Kokkos [2] to implement our algorithms so we can easily evaluate them on different architectures. We demonstrate that *performance portability* is an ideal goal, but in reality one needs different algorithms on different architectures, as the manycore architectures differ in their performance characteristics. Our results show which algorithmic features are important for different architectures and will allow us to choose features as architectures evolve.

Finally, while the number of colors is a good metric for the graph coloring algorithm itself, it may or may not translate into real parallel performance in an actual application using graph coloring. Therefore, we ask: “What is the real impact on parallel performance of a kernel due to different numbers of colors?” We study this with a parallel implementation of a Symmetric Gauss-Seidel (SGS) preconditioner that uses graph coloring as a way to improve parallelism. We show the impact of a good coloring on this SGS kernel with a preconditioned conjugate gradient solver using a finite-element mini application and matrices from the U. of Florida collection [3]. Our main contributions in this paper follow:

- 1) A new thread-scalable edge-based coloring algorithm;
- 2) A study of implementation issues for vertex- and edge-based methods;
- 3) Empirical comparisons on a range of graphs on both Xeon Phi and GPU;
- 4) Demonstrated impact on solvers using multicolored Gauss-Seidel preconditioning.

## II. BACKGROUND AND RELATED WORK

Let  $G = (V, E)$  be an undirected graph, where  $V$  and  $E$  correspond to the set of vertices and edges, respectively.  $|V|$  and  $|E|$  denote the number of vertices and edges in the graph. The set of the neighbors of a vertex  $v$  is referred as its adjacency,  $adj(v)$ . The cardinality of a vertex's adjacency is denoted  $\delta(v)$ , and the maximum degree of a graph is  $\Delta = \max_{v \in V} \delta(v)$ . Distance-1 graph coloring  $C : V \rightarrow N$  is a function that assigns a color to each vertex that satisfies  $C(v) \neq C(u)$  for all edges  $(u, v) \in E$ . Therefore, distance-1 coloring assigns colors to vertices such that each vertex has a different color from all of its neighbor vertices. The number of distinct colors assigned in the coloring problem is referred to as  $|C|$ . The graph coloring problem that minimizes  $|C|$  is NP-Hard and difficult to approximate [4]; therefore, various heuristics have been studied in the literature.

A vast amount of research has been done on graph coloring. The 2nd DIMACS Implementation Challenge (<http://dimacs.rutgers.edu/Challenges/>) included graph coloring. Although the graph coloring is NP-Hard, in practice, simple greedy heuristics [5] often obtain near optimal solutions. In this greedy heuristic, for each vertex  $v$ ,  $adj(v)$  is traversed and  $v$  is assigned an available color. In the *first-fit* version, the smallest available color is chosen. This can be implemented in linear  $O(n + m)$  time and it gives a  $\Delta + 1$  coloring, i.e., a coloring with  $|C| \leq \Delta + 1$ . A FORBIDDEN array is typically used to keep track of the neighbors' colors. The greedy method is sensitive to the vertex ordering, so special vertex orderings (e.g., Largest-First, Smallest-Last) can be used to reduce the number of colors [6].

The greedy method is inherently sequential and, thus, difficult to parallelize. Jones and Plassmann [7] proposed a parallel algorithm (JP) based on independent sets. Each vertex in an independent set can be colored concurrently (but might be assigned different colors). Finding independent sets can itself be parallelized using Luby's method [8]. The JP algorithm is often slow as it does more work than the serial greedy algorithm and may need a large number of synchronization points. One advantage of the JP method is one can apply vertex ordering to reduce the number of colors. Hasenplaugh et al. [9] implemented the JP algorithm for multicore CPU and proposed a new vertex ordering (largest log-degree first).

Gebremedhin and Manne [1] proposed a multithreaded parallel algorithm based on the alternative approach of *optimistic* (speculative) coloring. Each thread performs greedy coloring asynchronously, but there may be conflicts due to two threads coloring two neighboring vertices at the same time. Therefore, conflict detection and conflict resolution phases are needed. The conflict resolution can be done in serial [1] or iteratively in parallel [10]. We focus on the latter approach as it is known to be more scalable. The Gebremedhin-Manne (GM) algorithm has been extended to distributed-memory [10]. The iterative approach first proposed there was later adopted on shared-memory systems [11], [12]. A hybrid MPI and OpenMP implementation was described by Sariyüce et al. [13]. A

variation to reduce the number of kernel launches was recently proposed by Rokos et al. [14].

Most previous work has focused on multicore machines with small numbers of cores. Saule and Çatalyürek first evaluated optimistic coloring on the Intel MIC (Xeon Phi) [12]. Çatalyürek et al. [11] proposed a highly multithreaded data-flow algorithm (related to the JP algorithm) for Cray XMT, but because it relies on hardware support for full-empty bits, it is not suitable for current GPU. Yet little work has been done on coloring on GPU. Grosset et al. [15] adapted the GM algorithm to GPU, but parts of the conflict resolution was done on CPU. Surprisingly, they found that the number of colors was actually reduced on GPU. Naumov et al. [16] developed a fast coloring heuristic that is part of the widely used cuSPARSE library. This algorithm can be considered a variation of Luby's algorithm where all vertices in an independent set are assigned the *same* color and this color is never used again. It is simpler and often faster than the JP algorithm but produces more colors. Che et al. [17] studied graph coloring on GPU with variations of the JP algorithm. They observed load imbalance due to variation in vertex degrees and proposed two strategies to address this issue.

### A. Optimistic Parallel Graph Coloring

---

#### Algorithm 1 IPGC: Iterative Parallel Graph Coloring

---

**Require:**  $G = (V, E)$   
1:  $C(v) \leftarrow 0$ , for all  $v \in V$   
2:  $CONF \leftarrow V$   
3: **while**  $CONF \neq \emptyset$  **do**  
4:    $ASSIGNCOLORS(G, C, CONF)$   
5:    $CONF \leftarrow DETECTCONFLICTS(G, C, CONF)$   
6: **return**  $C$

---



---

#### Algorithm 2 IPGCAC: IPGC - ASSIGNCOLORS

---

**Require:**  $G = (V, E), C, CONF$   
1: Allocate private FORBIDDEN with size max degree.  
2: **for**  $v \in CONF$  in parallel **do**  
3:    $FORBIDDEN \leftarrow false$   
4:    $FORBIDDEN(C(u)) \leftarrow true$  for  $u \in adj(v)$   
5:    $C(v) \leftarrow \min \{i > 0 | FORBIDDEN(i) = false\}$   
6: **return**  $C$

---



---

#### Algorithm 3 IPGDC: IPGC- DETECTCONFLICTS

---

**Require:**  $G = (V, E), C, CONF$   
1:  $NEWCONF \leftarrow \emptyset$   
2: **for**  $v \in CONF$  in parallel **do**  
3:   **for**  $u \in adj(v)$  **do**  
4:     **if**  $C(u) = C(v)$  and  $u < v$  **then**  
5:        $ATOMIC\ NEWCONF \leftarrow NEWCONF \cup v$   
6: **return**  $NEWCONF$

---

Algorithms 1, 2, and 3 describe iterative optimistic (speculative) parallel graph coloring for shared-memory [11], [12], which is based on [1]. The vertices are colored speculatively within  $ASSIGNCOLORS$ . Then  $DETECTCONFLICTS$  detects

conflicts that occurs due to race conditions. The conflicted vertices are recolored in the next iteration, and the algorithm terminates when there are no conflicts. We refer to this algorithm as IPGC; our implementation uses improvements from [11] where initialization of FORBIDDEN is moved outside of the loop. IPGC can be classified as a *vertex-based* parallel graph coloring algorithm, since the coloring process of a single vertex is always handled by a single thread. To our knowledge, all existing parallel graph coloring algorithms are vertex-based algorithms. Note that each thread has its own private FORBIDDEN in algorithm 2. FORBIDDEN must be large enough for all possible colors, so it typically has size  $\Delta$ . However, allocating such a private array for each thread is problematic in massively threaded architectures.

### B. Kokkos Library

Kokkos [2] is a C++ library that provides an abstract thread parallel programming environment and enables performance portability for common multi- and many-core architectures. It provides a single programming interface but enables different optimizations for backends such as OpenMP and CUDA. We used only a small subset of Kokkos’ features, mainly `parallel_for`, `parallel_scan`, `atomics`, and `views` (arrays). Using Kokkos allows us to run the same code on Xeon Phi and GPU just with different compile options.

## III. ALGORITHMS

### A. Vertex-Based Graph Coloring

Although the vertex-based iterative coloring algorithm is quite simple, several implementation details are often ignored. We present two key optimizations to these algorithms. First, we improve how FORBIDDEN arrays are handled. This optimization is aimed at enabling the VB algorithms run on an architectures with lots of threads. Second, we eliminate the FORBIDDEN array altogether and change how it is represented in memory in order to be scalable in GPUs.

1) *Fixed size FORBIDDEN array*: The simplest way to eliminate the need for private FORBIDDEN arrays in *each thread* is to just try each possible color in increasing order until a valid color is found. This would require  $O(\delta(v)^2)$  work for each vertex  $v$ , so the total complexity is  $O(\Delta|E|)$  for the most expensive round. Instead, a better trade-off between memory and run-time is to keep a small FORBIDDEN array of fixed size (MAXFORBID) and to traverse the adjacency list of some vertices more than once. For graphs requiring only a small number of colors, it is sufficient to traverse the adjacency list of the vertices once, but those needing more colors will require multiple passes of the adjacency list for some vertices. The resulting ASSIGNCOLORS function is listed in Algorithm 4 (called VB hereafter).

Algorithm 4 allocates arrays with size MAXFORBID in *each thread*. We introduce the idea of COLORRANGE for handling more colors than MAXFORBID. Only the colors within a COLORRANGE are considered in one pass. If a color cannot be found in this range, the adjacency list is iterated over again with the next COLORRANGE. FORBIDCOLOR forbids a color

---

### Algorithm 4 VB: IPGC- ASSIGNCOLORS with Fixed Forbidden Array

---

**Require:**  $G = (V, E), C, \text{CONF}$   
1: Allocate private FORBIDDEN with size MAXFORBID  
2: **for**  $v \in \text{CONF}$  in parallel **do**  
3:   OFFSET  $\leftarrow 0$   
4:   **while**  $v$  is not colored **do**  
5:     FORBIDDEN  $\leftarrow \text{false}$   
6:     COLORRANGE  $\leftarrow [\text{OFFSET}, \text{OFFSET} + \text{MAXFORBID})$   
7:     **for**  $u \in \text{adj}(v)$  and  $C(u) \in \text{COLORRANGE}$  **do**  
8:       FORBIDDEN  $\leftarrow \text{FORBIDCOLOR}(\text{FORBIDDEN}, C(u), \text{OFFSET})$   
9:      $C(v) \leftarrow \text{SETCOLOR}(\text{FORBIDDEN}, \text{OFFSET})$   
10:    OFFSET  $\leftarrow \text{OFFSET} + \text{MAXFORBID}$   
11: **return**  $C$

---

(by setting  $\text{FORBIDDEN}(C(u) - \text{OFFSET})$  to true), only if the neighbor’s color is within the current color range. At the end of a pass, SETCOLOR picks the smallest available color in FORBIDDEN and shifts it by the OFFSET. In the rest of the paper, we use  $\text{MAXFORBID} = 64$ , as it balanced the memory requirements and work for both GPUs and Xeon Phi’s.

2) *Bitwise representation of FORBIDDEN*: Even a thread-private array of size 64 is sometimes too big to be handled in registers and gets allocated in the local memory on GPUs. Frequent accesses to this array may cause VB to suffer from memory stalls. To prevent this, we introduce a bitwise representation of FORBIDDEN using a single (long) integer instead of an array. A 0 (1) bit in FORBIDDEN corresponds to an available (forbidden) color. An integer FORBIDDEN can be stored on thread registers on GPUs, minimizing the memory latency penalty. This requires slight changes to FORBIDCOLOR and SETCOLOR in Algorithm 4. When a neighbor with a color in current range is encountered, FORBIDCOLOR sets the bit with index  $C(v) - \text{OFFSET}$  to one. Once the traversal is completed, SETCOLOR picks a color by finding the first available bit in FORBIDDEN. SETCOLOR can be done in  $O(1)$  using BITWISEAND and two’s complements of FORBIDDEN. We will refer to this algorithm, using a *long integer* to represent FORBIDDEN, as VB BIT.

### B. Edge-Based Graph Coloring

Vertex-based algorithms have several disadvantages on SIMD architectures (GPUs). For example, the thread that handles the vertex with the highest degree might become the bottleneck, and the other threads within the same warp stall and wait for the completion of this thread. This is especially problematic for graphs with large variation in vertex degrees. In addition, neighbor list traversals are not coalesced. Such issues can be avoided with edge-based algorithms where threads own edges, rather than vertices.

Algorithm 5 shows a simple edge-based coloring approach. In this approach, we assume that  $\text{VFORBIDDEN}(v)$  holds the list of the forbidden colors for vertex  $v$ , initialized to  $\emptyset$ . ASSIGNCOLORS goes over all the vertices and picks the smallest available color for  $v$  based on  $\text{VFORBIDDEN}(v)$ . DETECTCONFLICTS loops over all edges and checks if the

---

**Algorithm 5** EB: Edge-Based Graph Coloring

---

**Require:**  $G = (V, E)$

- 1: SET  $VFORBIDDEN(v) = \emptyset$  for all  $v \in V$
  - 2: **while** there exists a  $v \in V$  that is not colored **do**
  - 3:  $C \leftarrow ASSIGNCOLORS(V, VFORBIDDEN)$
  - 4:  $C \leftarrow DETECTCONFLICTS(E, C)$
  - 5:  $VFORBIDDEN \leftarrow FORBIDCOLORS(E, C, VFORBIDDEN)$
  - 6: **return**  $C$
- 

two endpoints of an edge have the same color. If so, it marks one endpoint with a conflict. Then, FORBIDCOLORS goes over all edges, and *atomically* updates VFORBIDDEN of vertices based on the colors of their neighbors.

Figure 1 gives an example workflow of the simple edge-based graph coloring algorithm for a simple graph. It shows three rounds of edge-based coloring with two threads. The rounds are shown as columns and the functions within a round are shown as rows. The vertices and edges processed by different threads are differentiated by orange and purple lines. Initially, all vertices have no colors, and their VFORBIDDEN is  $\emptyset$ . In the first round, ASSIGNCOLORS colors all vertices red (first fit). Next, DETECTCONFLICTS goes over all the edges and resets the color of the vertex with the smaller index in case of a conflict. For example, the color of  $v_1$  is reset by the first thread because of the conflict  $(v_1, v_2)$ . Then FORBIDCOLORS loops over the edges, and each thread writes the color of the vertex at one end to VFORBIDDEN of the other end. This write operation might require atomic operations based on the data structure used for VFORBIDDEN. For example, in the first round, the first thread writes red to VFORBIDDEN of  $v_1$  because of the edge  $(v_1, v_2)$ . FORBIDCOLORS does nothing for the edges where both ends have no color such as  $(v_1, v_3)$ . Further rounds are also shown in the figure for clarity.

**Lemma 1.** *Algorithm 5 terminates with a correct coloring.*

*Proof:* We show correctness in two steps. First, the coloring must be correct if the algorithm terminates, since DETECTCONFLICTS resets the color of any conflicted vertex. Second, the number of colored vertices strictly increases in each round. As creation of the forbidden colors and assignment of actual colors happen independently, color assignments in prior rounds cannot cause conflicts in later rounds. Conflicts may arise only between vertices that are colored in the same round. However, even if all the vertices that are colored in the same round become conflicted, DETECTCONFLICTS leaves at least one vertex with the correct color, as a result of the index-based tie-breaking when conflicts arise irrespective of race conditions. Therefore, the algorithm will terminate with a correct coloring. ■

**Lemma 2.** *Algorithm 5 will result in  $\Delta + 1$  coloring.*

*Proof:* We claim that for any vertex  $v$ , the cardinality of  $VFORBIDDEN(v)$  can be at most  $\delta(v)$ . As long as each edge incident on  $v$  contributes only one single color to  $VFORBIDDEN(v)$ , ASSIGNCOLORS for a vertex will pick a color that is at most  $\delta(v) + 1$ . In this algorithm, no more

than one color can be forbidden because of the same edge as FORBIDCOLORS sets  $VFORBIDDEN(v)$  of one of the incident vertices with the color of the other. Furthermore, colors do not change in FORBIDCOLORS, eliminating any possibility of multiple colors for a vertex due to race conditions. Thus, each vertex  $v$  needs at most  $\delta(v) + 1$  colors, giving a  $\Delta + 1$  coloring. ■

While, we use Algorithm 5 and Figure 1 to show the simple edge-based graph coloring algorithm, this method does poorly in practice. We do not show this method in the rest of the paper, but instead modify it with three sophisticated optimizations. The three issues with the simple method are given below.

**Memory:** Algorithm 5 requires one FORBIDDEN array per vertex. To reduce memory usage, we propose the use of the color sets ( $CS$ ) as a counter to indicate the COLORRANGE (Section III-B1).

**Convergence issues:** In Figure 1, even when there are no race conditions, the algorithm ends up with conflicts because VFORBIDDEN is created with partial information based on edges with at least one colored vertex. This results in conflicts when an edge has both vertices uncolored, which causes the algorithm to need too many coloring rounds to converge. We introduce an optimization called tentative coloring to avoid such situations (Section III-B2).

**High number of edge traversals:** There are far more edges than vertices in a typical graph. This makes the traversals in an edge-base algorithm much more expensive than those in a vertex-based one. As a result, maintaining an edge work list and filtering them become crucial for obtaining performance (Section III-B3).

1) *Use of Color Set:* We reduce the memory requirement to  $O(|V|)$  by using COLORRANGE and bit manipulations similar to VB BIT. COLORRANGE is thread-private in the VB algorithm; therefore, it is dynamically created and incremented during the neighbor traversals. This is not possible for the edge-based algorithm, as gathering forbidden colors and assigning colors are done in separate phases. We introduce a static counter per vertex, known as a color set  $CS(v)$ , to indicate the COLORRANGE of the vertex  $v$ . In each round, we consider only those neighbors that are in the same COLORRANGE to set  $VFORBIDDEN(v)$ . If no available colors are found at the end of the round,  $CS(v)$  is incremented, and *colors from the next color range are sought in the next round*. Such static  $CS(v)$  storage also allows us to avoid conversions to bits from colors. Throughout the algorithm, a 32 bit integer  $C(v)$  is used to represent at most 32 colors. More colors are represented using  $CS(v)$ . At the end of the coloring, they are converted back to the actual color using  $CS(v) \times 32 + \log_2(C(v))$ .

Algorithms 6 and 7 show the modified FORBIDCOLORS and ASSIGNCOLORS. In FORBIDCOLORS, edges are processed only if the connected vertices have the same  $CS$ , one vertex is colored, and the other is not. Then, the color of the one end is added to the VFORBIDDEN of the other vertex using a BITWISEOR operation (denoted with ‘|’ in the algorithms).

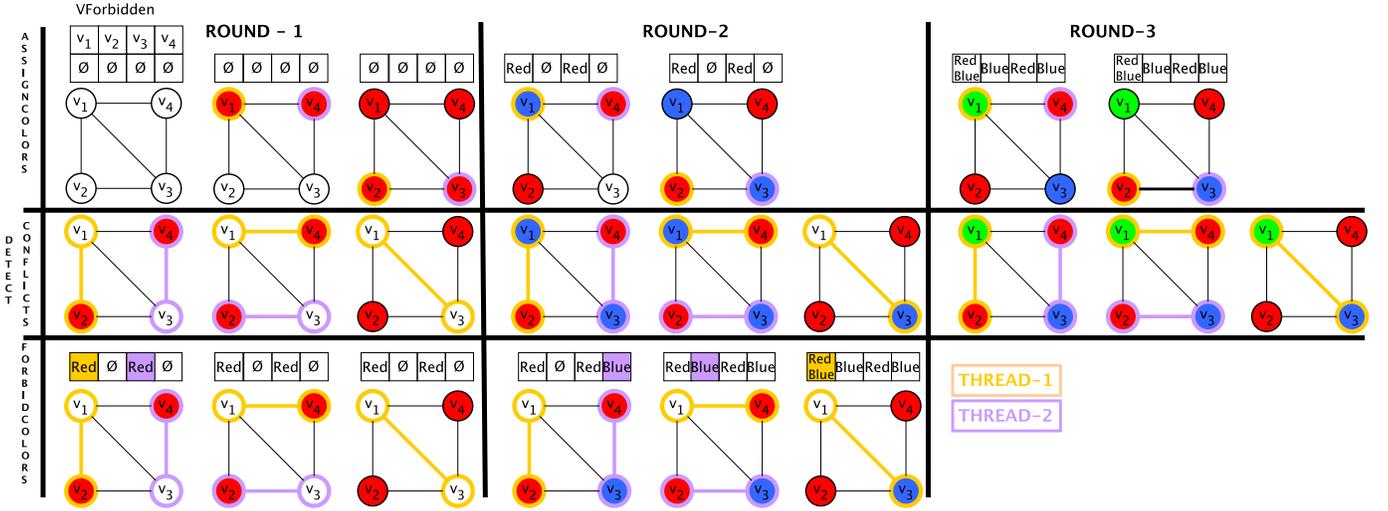


Fig. 1: Workflow of a Simple Edge-Based Graph Coloring Algorithm. The functions are given as rows, while the iterations are given as the columns of the figure. The vertices and edges that are lined with orange and purple are those processed by thread-1 and thread-2, respectively. Note that ASSIGNCOLORS loops over the vertices, while DETECTCONFLICTS and FORBIDCOLORS loop over the edges.

---

#### Algorithm 6 EB: FORBIDCOLORS

---

**Require:**  $E, CS, C, VFORBIDDEN$

- 1: **for**  $(u, v) \in E$  and  $CS(u) = CS(v)$  in parallel **do**
  - 2:   **if**  $u$  is colored,  $v$  is not colored **then**
  - 3:     ATOMIC  $VFORBIDDEN(v) \leftarrow VFORBIDDEN(v) \mid C(u)$
  - 4:   **if**  $v$  is colored,  $u$  is not colored **then**
  - 5:     ATOMIC  $VFORBIDDEN(u) \leftarrow VFORBIDDEN(u) \mid C(v)$
- 

#### Algorithm 7 EB: ASSIGNCOLORS

---

**Require:**  $V, CS, C, VFORBIDDEN$

- 1: **for**  $v \in V$  and  $C(v) = 0$  in parallel **do**
  - 2:   **if**  $VFORBIDDEN(v)$  has bits set to zero **then**
  - 3:      $C(v) \leftarrow \text{FIRSTAVAILABLEBIT}(VFORBIDDEN(v))$
  - 4:   **else**
  - 5:      $CS(v) \leftarrow CS(v) + 1$
  - 6:      $VFORBIDDEN(v) \leftarrow 0$
- 

In ASSIGNCOLORS, the first available bit of VFORBIDDEN is found in  $O(1)$  as in VB BIT.

2) *Improving Convergence:* We propose tentative coloring to reduce the number of rounds of the edge-based algorithm. In this step, we process edges with two uncolored vertices, and force them to let each other know about the other's tentative color. We give the details of TENTATIVECOLOR in Algorithm 8, which is called right after FORBIDCOLORS in Algorithm 5 in the modified algorithm.

TENTATIVECOLOR iterates over the edges, and processes only those having both vertices with tentative colors or no colors. Here, tentative colors refer to the colors given in this step. Tentative colors are marked by using the negative of the intended color as a flag. (With this change, a 32 bit integer can now represent 31 different colors, and 1st bit is used as tentative color flag.) In addition, we introduce TVFORBIDDEN here, which has the same structure as VFORBIDDEN, but holds the forbidden tentative colors, to avoid mixing tentative and

---

#### Algorithm 8 EB: TENTATIVECOLOR

---

**Require:**  $E, CS, C, VFORBIDDEN, TVFORBIDDEN$

- 1: **for**  $(u, v) \in E$  and  $CS(u) = CS(v)$  in parallel **do**
  - 2:   **if** neither  $u$  nor  $v$  are colored **then**
  - 3:      $ALLFORBID \leftarrow VFORBIDDEN(v) \mid TVFORBIDDEN(v)$
  - 4:      $C(v) \leftarrow -\text{FIRSTAVAILABLEBIT}(ALLFORBID)$
  - 5:     ATOMIC  $TVFORBIDDEN(u) \leftarrow TVFORBIDDEN(u) \mid -C(v)$
  - 6:   **if**  $u$  and  $v$  are tentatively colored and  $C(u) = C(v)$  **then**
  - 7:      $ALLFORBID \leftarrow VFORBIDDEN(v) \mid TVFORBIDDEN(v)$
  - 8:      $C(v) \leftarrow -\text{FIRSTAVAILABLEBIT}(ALLFORBID)$
  - 9:     ATOMIC  $TVFORBIDDEN(u) \leftarrow TVFORBIDDEN(u) \mid -C(v)$
  - 10:   **if**  $u$  is tentatively colored,  $v$  is not colored **then**
  - 11:     ATOMIC  $TVFORBIDDEN(u) \leftarrow TVFORBIDDEN(u) \mid -C(v)$
  - 12:   **if**  $u$  is tentatively colored,  $v$  is not colored **then**
  - 13:     ATOMIC  $TVFORBIDDEN(u) \leftarrow TVFORBIDDEN(u) \mid -C(v)$
- 

assigned colors. When processing an edge  $(u, v)$ , there are various scenarios in TENTATIVECOLOR:

1. If neither  $u$  nor  $v$  is colored, we pick a tentative color for  $v$  having larger index (index comparison is omitted in Algorithm 8 for simplicity) using  $VFORBIDDEN(v)$  and  $TVFORBIDDEN(v)$ , and add the tentative color of  $v$  to  $TVFORBIDDEN(u)$ .
2. If  $u$  is tentatively colored, and  $v$  is not colored yet, we add the tentative color of  $u$  to  $TVFORBIDDEN(v)$  and vice versa.
3. If both  $u$  and  $v$  are tentatively colored, we check for conflicts. In case of conflict, the vertex with smaller index is recolored and this new tentative color is added to  $TVFORBIDDEN$  of the other vertex.

With tentative coloring, we also change the ASSIGNCOLOR to use TVFORBIDDEN as well as VFORBIDDEN as shown in Algorithm 9. In the updated ASSIGNCOLOR, threads traverse vertices. If the vertex is tentatively colored, it removes the tentative flag by negating the color. Otherwise, it picks the first available color using VFORBIDDEN and TVFORBIDDEN. If all of the bits in ALLFORBID are 1, all the neighbors have

---

**Algorithm 9** EB: ASSIGNCOLOR using Tentative Colors

---

**Require:**  $V, CS, C, VFORBIDDEN, TVFORBIDDEN$

- 1: **for**  $v \in V$  in parallel **do**
- 2:   **if**  $C(v)$  is tentatively colored **then**
- 3:      $C(v) \leftarrow -C(v)$
- 4:   **if**  $v$  is not colored **then**
- 5:      $ALLFORBID \leftarrow VFORBIDDEN(v) \mid TVFORBIDDEN(v)$
- 6:     **if**  $ALLFORBID$  has bits set to zero **then**
- 7:        $C(v) \leftarrow FIRSTAVAILABLEBIT(ALLFORBID)$
- 8:     **else**
- 9:       **if**  $VFORBIDDEN(v)$  has bits set to zero **then**
- 10:          $CS(v) \leftarrow CS(v) + 1$
- 11:          $VFORBIDDEN(v) \leftarrow 0$

---

used all the colors in the  $CS(v)$ . In such cases,  $CS(v)$  is incremented and an available color is searched within new  $CS(v)$  during the next round. It is very important that we do not increase the number of colors and stay within  $\delta + 1$  upper bound with the addition of tentative coloring. This upper bound is satisfied as long as only one color for each neighbor is added to  $VFORBIDDEN$  or  $TVFORBIDDEN$ .  $TVFORBIDDEN(v)$  might include the tentative color of a vertex  $u$ , which might get recolored in another  $CS(v)$  because of a conflict. In this case, more than one color might be forbidden because of  $u$ , potentially causing a violation of the  $\delta + 1$  upper bound, as  $v$  might not know about the available color in a lower  $CS(v)$ . We solve this problem by incrementing  $CS$  based only on the finalized colors of  $VFORBIDDEN$  and not on  $TVFORBIDDEN$  (Algorithm 9). This guarantees that only one color is taken into account for each neighbor, thus, the algorithm still satisfies  $\delta + 1$  upper bound.

3) *Reducing the edge-traversals in each iteration:* All functions used in EB except ASSIGNCOLOR perform traversals on  $E$ . It is possible to reduce the size of the  $E$  throughout the execution. We introduce a new  $CREATENEWEDGELIST$  function after  $DETECTCONFLICT$  in Algorithm 5 (not shown because of space considerations), to reduce the size of  $EdgeList$  as the algorithm progresses.  $EdgeList$  is initially set to  $E$ , and in each round  $CREATENEWEDGELIST$  removes three types of edges from this set:

1. Edges with two colored vertices: The vertices that remain colored after  $DETECTCONFLICT$  will not have further conflicts thanks to  $FORBIDCOLORS$ , and their colors will be final. Thus, such edges can be removed from  $EdgeList$ .
2. An edge that has been seen by  $FORBIDCOLORS$  with one end point colored: Similar to the previous case, a vertex that survived with a color after  $DETECTCONFLICT$  is guaranteed not to have conflicts. Therefore, such edges will not provide any new information and can be removed from  $EdgeList$ .
3. Edges in which  $CS$  of its colored vertex is lower than the color set of the uncolored vertex. These are edges with one colored and one uncolored end; however, such edges might never be processed by  $FORBIDCOLORS$ , since color set of the uncolored vertex is higher than the colored one. In such cases, these edges will not give any new forbidden color information; therefore, they can be removed from  $EdgeList$ .

This proposed edge-based algorithm will be referred as EB for the rest of the paper. EB follows Algorithm 5 with the three optimizations above and uses 32 bit integers to store colors and forbidden colors.

#### IV. IMPLEMENTATION ISSUES

We describe choices of implementation and data structures to consider when designing multithreaded graph algorithms.

##### A. Maintaining the Worklist

Both vertex-based and edge-based algorithms maintain a dynamic conflict list (worklist), an array, in order to reduce the amount of work done in later rounds. Maintaining such dynamic arrays is challenging on highly multithreaded platforms. One approach is to use *atomic* operations to maintain the length of the list. Here, a global counter is incremented with *atomic\_fetch\_and\_add* by each thread in order to obtain the position in the array to which it writes a vertex (or edge) index. We refer to algorithms using this method as **ATOMIC**. Another approach is to use parallel prefix sum (PPS). Here, conflicted vertices (or edges) are marked with binary numbers (1 for conflicts, 0 otherwise), and a parallel prefix sum is used to get the position to write in the new worklist. **ATOMIC** can suffer from the critical region access when the accesses are frequent, while PPS requires two sweeps of the array. In addition, **ATOMIC** might result in a scattered worklist where consecutive vertices end up far apart, while PPS preserves the order of the vertices in the worklist, which might result in better cache usage in MIC, and better coalesced accesses on GPU. We implemented both versions for both vertex- and edge-based algorithms. Experiments showed the impact is negligible on vertex-based algorithms and significant on edge-based algorithms. We compare both options for the edge-based algorithm in Section V.

##### B. Edge Filtering

In the vertex-based algorithms, due to the limited size of the forbidden array, the neighbors of a vertex  $v$  might be traversed multiple times. In each traversal, only the colors in a color range are considered. Therefore, a neighbor vertex that has been considered in a previous traversal is not needed in further traversals with different color ranges. Thus, one can do an *edge filtering* for the proposed vertex-based algorithms. During each traversal, we place the considered neighbors to the front (or end) of the adjacency list, and skip such vertices in later traversals. These swaps require extra memory operations; however, they are helpful for graphs with high numbers of colors. In the experiments, we report the results with this edge-filtering (EF) optimization in vertex-based algorithms.

##### C. Chunk Size for Vertices

It is often difficult to leverage SIMD instructions (e.g. GPUs) for graph coloring purposes [14]. Because of the SIMD nature of GPUs, the algorithms might incur many conflicts due to coloring decisions for neighboring vertices made within the same clock cycle. To prevent such cases, we introduce the

notion of chunk sizes, where each thread is assigned a chunk of consecutive vertices. This has no effect on multi-core and some many-core architectures where threads are already assigned consecutive chunks. On the other hand, this update harms the coalesced memory accesses in GPUs, but reduces the number of conflicts, and therefore the number of the rounds in the algorithm. In initial experiments, we have tested our dataset with different chunk sizes on GPUs. Over all the datasets, using chunk size of 8 reduces the number of rounds by 36% and the execution time by 6%. As expected, it increases the cost of a single iteration. The reduction in time is greater on graphs where the vertices have uniform degree, but run time increases on more irregular graphs. We used a chunk size of 8 for vertex-based algorithms on GPUs in Section V.

#### D. Improved Conflict Resolution

When a color conflict is discovered, only one of the vertices corresponding to that edge needs to be recolored. We used the simple rule to recolor the lower numbered vertex. One could also use a random number for each vertex but we observed this made little difference in practice. A more interesting strategy is to recolor the vertex of lower degree, in order to reduce the risk of new conflicts due to recoloring. Also, in serial, the Largest-First and Smallest-Last ordering strategies have proven effective for reducing the number of colors. However, querying the degree information requires extra memory costs. Moreover, obtaining this information can be expensive in edge-based algorithms. In our experiments, we tested this scheme on our vertex-based algorithms. For most of the graphs, the gain did not amortize the cost of extra memory access. However, it had significant improvements on the highly irregular graphs. For example, in the `kron_g500logn21` graph from our dataset, it reduced the execution time on GPUs up to 2.5 times. Nevertheless, we omit this feature in the experiments as it did not help most of the graphs in our dataset.

### V. EXPERIMENTS

We evaluate the performance of the proposed coloring algorithms on two different manycore architectures: GPUs and Intel Xeon Phi (MIC). We use single nodes of the clusters *Shannon* (GPU) and *Compton* (Xeon Phi) at Sandia. A single node in Shannon has two NVIDIA Tesla K20X GPUs with compute capability 3.5 and 6 GB of global memory. GPU cards have 2688 cores with peak memory bandwidth 250 GB/s. Each node on Compton has a Xeon Phi MIC card with 57 cores and 4 hyperthreads at 1.1 GHz with 6 GB memory. As all algorithms had their peak performance with 228 threads, each algorithm is run with fully utilized MIC threads for the experiments. The proposed algorithms are implemented using the Kokkos Library, and compiled using the version within the Trilinos 12.2 release, with Cuda 7.5.7 and icc 15.0.2.

For the experiments, we used graphs from the UFL Sparse Matrix Collection [3]. The graphs and their properties are listed in Table I. We used no reordering. We evaluate the performance of the coloring algorithms in terms of execution time, as well as the number of colors found by the algorithms.

TABLE I: The graphs used in the experiments. In a preprocessing step, each graph is symmetrized, and self edges are removed. The reported number of edges corresponds to the number of edges after this preprocessing.  $\delta_{avg}$  is the average degree in the graph ( $\frac{|E|}{|V|}$ ). The last column shows the ratio of the standard deviation of the vertex degrees to the average degree, as a measure of the regularity of the graph. Graphs in which vertices have similar degrees have lower ratios (regular graphs), while those with highly varying degrees have larger ratios (irregular graphs).

Graph	Class	V	E	$\delta_{avg}$	$\frac{\text{std}(\delta(v))}{\delta_{avg}}$
circuit5M	Circuit Problem	5.5M	53.9M	9.71	139.72
Audikw_1	PDE Problem	0.9M	76.7M	81.28	0.52
Bump_2911	PDE Problem	2.9M	124.8M	42.87	0.16
Queen_4147	PDE Problem	4.1M	325.3M	78.45	0.08
kron_g500-logn21	Artificial Network	2.0M	182.1M	86.82	8.70
indochina-2004	Web Crawl	47.4M	302.0M	40.72	61.35
hollywood-2009	Social Network	1.1M	112.8M	98.91	2.75
rgg_n_2_24_s0	RandomGeometric	16.8M	265.1M	15.80	0.25
soc-LiveJournal1	Social Network	4.8M	85.7M	17.68	2.94
europa_osm	Road Network	50.9M	108.1M	2.12	0.23

#### A. Graph Coloring Performance

In Figure 2, we present the execution time and the number of rounds of the coloring algorithms for seven representative graphs from our dataset. (Other graphs had similar results and are not shown here to save space.) The reported values denote averages of 5 and 10 runs for each graph on GPU and MIC, respectively. We have three main algorithms: two vertex-based methods (VB, VB BIT), and an edge-based algorithm (EB). Vertex-based algorithms have two variants each: using edge-filtering (EF) or not. The edge-based algorithm also has two variants: ATOMIC and PPS. We compare our implementations with the coloring method in cuSPARSE on GPUs, and IPGC implementation by [12] on MICs.

Figure 2 shows that EB usually outperforms other algorithms on GPUs, while its performance is slightly worse than others on MIC. Since MIC threads have more coarse-grain parallelism, the architecture is more “forgiving” to thread load-imbalances than GPUs, which have finer-grain parallelism (SIMD). Moreover, EB uses atomic operations, which are faster on GPU. However, edge-based algorithms still have advantage over the vertex-based ones on highly irregular graphs. We expect the architectures to be more sensitive to the thread load-imbalances with increasing number of threads, which is likely to increase the importance of the edge-based algorithms.

The number of rounds with EB is significantly higher than with the vertex-based algorithms. But since the execution time of a round is much lower for EB, it still obtains significant speed-ups on GPU (up to 86X) w.r.t. vertex-based algorithms, especially on irregular graphs. EB with PPS usually performs better on both architectures with a few exceptions.  $O(|E|)$  atomic operations by many threads create contention and may become quite expensive. In addition, atomic operations randomly distributes the consecutive edges, harming coalesced memory access on GPUs and caching on MICs. However, ATOMIC reduces the number of rounds on GPUs. Since edges are scattered, edges that are connected to the same source vertex are processed by different GPU warps in the tentative

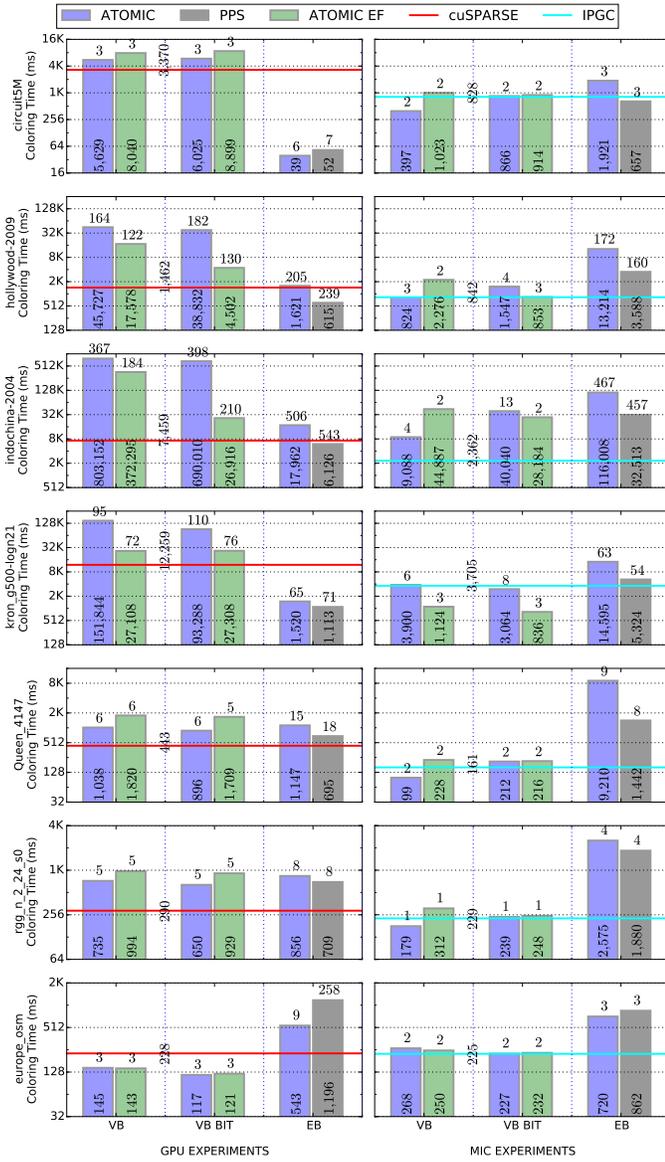


Fig. 2: Experiment results for seven graphs. Each row corresponds to a specific matrix, and left and right columns correspond to GPU and MIC experiments, respectively. Note the  $y$  axis is log scaled, has different ranges for different matrices, and it is given in milliseconds for the sake for presentation. The numbers inside the bars correspond to the execution time in milliseconds. The numbers on top of the bars show the number of rounds each algorithm takes.

coloring. This significantly reduces the number of conflicts, and, thus, the number of rounds. However, this reduction usually does not amortize the cost of the atomic operations and non-coalesced edge access, except for europe\_osm, in which the number of rounds is reduced dramatically.

In general, vertex-based algorithms have better performance on MICs, except for europe\_osm. This graph has a very low  $\delta$  and  $\text{std}(\delta(v))$ . Therefore, threads do not suffer from load-imbalances and have better coalesced accesses, which allows the vertex-based algorithms to utilize the GPU better. Similarly, vertex-based algorithms are slightly better on

rgg\_n\_2\_24\_s0 on GPUs. Even though this graph has higher  $\delta$ , it is a very regular graph. Vertices have similar number of neighbors, which eliminates the thread-divergence and load-imbalance of vertex-based algorithms.

Among the vertex-based algorithms, VB BIT is slightly better on GPUs, while VB has better performance on MIC. The forbidden array in VB is stored on a slow local memory on GPUs, and avoiding those accesses with VB BIT improves performance up to 13.8 times (indochina-2004 ATOMIC EF). The cost of the extra bit operations is amortized by avoiding local memory accesses. On the other hand, storage of FORBIDDEN can take advantage of caching on MIC, reducing the cost of the accesses to the array. As a result, the extra bit operations in VB BIT become more expensive than the accesses to the arrays, making VB BIT slower w.r.t. VB.

EF usually helps on graphs for which the number of colors is high (higher than 64, the fixed forbidden size) e.g. hollywood-2009, indochina-2004 and kron-g500-logn21. For example, edge-filtering for indochina-2004 on GPUs reduced one instance's execution time from 690 seconds to 27 seconds. EF reduces the thread-divergence in these graphs. In addition, since it reduces the time that a greedy coloring decision takes (making the color information available faster), it helps to reduce the number of conflicts in each round, and reduce the number of rounds. However, the extra memory read and writes have a negative impact for the graphs with fewer colors. Moreover, on MIC, edge-filtering improvements are slightly smaller than those on GPUs. Since thread-divergence is not an issue on MIC, EF helped to reduce only the number of rounds, which amortizes the cost of extra memory overhead only in a few instances (e.g., kron-g500-logn21).

As expected, cuSPARSE is usually much faster than the vertex-based algorithms on GPUs, but uses many more (about 4X) colors. EB outperforms cuSPARSE on irregular graphs; however, its run time is slightly worse on more regular ones.

IPGC usually obtains similar performances to the vertex-based algorithms, and it does not run out of memory on MIC for our dataset. However, the memory is likely to be a problem for IPGC as the number of threads increases. Moreover, VB often had lower execution time than IPGC. We believe that the use of the smaller memory made the algorithm more cache-friendly, improved the performance over IPGC in most of the graphs. However, IPGC is significantly faster on indochina-2004, in which the number of colors found is roughly 7K, which causes 100 times more edge-list traversals for VB.

In Table II, we report the average number of colors found by the algorithms with multiple runs. Only single variants are shown for vertex- and edge-based algorithms, as results do not change significantly with other variants. The fewest colors are usually found by EB, which, on average, has 4.12 times fewer colors than cuSPARSE and 8% fewer than VB.

Table III shows the geometric mean of the overall coloring execution time of the coloring algorithms. EB with PPS has the best execution time on GPU: 1.49 times faster than cuSPARSE. On the other hand, VB obtains the overall best execution time on MIC: 10% faster than IPGC. Figure 3

TABLE II: The average number of colors found

GPU	GPU			MIC		
	cuSPARSE	VB	EB	IPGC	VB	EB
audikw_1	160.0	56.2	50.4	60.4	60.9	51.9
Bump_2911	96.0	33.4	32.0	38.7	39.0	36.3
circuit5M	341.0	9.4	7.0	7.0	8.0	6.1
hollywood-2009	2317.0	2209.0	2209.0	2209.0	2211.3	2209.0
indochina-2004	7030.0	6849.4	6850.2	6849.0	6848.0	6849.7
kron_g500-logn21	3465.0	832.2	645.6	799.8	825.8	632.9
Queen_4147	128.0	49.8	44.0	54.0	51.0	48.4
soc-LiveJournal1	557.0	344.6	330.4	332.3	332.0	330.5
rgg_n_2_24_s0	64.0	24.8	24.8	24.0	23.0	25.0
europa_osm	32.0	6.0	6.0	5.0	5.0	5.0
GEOMEAN:	312.37	82.09	75.84	88.61	89.14	75.28

TABLE III: Geometric mean of execution times (seconds)

GPU:	cuSPARSE	VB	VB EF	VBBIT	VBBIT EF	EB ATOMIC	EB PPS
	0.79	3.38	2.96	2.84	1.92	0.73	0.53
MIC	IPGC	VB	VB EF	VBBIT	VBBIT EF	EB ATOMIC	EB PPS
	0.42	0.38	0.66	0.64	0.53	5.64	1.78

shows the performance profile of the algorithms on GPUs and MICs. The best performance is measured among the reported variants of the algorithms. As seen in Figure 3a, EB, cuSPARSE and VB BIT have best performance for 5, 4 and 1 graphs, respectively. cuSPARSE and EB have the overall best performance, but EB obtains far fewer colors than cuSPARSE. On MIC, VB, IPGC, and VB BIT obtain the best performance on 7, 2 and 1 graphs, respectively.

### B. Scaling of coloring

Figure 4 presents the scalability of the algorithms in terms of the size of the graph. We perform this experiment using RMAT graphs from the Graph500 benchmark ( $a = .57, b = c = .19$ ); see Table IV. Figure 4 shows the execution time per million edges. On GPU, we observe all algorithms except VB have better than linear scaling with the increasing size of the graph. For these highly irregular graphs, EF greatly benefits the vertex-based algorithms. The best performance is achieved by EB. On the other hand, on MIC (Figure 4b), the algorithms show less than linear scaling. One interesting result is that EB is much faster than the vertex-based algorithms, even on MIC. Because RMAT graphs have much greater irregularity than the graphs in Table I, the vertex-based algorithms suffer from thread load-imbalances, even on MIC.

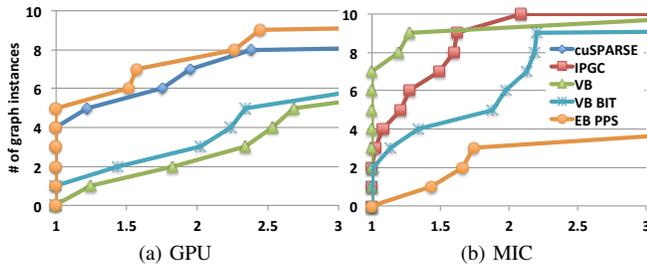


Fig. 3: Performance profile of the algorithms' execution time. Point  $(x, y)$  indicates that in  $y$  graph instances, the algorithm obtains a performance that is at most  $x$  times worse than the best run time.

TABLE IV: The RMAT graphs used in the scaling experiments

Graph	$ V $	$ E $	$\delta_{avg}$	$\frac{std(\delta(v))}{\delta_{avg}}$
rmat_20	1M	8.4M	8.00	754.31
rmat_22	4.2M	33.6M	8.00	1293.60
rmat_24	16.8M	134.2M	8.00	2200.37

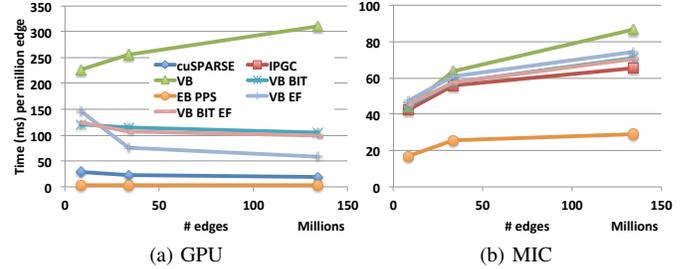


Fig. 4: Scaling results for increasing graph size. The  $y$  axis is execution time per million edges. A flat line represents linear scaling.

### C. Application: Multicolor Gauss-Seidel

To show the importance of coloring on a real application, we solve linear systems  $Ax = b$  using the Symmetric Gauss-Seidel (SGS) method as preconditioner for the conjugate gradient (CG) method. The Gauss-Seidel method is the stationary iteration  $(L + D)x^{k+1} = Ux^k$  [18], where  $L, D$ , and  $U$  are the lower, diagonal, and upper parts of the matrix  $A$ , respectively. The algorithm traverses rows in a certain (e.g., natural) order, and within the execution,  $x$ 's values are updated dynamically, and used in the calculation of the later  $x$  elements corresponding to other rows. Therefore, old values of  $x$  are multiplied with the upper diagonal, while the updated  $x$  values are multiplied with the lower diagonal. These dependencies make the algorithm difficult to parallelize. Graph coloring can be used to find sets of independent rows that can be processed simultaneously. Fewer colors correspond to fewer synchronization points in SGS, and more parallelism between synchronization points. This changes the traversal order, and this matrix reordering may impact the convergence rate [18].

We implemented multicolored SGS and preconditioned CG (PCG) using Kokkos and used the coloring from cuSPARSE and EB on various graphs. We used the FENL example from Trilinos with various problem sizes. FENL solves a nonlinear advection-diffusion equation, discretized using linear finite elements. The resulting nonlinear system is solved with Newton's method, which at each iteration solves a linear system. The number of colors found, the number of iterations that CG and PCG took, their overall execution times, and the preconditioning time are shown in Table V.

Unpreconditioned CG did not complete in 12 hours for Queen\_4147, Bump\_2911, and audikw\_1. For the FENL problems, using PCG with SGS reduced the number of iterations up to 2.88 times, and overall execution times up to 39%. EB usually finds roughly 4 times fewer colors than cuSPARSE in these problems. This difference reduces the overall execution time by 26% to 7%, and preconditioning time (the actual

TABLE V: The overall execution time and SGS preconditioning time (in seconds) of (P)CG Solver with different coloring methods.

Matrix	Algorithm	# Colors	# Iterations	SGS Time	PCG Time
Fenl_50	EB	17.2	85	<b>0.25</b>	<b>0.54</b>
	cuSPARSE	64	76	0.46	0.73
	CG		219		0.73
Fenl_100	EB	18	162	<b>2.38</b>	<b>5.84</b>
	cuSPARSE	64	148	3.07	6.24
	CG		425		8.94
Fenl_180	EB	18	266	<b>22.70</b>	<b>55.90</b>
	cuSPARSE	74	263	28.40	61.20
	CG		736		91
G3_Circuit	EB	5	1,897	<b>6.84</b>	<b>12.17</b>
	cuSPARSE	32	1,701	13.3	18.05
	CG		16,855		42.30
audikw_1	EB	50	2,780	<b>136.44</b>	<b>337.52</b>
	cuSPARSE	160	2,708	179.75	375.64
Bump_2911	EB	33	6,110	<b>411.96</b>	<b>1107.11</b>
	cuSPARSE	96	5,990	507.05	1188.56
Queen_4147	EB	44	13,504	<b>2376.45</b>	<b>6575.64</b>
	cuSPARSE	128	13,789	2788.34	7076.42

parallelized region with coloring) by 56% to 20% w.r.t. cuSPARSE. There is a small increase in the number of iterations when using a better coloring (fewer colors). The execution time difference is larger on the smaller problems, since their cost of synchronization is more visible.

On G3\_Circuit, SGS reduces the number of iterations up to 90%, and overall execution time up to 71% w.r.t. CG. For this matrix, cuSPARSE finds 6.4 times more colors than EB, making it 94% and 48% slower on preconditioning and overall time, respectively. Similarly, on larger matrices (audikw\_1, Bump\_2911 and Queen\_4147) EB reduces preconditioning time from 15-25%, and overall solve time 7-10% w.r.t. cuSPARSE, by reducing the synchronization points about 66%.

## VI. CONCLUSION

We have presented a new parallel, optimistic, edge-based algorithm for the graph coloring problem. We have also proposed several improvements to the traditional speculative vertex-based algorithm, and discussed implementation issues on manycore architectures. Experiments show that the edge-based method is usually faster on GPU but vertex-based is usually faster on the Xeon Phi (except for highly irregular graphs such as RMAT). The number of colors is typically slightly lower for the edge-based algorithm than the vertex-based one, but they both produce significantly fewer colors (typically 4X, but up to 48X) than that of cuSPARSE. Our results show that it is challenging to develop graph algorithms for manycore architectures. The best algorithm depends not only on the architecture, but also on the graph. For applications using graph coloring, the quality of the coloring is often more important than the run time. We demonstrated how reducing the number of colors benefits the solution of linear systems using a multi-colored Gauss-Seidel preconditioner. Future work includes studying vertex ordering, which may reduce the number of colors but at some cost in run time. We conjecture our algorithms can be extended to other coloring problems, such as balanced coloring [19] and distance-2 coloring.

**Acknowledgements:** We thank Erik Saule and Ümit Çatalyürek for providing their coloring code [12]. Sandia Na-

tional Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Dept. of Energy's National Nuclear Security Administration (NNSA) under contract DE-AC04-94AL85000. This work is supported by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, and by the NNSA's Advanced Simulation and Computing (ASC) program.

## REFERENCES

- [1] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency: Practice & Experience*, vol. 12, no. 12, pp. 1131–1146, 2000.
- [2] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J Parallel Distrib Comp*, vol. 74, no. 12, pp. 3202–3216, 2014.
- [3] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans Math Softw*, vol. 38, no. 1, pp. 1–25, 2011.
- [4] D. Zuckerman, "Linear degree extractors and the inapproximability of max clique and chromatic number," in *Proc 38th ACM Symp Theory of Computing*, 2006, pp. 681–690.
- [5] D. W. Matula, G. Marble, and J. Isaacson, "Graph coloring algorithms," in *Graph Theory and Computing*, R. Read, Ed. Academic Press, 1972, pp. 109–122.
- [6] 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. 1–31, 2013.
- [7] M. T. Jones and P. Plassmann, "A parallel graph coloring heuristic," *SIAM J Sci Comput*, vol. 14, no. 3, pp. 654–669, 1993.
- [8] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM J Comput*, vol. 15, no. 4, pp. 1036–1055, 1986.
- [9] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *Proc 26th ACM Symp Parallelism Algorithms Architectures (SPAA'14)*, 2014, pp. 166–177.
- [10] D. Bozdag, A. H. Gebremedhin, F. Manne, E. G. Boman, and U. V. Çatalyürek, "A framework for scalable greedy coloring on distributed-memory parallel computers," *J Parallel Distrib Comp*, vol. 68, no. 4, pp. 515–535, 2008.
- [11] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multi-core and massively multithreaded architectures," *Parallel Comput*, vol. 38, no. 10, pp. 576–594, 2012.
- [12] E. Saule and Ü. V. Çatalyürek, "An early evaluation of the scalability of graph algorithms on the intel mic architecture," in *Proc 26th IEEE Internat Parallel Distrib Proc Symp Workshops & PhD Forum (IPDPSW12)*, 2012, pp. 1629–1639.
- [13] A. E. Sariyuce, E. Saule, and U. V. Çatalyürek, "Scalable hybrid implementation of graph coloring using mpi and openmp," in *Proc IEEE 26th Internat Parallel & Distrib Proc Symp Workshops & PhD Forum (IPDPSW12)*, 2012, pp. 1744–1753.
- [14] G. Rokos, G. Gorman, and P. Kelly, "A fast and scalable graph coloring algorithm for multi-core and many-core architectures," in *Euro-Par 2015: Parallel Processing*, ser. Lecture Notes in Computer Science, J. L. Träff, S. Hunold, and F. Versaci, Eds. Springer Berlin Heidelberg, 2015, vol. 9233, pp. 414–425.
- [15] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, "Evaluating graph coloring on GPUs," in *Proc 16th ACM Symp Principles Practice Parallel Prog (PPoPP'11)*, 2011, pp. 297–298.
- [16] M. Naumov, P. Castonguay, and J. Cohen, "Parallel graph coloring with applications to the incomplete-LU factorization on the GPU," NVIDIA, Tech. Rep., 2015.
- [17] S. Che, G. Rodgers, B. Beckmann, and S. Reinhardt, "Graph coloring on the GPU and some techniques to improve load imbalance," in *Proc 29th IEEE Internat Parallel Distrib Proc Symp Workshop (IPDPSW15)*, 2015, pp. 610–617.
- [18] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.
- [19] H. Lu, M. Halappanavar, D. Chavarria-Miranda, A. Gebremedhin, and A. Kalyanaraman, "Balanced coloring for parallel computing applications," in *Proc 29th IEEE Internat Parallel Distrib Proc Symp (IPDPS15)*, 2015, pp. 7–16.