High-Performance and Balanced Parallel Graph Coloring on Multicore Platforms

Christina Giannoula[1], Athanasios Peppas[1], Georgios Goumas[1] and Nectarios Koziris[1]

[1]School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece.

Contributing authors: cgiannoula@cslab.ece.ntua.gr; apeppas@cslab.ece.ntua.gr; goumas@cslab.ece.ntua.gr ; nkoziris@cslab.ece.ntua.gr;

Abstract

Graph coloring is widely used to parallelize scientific applications by identifying subsets of independent tasks that can be executed simultaneously. Graph coloring assigns colors the vertices of a graph, such that no adjacent vertices have the same color. The number of colors used corresponds to the number of parallel steps in a real-world end-application. Therefore, the total runtime of the graph coloring kernel adds to the overall parallel overhead of the real-world end-application, whereas the number of the vertices of each color class determines the number of the independent concurrent tasks of each parallel step, thus affecting the amount of parallelism and hardware resource utilization in the execution of the real-world end-application. In this work, we propose a high-performance graph coloring algorithm, named ColorTM, that leverages Hardware Transactional Memory (HTM) to detect coloring inconsistencies between adjacent vertices. ColorTM detects and resolves coloring inconsistencies between adjacent vertices with an eager approach to minimize data access costs, and implements a speculative synchronization scheme to minimize synchronization costs and increase parallelism. We extend our proposed algorithmic design to propose a balanced graph coloring algorithm, named BalColorTM, with which all color classes include almost the same number of vertices to achieve high parallelism and resource utilization in the execution of the real-world end-applications. We evaluate ColorTM and BalColorTM using a wide variety of large real-world graphs with diverse characteristics. ColorTM and BalColorTM improve performance by 12.98× and 1.78× on average using 56 parallel threads compared to prior state-of-the-art approaches. Moreover, we study the impact of our proposed graph coloring algorithmic designs on a popular end-application, i.e., Community Detection, and demonstrate the ColorTM and BalColorTM can provide high performance improvements in real-world end-applications across various input data given.

Keywords: parallel graph coloring, balanced graph coloring, distance-1 coloring, pagerank, community detection, multicore architectures, shared memory architectures, parallel graph algorithms, high-performance computing, HPC

1 Introduction

Graph coloring assigns colors to the vertices of a graph such that any two adjacent vertices have different colors. Graph coloring kernel is widely used in many important real-world applications including the conflicting job scheduling [1–5], register allocation [6–10], sparse linear algebra [11–14], machine learning (e.g., to select non-similar samples that form an effective training set), and chromatic scheduling of graph processing applications [15–18]. For instance, the chromatic scheduling execution is as follows: given the vertex coloring of a graph, chromatic scheduling performs N steps that are executed *serially*, where N is the number of colors used to color the graph, and at each step the vertices assigned to the *same* color are processed *in parallel*, i.e., representing independent tasks that are executed concurrently. In addition, it is of vital importance that programmers manage the registers of modern CPUs effectively, and thus compilers [9, 10] optimize the register allocation problem via graph coloring: compilers construct undirected graphs, named register inference graphs (RIGs), with vertices representing the variables used in the source code and edges between vertices representing variables that are simultaneously active at some point in the program execution, and then compilers leverage the graph coloring kernel to identify independent variables that can be allocated on the same registers, i.e., if there *no* edge in the RIG connecting the associated vertices of the variables.

To achieve high system performance in the aforementioned real-world scenarios, software designers need to improve three key aspects of the graph coloring kernel. First, they need to minimize the number of colors used to color the input graph. For example, in the chromatic scheduling scheme minimizing the number of colors used to color the graph reduces the number of the sequential steps performed in the multithreaded end-application. However, minimizing the number of colors in graph coloring is an NP-hard problem [19], and thus prior works [3, 4, 11, 20–26] introduce ordering heuristics that generate effective graph colorings with a relatively small number of colors. Second, given that the execution time of the graph coloring kernel adds to the overall parallel overhead of the real-world end-application, software engineers need to design high-performance graph coloring algorithms for modern multicore computing systems. Third, an effective graph coloring for a real-world end-application necessitates a balanced distribution of the vertices across the color classes, i.e., the sizes of the color classes to be almost the same. Producing color classes with high skew in their sizes, i.e., high disparity in the number of vertices distributed across color classes, typically causes load imbalance and low resource utilization in real-world end-application. For example, in the register allocation scenario high disparity in the sizes of the color classes results to a large number of registers needed (high financial costs), equal to the size of the largest color class produced, while a *large* portion of the registers remains idle (unused) for a *long* time during the program execution (i.e., in time periods corresponding to many color classes with small sizes), thus causing low hardware resource utilization. Therefore, software designers need to propose *balanced* and *fast* graph coloring algorithms for commodity computing systems. Our **goal** in this work is to improve the last two key aspects of the graph coloring kernel by introducing high-performance and balanced multithreaded graph coloring algorithms for modern multicore platforms.

With a straightforward parallelization of graph coloring, coloring conflicts may arise when two parallel threads assign the same color to adjacent vertices. To deal with this problematic case, recent works [27–31] perform two additional phases: a conflict detection phase, which iterates over the vertices of the graph to detect coloring inconsistencies between adjacent vertices, and a conflict resolution phase, which iterates over the detected conflicted vertices to resolve the coloring inconsistencies via recoloring. Nevertheless, these prior works [27–31] are still inefficient, as we demonstrate in Section 5, because they need to traverse the *whole* graph at least two times (one for coloring the vertices and one for detecting coloring conflicts), and also detect and resolve coloring conflicts with a *lazy* approach, i.e., much later in the runtime compared to the time that the coloring conflicts appeared. As a result, prior approaches access the conflicted vertices of the graph multiple times, mainly using the expensive last levels of the memory hierarchy (e.g., main memory) of commodity multicore platforms, thus incurring high data access costs.

In this work, we present ColorTM [32][1], a high-performance graph coloring algorithm for multicore platforms. ColorTM is designed to provide low synchronization and data access costs. Our algorithm proposes (a) an *eager* conflict detection and resolution approach, i.e., immediately detecting and resolving coloring inconsistencies when they arise, such that to minimize data access costs by accessing conflicted vertices immediately using the low-cost lower levels of the memory hierarchy of multicore platforms, and (b) a *speculative* computation and synchronization scheme, i.e., leveraging Hardware Transactional Memory (HTM) and speculatively performing computations and data accesses *outside* the critical section, such that to provide high levels of parallelism and low synchronization costs by executing multiple small and short critical sections in parallel. Specifically, ColorTM consists of three steps: for each vertex on the graph, it (i) speculatively finds a candidate legal color by recording the colors of the adjacent vertices, (ii) validates and updates the

[1] This paper is an extended version of [33]

color of the current vertex by checking the colors of the critical adjacent vertices within an HTM transaction to detect potential coloring conflicts, and (iii) eagerly repeats steps (i) and (ii) for the current vertex multiple times until a valid coloring is found.

However, ColorTM does not provide any guarantee on the sizes of the color classes relative to each other. As we demonstrate in our evaluation (Section 5), the color classes produced by ColorTM for a real-world graphs have high disparity in the number of vertices across them, thus causing load imbalance and low resource utilization in real-world end-applications. Therefore, we extend our algorithmic design to propose a *balanced* graph coloring algorithm, named BalColorTM [32]. BalColorTM achieves high system performance and produces highly balanced color classes, i.e., having almost the same number of vertices across color classes, targeting to provide high hardware resource utilization and load balance in the real-world end-applications of graph coloring.

We evaluate ColorTM and BalColorTM on a dual socket Intel Haswell server using a wide variety of large real-world graphs with diverse characteristics. ColorTM improves performance by 12.98× on average using 56 parallel threads compared to state-of-the-art graph coloring algorithms, while providing similar coloring quality. BalColorTM outperforms prior state-of-the-art balanced graph coloring algorithms by 1.78× on average using 56 parallel threads, and provides the best color balancing quality over prior schemes (See Section 5). Finally, we study the effectiveness of our proposed ColorTM and BalColorTM in parallelizing a widely used real-world end-application, i.e., Community Detection [34], and demonstrate that our proposed algorithmic designs can provide significant performance improvements in real-world scenarios. ColorTM and BalColorTM are publicly available [32] at github.com/cgiannoula/ColorTM.

This paper makes the following contributions:

- We design high-performance and balanced graph coloring algorithms, named ColorTM and BalColorTM, for modern multicore platforms.
- We leverage HTM to efficiently detect coloring inconsistencies between adjacent vertices (processed by different parallel threads) with low synchronization costs. We propose an eager conflict resolution approach to efficiently resolve coloring inconsistencies in multithreaded executions by minimizing data access costs.
- We evaluate ColorTM and BalColorTM using a wide variety of large real-world graphs and demonstrate that they provide significant performance improvements over prior state-of-the-art graph coloring algorithms. Our proposed algorithmic designs significantly improve performance in multithreaded executions of real-world end-applications.

2 Prior Graph Coloring Algorithms

In this section, we describe prior state-of-the-art graph coloring algorithms [27–31]. Section 2.1 presents the sequential graph coloring algorithm. Section 2.2 describes prior parallel (no guarantee on the sizes of color classes) graph coloring algorithms proposed in the literature, while Section 2.3 presents prior balanced (color classes are highly balanced) graph coloring algorithms proposed in the literature.

2.1 The Greedy Algorithm

Figure 1 presents the sequential graph coloring algorithm, called *Greedy* [1]. Consider an undirected graph $G = (V, E)$, and the neighborhood $N(v)$ of a vertex $v \in V$ defined as $N(v) = \{u \in V : (v, u) \in E\}$. For each vertex v of the graph, Greedy records the colors of $v's$ adjacent vertices in a forbidden set of colors, and assigns the minimum legal color to the vertex v based on the forbidden set of colors.

```
Input: Graph G=(V,E)
Let N(v) be the adjacent vertices of the vertex v
for each $v \in V$ do
 forbidColors = $\emptyset$
 for each $u \in N(v)$ do
  forbidColors = forbidColors $\cup$ {$u$.color}
 $v$.color = minLegalColor(forbidColors)
```

Fig. 1 The Greedy algorithm.

The Greedy approach produces at most $\Delta + 1$ colors [1], where Δ is the degree of the graph G. The degree of the graph is defined as $\Delta = max_{v \in V}\{deg(v)\}$, where $deg(v)$ is the degree of a vertex v, which is the number of its adjacent vertices $deg(v) = |N(v)|$. However, finding the minimum number of colors to color a graph G is an NP-hard problem [35]. In this work, we have experimented with the *first-fit* ordering heuristic [1], in which the vertices of the graph are processed and colored in the order they appear in the input graph representation G, since this heuristic can provide high coloring quality based on prior works [1, 21, 36]. We leave the exploration of other ordering heuristics for future work.

2.2 Prior Parallel Graph Coloring Algorithms

To parallelize the graph coloring problem, the vertices of the graph are distributed among parallel threads. However, due to crossing edges, the coloring subproblems assigned to parallel threads are not independent, and the parallel algorithm may terminate with an invalid coloring. Specifically, a race condition arises when two parallel threads assign the same color to adjacent vertices. The algorithm implies that when a parallel thread updates the color of a vertex, the forbidden set of colors of the adjacent vertices has not been changed. Thus, the nature of this algorithm imposes that the reads to the colors of the adjacent vertices of a vertex v have to be executed *atomically* with the write-update to the color of the vertex v.

2.2.1 The SeqSolve Algorithm

Figure 2 presents the parallel graph coloring algorithm proposed by Gebremedhin et al. [28], henceforth referred to as SeqSolve. This algorithm consists of three steps: (i) multiple parallel threads iterate over the whole graph and speculatively color the vertices of the graph with *no* synchronization (lines 3-6), (ii) multiple parallel threads iterate over the whole graph and detect coloring inconsistencies that appeared in the (i) step (lines 7-13), and (iii) only *one* single thread resolves the detected coloring inconsistencies by re-coloring the conflicted vertices (lines 14-16). Since the (iii) step is executed by only a single thread, no coloring inconsistencies appear after this step. Note that when a coloring conflict arises between two adjacent vertices, only *one* of the involved adjacent vertices needs to be re-colored, e.g., using a simple order heuristic among the vertices (line 11).

```
 1 Input: Graph G=(V,E)
 2 Let $N(v)$ be the adjacent vertices of the vertex $v$
 3 // Speculative Coloring - Step (i)
 4 for each $v \in V$ do in parallel
 5   Assign the minimum legal color to the vertex $v$
 6 __barrier__
 7 // Detection of Coloring Inconsistencies - Step (ii)
 8 R = $\emptyset$ // Set of Conflicted Vertices
 9 for each $v \in V$ do in parallel
10   for each $u \in N(v)$ do
11     if (($v$.color == $u$.color) && ($v$ < $u$))
12       R = R $\cup$ v
13 __barrier__
14 // Sequential Resolution of Coloring Conflicts - Step (iii)
15 for each $v \in R$ do
16   Assign the minimum legal color to the vertex $v$
```

Fig. 2 The SeqSolve algorithm.

In the SeqSolve algorithm, we make three key observations. First, if the number of coloring conflicts arised in a multithreaded execution is low, the algorithm might scale well [28]. However, as the number of parallel threads increases and the graph becomes denser, i.e., the vertices of the graph have a large number of adjacent vertices, many more coloring conflicts arise in multithreaded executions. In such scenarios, a large number of coloring inconsistencies is resolved sequentially, i.e., by only *one* single thread, thus achieving limited parallelism. Second, we note SeqSolve traverses the whole graph at least two times, i.e., step (i) and step (ii). Assuming large real-world graphs that do not typically fit in the Last Level Cache (LLC) of contemporary multicore platforms, the whole graph is traversed twice using the main memory, thus incurring high data access costs. Third, we observe that SeqSolve detects and resolves the coloring conflicts *lazily*, i.e., much later in the runtime compared to the time that the coloring conflicts appears. Specifically, a coloring inconsistency in a vertex v might appear in step (i). However, SeqSolve detects the coloring inconsistency in vertex v in step (ii), i.e., after *first* coloring *all* the remaining vertices of the graph. Similarly, SeqSolve resolves the coloring inconsistency of the vertex v in step (iii), i.e., after *first* detecting if coloring inconsistencies exist in *all* the remaining vertices of the graph (step (ii)). As a result, many conflicted vertices are accessed multiple times in the runtime, however with a lazy approach, i.e., accessing them through the expensive last levels of the memory hierarchy of commodity platforms, thus incurring high data access costs.

2.2.2 The IterSolve Algorithm

Figure 3 presents the parallel graph coloring algorithm proposed by Boman et al. [27, 30], henceforth referred to as IterSolve. This algorithm consists of two repeated steps: (i) multiple parallel threads iterate over the uncolored vertices of the graph and speculatively color the uncolored vertices of the graph with *no* synchronization (lines 5-8), (ii) multiple parallel threads iterate over the recently-colored vertices of the graph and detect coloring inconsistencies appeared in the (i) step (lines 9-15). The steps (i) and (ii) are iteratively repeated until there are *no* coloring inconsistencies in any adjacent vertices of the graph.

```
 1 Input: Graph G=(V,E)
 2 Let $N(v)$ be the adjacent vertices of the vertex $v$
 3 $U = V$
 4 while $U \neq \emptyset$
 5   // Speculative Coloring - Step (i)
 6   for each $v \in U$ do in parallel
 7     Assign the minimum legal color to the vertex $v$
 8   __barrier__
 9   // Detection of Coloring Inconsistencies - Step (ii)
10   R = $\emptyset$ // Set of Conflicted Vertices
11   for each $v \in U$ do in parallel
12     for each $u \in N(v)$ do
13       if (($v$.color == $u$.color) && ($v$ < $u$))
14         R = R $\cup$ v
15  __barrier__
16   $U = R$
```

Fig. 3 The IterSolve algorithm.

In the IterSolve algorithm, we make four key observations. First, the programmer needs to explicitly define forward progress in the source code, so that the IterSolve algorithm terminates. Specifically, to ensure forward progress when a coloring inconsistency appears between two adjacent vertices, the programmer needs to *explicitly* define only *one* of them to be re-colored (line 13), e.g., based on the vertices' ids. Otherwise, the two adjacent vertices may *always* obtain the same color, if they are always being processed by different threads. Second, similarly to SeqSolve, IterSolve traverses the whole graph at least two times (steps (i) and (ii)), i.e., in the first iteration of the while loop in line 4, where the set U is equal to the set V (line 3). In the first iteration of the while loop, the whole large real-world graph is accessed through the main memory twice, thus incurring high data access costs. Third, similarly to SeqSolve, IterSolve detects and resolves the coloring conflicts *lazily*. Specifically, a coloring inconsistency in a vertex v might appear in step (i) (line 7), it is detected in step (ii) (line 13), i.e., after *first* coloring *all* the remaining uncolored vertices of the graph. Moreover, IterSolve resolves the coloring inconsistency of a vertex v in step (i) (with re-coloring), i.e., after *first* detecting if coloring inconsistencies exist in *all* the remaining recently-colored vertices of the graph (step (ii)). Thus, IterSolve incurs high data access costs on the many conflicted vertices, which are accessed multiple times in the runtime with *lazy* approach, through the last levels of the memory hierarchy of commodity platforms. Fourth, the iterative process of resolving coloring conflicts may introduce new conflicts, and thus, IterSove might need additional iterations to fix them. This scenario may happen when adjacent vertices are assigned to the *same* thread and incur coloring inconsistencies, they will be assigned and processed by *different* parallel threads in the next iteration. The authors of the original IterSolve papers [27, 30] empirically observe that a few iterations of IterSolve are needed to produce a valid coloring. However, the authors used *synthetic* and *not* real-world graphs in their evaluation. In addition, the more iterations are needed, the more *lazy* traversals on the conflicted vertices of the graph are performed, which can significantly degrade performance.

2.2.3 The IterSolveR Algorithm

Figure 4 presents the parallel graph coloring algorithm proposed by Rokos et al. [29], henceforth referred to as IterSolveR. Rokos et al. observed that the IterSolve algorithm (Figure 3) can be improved by merging the steps (i) and (ii) into a single *detect-and-re-color* step, thus eliminating one of the two barrier synchronizations of IterSolve (lines 8 and 15 in Figure 3). When a coloring inconsistency on a vertex v is found, the vertex v can be immediately re-colored (line 18 in Figure 4). However, the new re-coloring on the vertex v may *again* introduce a coloring inconsistency in multithreaded executions, since re-colorings are performed *concurrently* by multiple parallel threads (line 11). Therefore, the vertex v is marked as *recently-re-colored* vertex (line 19), and needs to be re-validated in the next iteration of IterSolveR. Overall, IterSolverR (Figure 4) first speculatively colors all the vertices of the graph and marks them as *recenlty-colored* vertices (lines 3- 6). Then, it executes one single repeated step (lines 8-21): multiple parallel threads iterate over the recently-colored vertices of the graph, and detect if coloring inconsistencies have appeared, which in that case are immediately resolved via re-coloring. This step is repeated until there are no *recently-re-colored* vertices: in one single iteration of this step, there are *no* coloring inconsistencies detected in any adjacent vertices of the graph.

```
 1 Input: Graph G=(V,E)
 2 Let $N(v)$ be the adjacent vertices of the vertex $v$
 3 // Speculative Coloring (Step 0)
 4 for each $v \in V$ do in parallel
 5     Assign the minimum legal color to the vertex $v$
 6 __barrier__
 7 $U = V$ // Mark all Vertices as Recently-Colored
 8 while $U \neq \emptyset$
 9   R = $\emptyset$ // Set of Recently-Colored Vertices
10   // Conflict Detection and Resolution (Step (i))
11   for each $v \in U$ do in parallel
12     bool conflict-detected = false
13     for each $u \in N(v)$ do
14       if (($v$.color == $u$.color) && ($v$ < $u$))
15         conflict-detected = true
16         break
17     if (conflict-detected == true)
18       Assign the minimum legal color to vertex $v$
19       R = R $\cup$ v // Mark vertex v as Recently-Colored
20   __barrier__
21   $U = R$
```

Fig. 4 The IterSolveR algorithm.

In the IterSolveR algorithm, even though one barrier synchronization is eliminated compared to IterSolve, we observe that IterSolveR still traverses the whole graph at least twice: (i) in Step 0 (lines 4-5), and (ii) in the first iteration of the while loop in line 8, where the set U is equal to the set V (line 7), including all the vertices of the graph. Thus, IterSolveR traverses the large real-world graph twice through the main memory, incurring high data access costs. In addition, we find that similarly to SeqSolve and IterSolve, the IterSolveR algorithm also detects the coloring inconsistencies *lazily*. Specifically, a coloring inconsistency on a vertex v might appear in the re-coloring process of lines 17-19, since the re-coloring process is *concurrently* executed on multiple conflicted vertices by multiple parallel threads. However, re-coloring inconsistencies of lines 17-19 are detected in the next iteration of the step (i) in lines 13-16, i.e., after *first* processing *all* the remaining vertices of the set U (line 11). Therefore, as we demonstrate in our evaluation (Section 5.2), IterSolveR is still inefficient, incurring high data access costs on multiple conflicted vertices which are accessed multiple times in the runtime with a *lazy* approach.

2.3 Prior Balanced Graph Coloring Algorithms

To provide a balanced coloring on a graph in which the color classes produced include almost the same number of vertices, an initial graph coloring is obtained using a balanced-oblivious algorithm (e.g., See Section 2.2), and subsequently the balanced graph coloring is obtained using a balanced-aware (henceforth referred to as balanced for simplicity) graph coloring algorithm, as we describe next. Specifically, given a graph $G = (V, E)$, we can assume that the number of colors produced by the initial coloring step (i) is C. A strictly balanced graph coloring results in the size of *each* color class being $b = V/C$. [2] Therefore, we refer to the color classes whose sizes are greater than b as *over-full* classes, and those whose sizes are less than b as *under-full* classes. Balanced graph coloring algorithms leverage the quantity of b, which can be extracted by first executing an initial balanced-oblivious graph coloring on the graph, in order to provide balanced color classes on a graph.

2.3.1 The Color-Centric (CLU) Algorithm

Figure 5 presents the color-centric balanced graph coloring algorithm proposed by Lu et al. [31], henceforth referred to as CLU. In this scheme, vertices belonging in the *same* color class are processed concurrently, and a *subset* of vertices from each over-full color class is moved to under-full color classes in order to achieve high color balance. Only vertices belonging in over-full color classes are considered for re-coloring, while graph coloring balance is achieved *without* increasing the number of color classes produced by the initial graph coloring.

The CLU algorithm (Figure 5) processes the over-full color classes sequentially (lines 6 and 16), while vertices belonging at the same over-full color class

[2] Please note that in our work we make the following assumption: in a real-world end-application, the vertices of the graph represent sub-tasks that have almost equal load/weights of computation. If the vertices of the input graph have different load/weights of computation, a pre-processing step needs to be applied in the original graph: vertices with large computation weights/load are split into multiple smaller vertices, each of them has one weight/load unit of computation.

are processed concurrently (line 8). CLU iterates over each vertex v of an over-full color class, and finds the minimum color of an under-full color class that is permissible to be assigned at the vertex v (line 11). If such a color exists, the vertex v is re-colored with a color of an under-full color class (lines 12-15). The CLU algorithm iterates over the vertices of each over-full color class until that particular over-full class becomes balanced at a certain point in the execution, i.e., until when its size becomes smaller or equal to b (lines 9-10). Then, the vertices belonging on that color class are no longer considered for re-coloring (line 10). Thus, this algorithm terminates when either vertex-balance across color classes is achieved or vertex-balance across color classes is no longer available, i.e., there are no more permissible re-colorings for any vertex belonging in an over-full color class.

```
1  Input: Graph G=(V,E)
2  Obtain an initial coloring on G
3  Let $C$ be the number of colors produced
4  Let $b = V/C$ be the perfect balance
5  Let $Q$ be the set of vertices of the over-full color classes
6  for each $c \in Q$ do // Process the Over-Full Color Classes
7    Let $R(c)$ be the set of vertices with color $c$
8    for each $v \in R(c)$ do in parallel
9      if (the size of the color class $c$ <= $b$)
10       continue // Color Class is Balanced
11     Let $k$ be the index of the minimum under-full color class that is
        permissible to vertex $v$
12     if ($k$ exists) // Re-Coloring
13       $v$.color = k
14       Atomically decrease the size of the color class $c$
15       Atomically increase the size of the color class $k$
16   __barrier__
```

Fig. 5 The CLU algorithm.

In the CLU algorithm, we make two key observations. First, parallel threads always process vertices of the *same* color, thus no coloring inconsistencies are produced: since vertices had the same color in the initial coloring, they are *not* adjacent vertices, and thus they can be re-colored with the same color of an under-full color class without violating correctness. This way, CLU requires *only* one iteration over the vertices of all the over-full color classes. Second, the parallel performance of CLU depends on the number of the over-full color classes produced in the initial coloring. CLU requires F steps, where F is the number of over-full color classes produced in the initial coloring. At *each* of these steps, i.e., for each over-full color class on the initial coloring, CLU introduces a barrier synchronization among parallel threads (line 16). This way, it increases the synchronization costs, which might significantly degrade scalability in multithreaded executions.

2.3.2 The Vertex-Centric (VFF) Algorithm

Figure 6 presents the vertex-centric balanced graph coloring algorithm proposed by Lu et al. [31], henceforth referred to as VFF. The VFF algorithm is the balanced graph coloring counterpart of the IterSolve algorithm (Figure 3). In this scheme, vertices from *different* color classes are processed concurrently by parallel threads. Thus, in contrast to CLU, VFF introduces coloring inconsistencies. However, similarly to CLU, in VFF only vertices belonging in over-full color classes are considered for re-coloring, i.e., to be moved to under-full color classes, while graph coloring balance is also achieved *without* increasing the number of color classes produced by the initial graph coloring.

Similarly to IterSolve, VFF (Figure 6) consists of two repeated steps: (i) multiple parallel threads iterate over vertices of over-full color classes and speculatively re-color them with permissible colors of under-full color classes, if possible (lines 8-18), and (ii) multiple parallel threads iterate over the recently re-colored vertices and detect coloring inconsistencies that appeared in the (i) step (lines 19-26). Similarly to CLU, VFF iterates over the vertices of an over-full color class until that particular over-full class becomes balanced at a certain point in the execution, i.e., until when its size becomes smaller or equal to b (lines 11-12). Then, the vertices belonging on that particular color class are no longer considered for re-coloring (line 12). The steps (i) and (ii) are iteratively repeated until there are *no* coloring inconsistencies in any adjacent vertices of the graph, and the algorithm terminates when either vertex-balance across color classes is achieved or vertex-balance across color classes is no longer available, i.e., there are no more permissible re-colorings for any vertex belonging in an over-full color class.

```
1  Input: Graph G=(V,E)
2  Let $N(v)$ be the adjacent vertices of the vertex $v$
3  Obtain an initial coloring on G
4  Let $C$ be the number of colors produced
5  Let $b = V/C$ be the perfect balance
6  Let $Q$ be the set of vertices of the over-full color classes
7  while $Q \neq \emptyset$ // Process the Over-Full Color Classes
8    // Speculative Re-Coloring - Step (i)
9    for each $v \in Q$ do in parallel
10     Let $c$ be the current color of the vertex $v$
11     if ((c != -1) && (the size of the color class $c$ <= $b$))
12       continue// Color Class is Balanced
13     Let $k$ be the index of the minimum under-full color class that is
        permissible to vertex $v$
14     if ($k$ exists)// Re-Coloring
15       $v$.color = k
16       Atomically decrease the size of the color class $c$
17       Atomically increase the size of the color class $k$
18   __barrier__
19   // Detection of Coloring Inconsistencies - Step (ii)
20   R = $\emptyset$ // Conflicted Vertices of Over-Full Color Classes
21   for each $v \in Q$ do in parallel
22     for each $u \in N(v)$ do
23       if (($v$.color == $u$.color) && ($v$ < $u$))
24         R = R $\cup$ v
25         $v$.color = -1
26  __barrier__
27   $Q = R$
```

Fig. 6 The VFF algorithm.

Since VFF is the balanced graph coloring counterpart of IterSolve, we report similar key observations for them. First, VFF detects and resolves the coloring conflicts *lazily*. Specifically, a coloring inconsistency in a vertex v might appear in step (i), while it is detected in step (ii), i.e., after *first* iterating over *all* the remaining vertices of over-full color classes. Moreover, VFF resolves the coloring inconsistency in a vertex v in step (i) (re-coloring), i.e., after *first* detecting if coloring inconsistencies exist in *all* the remaining recently re-colored vertices (in step (ii) of the previous iteration). Thus, VFF incurs high data access costs due to accessing multiple conflicted vertices in the runtime through the last levels of the memory hierarchy of commodity platforms. Second, the iterative process of resolving coloring conflicts may introduce new conflicts, and thus, VFF might need additional iterations to fix them. This scenario may happen when adjacent vertices are assigned to the *same* thread and incur coloring inconsistencies, they will be assigned and processed by *different* parallel threads in the next iteration. Note that the more iterations are needed, the more *lazy* traversals on the conflicted vertices of the graph are performed, which might significantly degrade performance.

2.3.3 The Recoloring Algorithm

Figure 7 presents the re-coloring balanced graph coloring algorithm proposed by Lu et al. [31], henceforth referred to as Recoloring. Recoloring is similar to the VFF (Figure 6) and IterSolve (Figure 3) schemes. The key idea of this algorithm is that after performing an initial graph coloring with C colors, *all* the vertices of the graph are re-colored, having an additional condition on the color selection in order to achieve better vertex balance across color classes compared to that produced by the initial graph coloring. Specifically, Recoloring leverages the perfect balance $b = V/C$ known from the initial graph coloring, and keeps track the sizes of the color classes during the execution in order to improve vertex balance across color classes as follows: each vertex is re-colored using the minimum permissible color k such that the size of the color class k is less than b.

Similarly to IterSolve and VFF, Recoloring (Figure 7) consists of two repeated steps: (i) multiple parallel threads iterate over all the vertices of the graph and speculatively re-color them with a new permissible color k, that satisfies the condition that the size of the color class k is less than b (lines 12-17), and (ii) multiple parallel threads iterate over the recently re-colored vertices and detect coloring inconsistencies that appeared in the (i) step (lines 18-25). The steps (i) and (ii) are iteratively repeated until there are *no* coloring inconsistencies in any adjacent vertices of the graph. In contrast to VFF and CLU, Recoloring does not guarantee that the graph color balance achieved uses the same number of colors with the initial graph coloring. To avoid producing a large number of color classes, the Recoloring scheme [31] (Figure 7) re-colors the vertices of the graph with the following order: assuming that the vertices of the graph are ordered such that the vertices of the same color class are listed consecutively (line 6), Recoloring iterates over the vertex sets of the color classes in the reverse order compared to that produced in the initial graph coloring, i.e., starting from the vertices assigned to the color class with the largest index (See line 8). The rationale behind this heuristic is that the vertices that are "difficult" to color, i.e., in the initial graph coloring they are assigned to a color class with large index, will be processed *early*, thus aiming to produce a small number of color classes. For more details, we refer the reader to [31].

```
Input: Graph G=(V,E)
Let $N(v)$ be the adjacent vertices of the vertex $v$
Obtain an initial coloring on G
Let $C$ be the number of colors produced
Let $b = V/C$ be the perfect balance
Let $K(j)$ be the set of vertices $u$ with color j
// $K(j) = \{u \in V, u.color = j\}$
Construct the order set $W = \{K(C), K(C-1), ..., K(1), K(0)\}$
Initialize the sizes of the $C$ color classes to 0
$Q = W$
while $Q \neq \emptyset$ // Re-Color the Whole Graph
  // Speculative Coloring - Step (i)
  for each $v \in Q$ do in parallel
    Let $k$ be the minimum color that is permissible to the vertex $v$ such that
     the size of the color class $k$ is less than $b$ // Balanced Color
     Classes
    $v$.color = k
    Atomically increase the size of the color class $k$
  __barrier__
  // Detection of Coloring Inconsistencies - Step (ii)
  R = $\emptyset$ // Set of Conflicted Vertices
  for each $v \in Q$ do in parallel
    for each $u \in N(v)$ do
      if (($v$.color == $u$.color) && ($v$ < $u$))
        Atomically decrease the size of the color class $v$.color
        R = R $\cup$ v
  __barrier__
  $Q = R$
```

Fig. 7 The Recoloring algorithm.

In Recoloring, we make three key observations. First, Recoloring traverses the whole graph, i.e., it re-colors *all* the vertices of the graph, while CLU and VFF re-color only a *subset* of the vertices of over-full color classes. As a result, Recoloring performs a much larger number of computations and memory accesses compared to VFF and CLU. Second, similarly to IterSolve and VFF, Recoloring detects and resolves coloring inconsistencies with a *lazy* approach, thus incurring high data access costs. Recoloring may also introduce new conflicts, thus resulting in additional iterations to fix them. Third, even though Recoloring employs a different vertex ordering heuristic to re-color vertices compared to that used in the initial graph coloring (vertices are colored with the order they appear in the input graph), there is *no* guarantee on the number of color classes that will be produced. As we demonstrate in our evaluation (Section 5.3), Recoloring might significantly increase the number of color classes produced compared to that produced in the initial graph coloring.

3 ColorTM: Overview

In the graph coloring kernel, there are three key optimization aspects: (i) minimize the number of colors used to color the vertices of the input graph, (ii) minimize the actual execution time it takes to color the vertices of the input graph, and (iii) minimize the actual execution time it takes to balance the vertices across the color classes produced. Our goal in this work is to improve the last two key aspects of the graph coloring kernel. Our proposed algorithmic design is a high-performance graph coloring algorithm for multicore platforms. ColorTM provides low synchronization and data access costs by relying on two key techniques, that we describe in detail next.

3.1 Speculative Computation and Synchronization

As already discussed, the graph coloring kernel implies that the reads to the colors of the adjacent vertices of a vertex v have to be executed atomically with the write-update to the color of the vertex v. Figure 8 presents a straightforward parallelization scheme of the graph coloring problem. A naive parallelization approach would be to distributed the vertices of the graph across parallel threads, and for each vertex to include within a critical section the whole block of code that computes and assigns a permissible color to that vertex. However, this approach results to large critical sections with large data access footprints and long duration, and significantly limits the amount of parallelism and the scalability to a large number of threads.

```
Input: Graph G=(V,E)
for each $v \in V$ do in parallel
  // Atomic Coloring Step (i)
  begin_critical_section
  Compute and assign the minimum legal color to the vertex $v$
  end_critical_section
```

Fig. 8 A Naive Approach.

We observe that it is not necessary to include inside the critical section (i) the computations performed to find a permissible color for a vertex v, and (ii) the accesses to *all* the adjacent vertices of the vertex v. Figure 9 presents an overview of ColorTM. For each vertex v, we design ColorTM to implement a speculative computation scheme through two sub-steps: (i) speculatively compute a permissible color k for the vertex v (line 5) without using synchronization and track the set of critical adjacent vertices (line 6), i.e., a subset of v's adjacent vertices that can cause coloring inconsistencies with the vertex v (See Section 4.2 for more details), and (ii) execute a critical section (using synchronization) that validates the speculative color k computed in step (i) over the colors of the critical adjacent vertices (lines 8-9) and assigns the color k to the vertex v, if the validation succeeds (lines 10-14). With the proposed speculative computation scheme, we provide small critical sections, i.e., having small data access footprints and short duration, thus achieving high amount of parallelism and high scalability to a large number of threads.

```
1  Input: Graph G=(V,E)
2  for each $v \in V$ do in parallel
3    RETRY:
4    // Speculative Computation
5    Compute a speculative minimum color $k$ that
        is permissible to the vertex $v$
6    Keep track the critical adjacent vertices
        of the vertex $v$
7    begin_critical_section
8    // Validate Coloring
9    Compare $k$ with the colors of the critical
        adjacent vertices
10   if (no coloring conflict)
11     v.color = $k$
12     end_critical_section
13   else
14     end_critical_section
15     goto RETRY // Eager Resolution
```

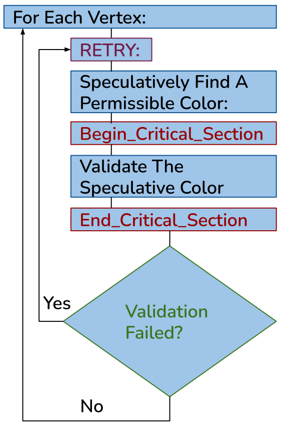

Fig. 9 ColorTM: Overview.

In addition, we leverage Hardware Transactional Memory (HTM) to implement synchronization on critical sections (lines 7, 12, and 14 of Figure 9). HTM enables a speculative synchronization mechanism: multiple critical sections of parallel threads are executed concurrently with an optimistic approach that they will not cause any data inconsistency, even though their data access sets might overlap. In contrast, fine-grained locking with software-based locks (e.g., provided by the pthread library) constitutes a more conservative synchronization approach: multiple critical sections of parallel threads are executed concurrently, *only* if their data access sets do *not* overlap. Therefore, HTM can enable a higher number of critical sections to be executed in parallel compared to that enabled with the fine-grained locking scheme. We provide more details in Section 4.1. With the speculative synchronization approach of HTM, ColorTM further minimizes synchronization costs and provides high amount of parallelism.

3.2 Eager Coloring Conflict Detection and Resolution

We design ColorTM to detect and resolve coloring inconsistencies *eagerly*, i.e., immediately detecting and resolving coloring inconsistencies at the time that the coloring conflicts appear. This way, the conflicted vertices are accessed multiple times, however within a short time during runtime. Therefore, application data corresponding to conflicted vertices can remain and be located in the first levels of the memory hierarchy of commodity platforms (i.e., in the low-cost cache memories), thus enabling ColorTM to improve performance by achieving low data access costs.

In Figure 9, parallel threads concurrently compute speculative colors for multiple vertices of the graph (lines 4-6), and at that time coloring inconsistencies may appear. Then, parallel threads *immediately* detect possible coloring conflict inconsistencies for the current vertices using synchronization (lines 7-14). This way, parallel threads detect conflicts by accessing application data with low access latencies, since the data accessed in lines 7-14 has just been accessed within a short time, i.e., in lines 4-6. Next, if coloring conflicts arise (line 13), parallel threads *immediately* resolve the coloring conflicts by directly retrying to find new colors for the current vertices (`goto RETRY` inline 15) (without proceeding to process new vertices). This way, parallel threads resolve conflict inconsistencies by accessing application data with low access latencies, since the data accessed in lines 4-6 after the execution of `goto RETRY` has just been accessed within a short time, i.e., in lines 7-14 of the previous iteration.

In ColorTM, we highlight two important key design choices. First, ColorTM executes only *one* single parallel step (line 2). In contrast to prior state-of-the-art parallel graph coloring algorithms [27–31], ColorTM *completely* avoids barrier synchronization among parallel threads: multiple parallel threads repeatedly iterate over each vertex of the graph until a valid coloring is found. By completely avoiding barrier synchronization, ColorTM can provide high scability. Second, ColorTM does not perform re-colorings to vertices: once a vertex is assigned a permissible color, it will *not* be re-colored again during the runtime. This way, colored vertices will *not* introduce coloring inconsistencies with vertices that will be processed next. Prior *lazy* iterative graph coloring schemes including IterSolve, IterSolveR, VFF and Recoloring do *not* use data synchronization when they assign permissible colors to vertices. This way, many vertices are re-colored multiple times with different colors during runtime, and thus new additional coloring inconsistencies might be introduced due to re-colorings. Instead, ColorTM employs HTM synchronization (lines 7, 12 and 14 of Figure 9) when it assigns permissible colors to vertices (line 11 of Figure 9). This way, vertices are assigned only *one* final color during the runtime, thus avoiding introducing new coloring inconsistencies due to re-colorings.

4 ColorTM: Detailed Design

ColorTM [33] is a high-performance graph coloring algorithm that leverages HTM to implement synchronization among parallel threads, and performs speculative computations outside the critical section in order to minimize the memory footprint and computations executed inside the critical section. In the section, we describe the detailed design and correctness of ColorTM. We also extend our proposed design to introduce a new balanced graph coloring algorithm, named BalColorTM, which evenly distributes the vertices of the graph across color classes.

4.1 Speculative Synchronization via HTM

ColorTM leverages HTM to implement synchronization among parallel threads instead of using fine-grained locking. As already discussed, HTM is a more optimistic synchronization approach and can provide higher levels of parallelism compared to the fine-grained locking scheme. Specifically, multiple critical sections with *overlapped* data access sets can be executed in parallel with HTM, while they need to be executed sequentially with fine-grained locking.

Figure 10 provides an example of the aforementioned scenario in graph coloring. Consider the scenario where thread $T1$ attempts to assign a color to the vertex v, and thread $T2$ attempts to assign a color to the vertex x. Thread $T1$ needs to *atomically* read the colors of the adjacent vertices of the vertex v, i.e., u, r, z vertices, and write the corresponding color to the vertex v. Similarly, Thread $T2$ needs to *atomically* read the colors of the adjacent vertices of the vertex x, i.e., u vertex, and write the corresponding color to the vertex x. With HTM (Figure 10a), $T1$'s and $T2$'s transactions can be executed and committed concurrently: neither the write-set of $T1$'s transaction does *not* conflict with the read-set of $T2$'s transaction, nor the write-set of $T2$'s transaction does *not* conflict with the read-set of $T1$'s transaction. Therefore, even though $T1$'s and $T2$'s critical sections have *overlapped* data access sets, i.e., both of them include the color of the vertex u in their read-sets, they can be executed concurrently with HTM. In contrast, with fine-grained locking, $T1$'s and $T2$'s critical sections are executed *sequentially* (Figure 10b): threads $T1$ and $T2$ compete to acquire the *same* lock, i.e., the lock associated with the vertex u, in order to execute their critical sections. Thus, only *one* of threads

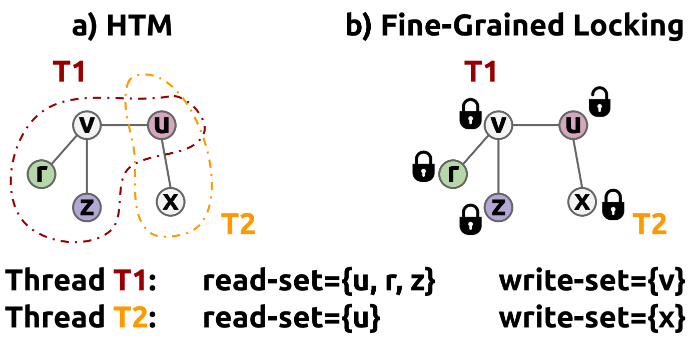
a) HTM
b) Fine-Grained Locking
T1
T2
v
u
r
z
x
T1
T2
v
u
r
z
x
Thread T1: read-set={u, r, z} write-set={v}
Thread T2: read-set={u} write-set={x}

Fig. 10 An example execution scenario in which threads $T1$ and $T2$ attempt to concurrently find colors for the vertices v and x, respectively, using a) HTM and b) fine-grained locking for synchronization. The white circles represent uncolored vertices, and the colorful circles represent vertices that have already obtained a color.

$T1$ and $T2$ will acquire the lock, and will proceed. Given that $T1$'s and $T2$'s critical sections have *overlapped* data access sets, i.e., both of them include the color of the vertex u in their read-sets, they will be executed sequentially when using the fine-grained locking scheme for synchronization. As a result, we conclude that in graph coloring HTM can provide higher levels of parallelism compared to fine-grained locking.

To this end, ColorTM employs HTM to deal with race conditions that arise when parallel threads concurrently process adjacent vertices. HTM can detect and resolve coloring inconsistencies among parallel threads as follows:

– **HTM can detect coloring conflicts:** HTM detects coloring conflicts that arise due to crossing edges. For a vertex v to be colored, we enclose within the transaction (i) the memory location that stores the color of the current vertex v (the transaction's write-set), and (ii) the memory locations that store the colors of the critical adjacent vertices of the vertex v (the transaction's read-set). When parallel threads attempt to concurrently update-write the colors of adjacent vertices using different transactions, the HTM mechanism detects read-write conflicts across the running transactions: a running transaction attempts to write the read-set of another running transaction. Figure 11 provides an example scenario on how HTM detects coloring inconsistencies among two parallel threads. When the thread $T1$ attempts to color the vertex v using HTM, the corresponding running transaction includes the memory location of the color of the vertex v in its write-set, and the memory locations of the colors of the v's adjacent vertices, i.e., u, r and z vertices, in its read-set. Similarly, when the thread $T2$ attempts to color the vertex u using HTM, the corresponding running transaction includes the memory location of the color of the vertex u in its write-set, and the memory locations of the colors of the u's adjacent vertices, i.e., v and x vertices, in its read-set. When $T1's$ and $T2's$ transactions are executed concurrently, HTM detects a read-write conflict either on the color of the vertex v or the color of the vertex u: either $T1's$ transaction attempts to write the read-set of $T2's$ transaction or $T2's$ transaction attempts to write the read-set of $T1's$ transaction. Therefore, one of the two running transactions will be aborted by the HTM mechanism, and the other one will be committed.

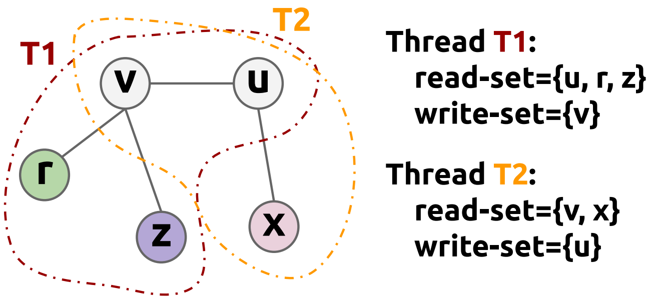

Fig. 11 An example execution scenario in which threads $T1$ and $T2$ attempt to concurrently update the colors of the vertices v and u respectively, using two different transactions, and the HTM mechanism detects read-write conflicts to their data sets. The white circles represent uncolored vertices, and the colorful circles represent vertices that have already obtained a color.

– **HTM can resolve coloring conflicts:** In case of n conflicting running transactions (read-write conflicts explained in Figure 11), the HTM mechanism aborts $n-1$ running transactions and commits only one of them. In prior graph coloring schemes such as SeqSolve (line 11 of Figure 2), IterSolve (line 13 of Figure 3), VFF (line 23 of Figure 6) and Recoloring (line 22 of Figure 7), the programmer explicitly defines a coloring conflict resolution policy among conflicted vertices to guarantee forward progress, i.e., the programmer explicitly defines which of the conflicted vertices will be recolored next. In contrast, in ColorTM when coloring conflicts arise among multiple running transactions, the programmer does *not* need to explicitly define a conflict resolution policy: the HTM mechanism itself commits one of the multiple conflicted transactions and aborts the remaining running transactions. Thus, the conflict resolution policy implemented in the underlying hardware mechanism of HTM determines which vertices will continue to be processed for coloring.

However, currently available HTM systems [37–40] are best-effort HTM implementations that do *not* guarantee forward progress: a transaction may always fail to commit and thus, a non-transactional execution path (*fallback path*) needs to be implemented. The most common fallback path is to implement a coarse-grained locking solution: each transaction can be retried up to a predefined number of times (pre-determined threshold), and if it exceeds this threshold, it fall backs to the acquisition of global lock, which allows only one single thread to execute its critical section. To implement this, the global lock is added to the transactions' read sets: inside the transaction the thread always reads the value of the global lock variable. During the multithreaded execution, when the transaction of a parallel thread exceeds the predefined threshold of retries, the parallel thread acquires the global lock by writing to the value of the global lock variable, and then the concurrent running transactions of the remaining threads are aborted (read-write conflict) and wait until the global lock is released.

4.2 Critical Adjacent Vertices

ColorTM implements a speculative computation approach to achieve high performance. Specifically, for each vertex v, all necessary computations to find a permissible color k are performed outside the critical section (line 5 in Figure 9) such that avoid unnecessary computations inside the critical sections. Within the critical section, ColorTM *only* validates the speculative color k (line 9 in Figure 9) by comparing it with the colors of the adjacent vertices of vertex v. However, the speculative color k for a vertex v does *not* need to be validated with the colors of *all* the adjacent vertices of vertex v: we observe that some adjacent vertices can be omitted from the validation process of the critical section, because they do not cause *any* coloring inconsistency with the vertex v. Specifically, we can omit from the validation step performed within the critical section the following adjacent vertices of vertex v:

1. **The adjacent vertices that are assigned to be processed by the same thread with the vertex v.** Given that the vertices of the graph are distributed across multiple threads, coloring conflicts cannot arise between adjacent vertices that are assigned to the *same* parallel thread. Therefore, we omit from the validation step of the critical section the adjacent vertices assigned to the same thread as the current vertex v.
2. **The adjacent vertices that have already obtained a color.** As already explained, ColorTM does *not* perform re-colorings to the vertices of the graph: once a vertex is assigned a permissible color within the critical section (using synchronization), it will *not* be re-colored again during runtime. Multiple parallel threads repeatedly iterate over a vertex until a valid coloring is found, which is assigned to it using data synchronization, and then proceed to the remaining vertices. Therefore, in ColorTM coloring conflicts do *not* arise between adjacent vertices that have already obtained a color: the colors assigned to adjacent vertices are taken into consideration in the computations performed outside the critical section (line 5 in Figure 9) to find a speculative color for the current vertex, and will not be modified when the critical section is executed (lines 7-15 in Figure 9), since ColorTM does *not* perform re-colorings. Therefore, adjacent vertices of a vertex v that have already obtained a color when the speculative coloring computation step (line 5 in Figure 9) is executed, do not cause any coloring inconsistency when critical section is executed (lines 7-15 in Figure 9). Hence, we can safely omit from the validation step of the critical section the adjacent vertices that have already been assigned a color.

Figure 12 presents an example execution scenario of a graph partitioned across two parallel threads $T1$ and $T2$. In Figure 12, the white vertices represent uncolored vertices and the colorful vertices represent vertices that have already obtained a color during runtime. In this scenario, threads $T1$ and $T2$ attempt to color the vertices v and u, respectively. According to our described

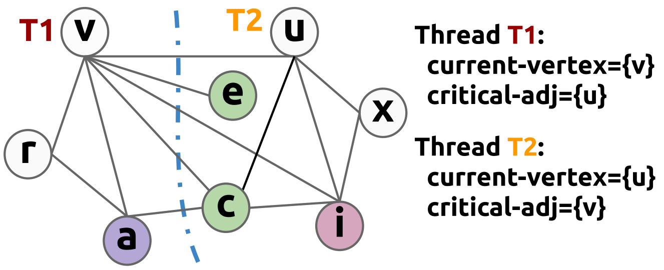

Fig. 12 An example execution scenario in which the graph is partitioned across two parallel threads. The white circles represent uncolored vertices, and the colorful circles represent vertices that have already obtained a color. When the threads $T1$ and $T2$ attempt to color the vertices v and u, respectively, the critical adjacent vertices that need to be validated within the critical section (HTM) are *only* the vertices u and v, respectively.

optimizations, the adjacent vertices that need to be validated inside the critical sections (via HTM) of the vertices v and u are *only* the vertices u and v, respectively.

Overall, for the current vertex v to be colored, the necessary adjacent vertices that need to be validated inside the critical section, referred to as *critical* adjacent vertices, are the *uncolored* adjacent vertices assigned to different parallel threads compared to the thread to which the vertex v is assigned to. By accessing inside the critical section *only* a few data needed to ensure correctness, ColorTM provides short critical sections and small transaction footprints, and achieves high levels of parallelism and low synchronization costs, i.e., low abort ratio in hardware transactions of HTM (See Section 9.1). Note that having large transactions footprints in HTM transactions can cause three important problems: (i) if the transaction read- and write-sets are large, the available hardware buffers of HTM may be oversubscribed (hardware overflow), and in that case the HTM mechanism will abort the running transactions due to capacity aborts, (ii) if the duration of a running transaction is long (e.g., due to expensive data accesses), the running transactions may be aborted due to a time interrupt (when the duration of a transaction exceeds the time scheduling quantum, the OS scheduler schedules out the software thread from the hardware thread and the transaction is aborted), and (iii) the longer the transactions last and the larger their data sets are, the greater the probability that running transactions are aborted due to (read-write) data conflicts among them.

4.3 Implementation Details

Figure 13 presents ColorTM in detail. ColorTM distributes the vertices of the graph across multiple threads, which color the vertices of the graph through one single parallel step (lines 4-29): multiple parallel threads repeatedly iterate over each vertex of the graph until a valid coloring on each vertex is performed.

For each vertex v, there are two sub-steps. In the first sub-step (lines 6-13), the parallel thread keeps track (i) the forbidden set of colors assigned to the adjacent vertices of the vertex v (line 10), and (ii) the critical adjacent vertices of the vertex v (lines 11-12), which are the uncolored adjacent vertices assigned

```
1  Input: Graph G=(V,E)
2  Let $N(v)$ be the adjacent vertices of the vertex $v$
3  Let tid be the unique id of each parallel thread
4  for each $v \in V$ do in parallel
5    RETRY:
6    // Speculative Computation
7    R = $\emptyset$ // Track Forbidden Colors
8    C = $\emptyset$ // Track Critical Adjacent Vertices
9    for each $u \in N(v)$ do
10     R = R $\cup$ $u$.color
11     if ((hasColor($u$) == false) && (get_threadID($u$) != tid))
12       C = C $\cup$ $u$ // Critical Adjacent Vertices Are the Uncolored Vertices Assigned to Another Thread
13   $k$ = compute_speculative_color(R)  // Compute a Speculative Color $k$ for the Vertex $v$
14   // Validate Coloring
15   if (C == $\emptyset$) // Skip the Validation Step, If There Are No Critical Adjacent Vertices
16     $v$.color = $k$
17   else
18     begin_transaction
19     bool valid = true
20     for each $u \in C$ do // Validate the Colors of the Critical Adjacent Vertices Over the Speculative Color
21       if ($u$.color == $k$)
22         valid = false
23         break
24     if (valid == true) // If the Validation Succeeded, Assign the Speculative Color to the Vertex $v$
25       $v$.color = $k$
26       end_transaction
27     else // If the Validation Failed, Immediately Retry to Find a New Color for the Vertex $v$
28       end_transaction
29       goto RETRY // Eager Resolution
```

Fig. 13 The ColorTM algorithm.

to different parallel threads (line 11), and then computes a speculative color k that is permissible for the vertex v using the `compute_speculative_color()` function (line 13). In the second sub-step (lines 14-29), the parallel thread validates and assigns (if allowed) the speculative color k to the vertex v using data synchronization via HTM (lines 18-29). Specifically, the colors of the critical adjacent vertices are compared to the speculative color k within a hardware transaction (lines 20-23) to ensure that the color k is still permissible to be assigned to the vertex v. If the validation succeeds (line 24), the color k is assigned to the vertex v within the same transaction (line 25) to ensure correctness: recall that the reads on the colors of the critical adjacent vertices need to be executed *atomically* with the write-update on the color of the vertex v. Instead, if the validation step fails due to a coloring inconsistency appeared during runtime (line 27), the parallel thread *repeatedly* and *eagerly* retries to find a new permissible color for the current vertex v (line 29). Note that if there are *no* critical adjacent vertices to be validated (line 15), the speculative color k is directly assigned to the vertex v *without* using synchronization (line 16).

Note that in the second sub-step (lines 14-29), ColorTM does *not* check if the colors of the critical adjacent vertices have not been modified since the first sub-step (lines 6-13). Instead, the validation of the second sub-step *only* checks that the colors of the critical adjacent vertices are different from the speculative color k computed in the first sub-step (line 13). In the meantime, different parallel threads may have just assigned new colors to critical adjacent vertices, which however are different from the color k, and thus causing *no* coloring inconsistencies. In that scenario, the validation of the second substep succeeds. This way, ColorTM provides high levels of parallelism: multiple parallel threads that have just assigned *different* colors than the color k to critical adjacent vertices of the vertex v will *not* cause any validation failure in the critical section of the vertex v, and the corresponding running transaction will be safely committed.

4.4 Progress and Correctness

We clarify in detail how ColorTM resolves the race conditions that may arise during runtime. There are two race conditions that may cause coloring inconsistencies in multithreaded executions. First, while a parallel thread computes a speculative color k for the vertex v (lines 9-13 of Figure 13), different parallel threads may have just assigned the color k to one or more adjacent vertices of the vertex v. In that scenario, the validation step of lines 20-23 of Figure 13 fails (line 22, 27), since the speculative color k has been assigned to at least one critical adjacent vertex (line 21). Then, the corresponding parallel thread will retry to find a new permissible color for the vertex v (line 29). Second, a race condition arises when n parallel threads (assuming $n > 1$) attempt to write-update the same color k to n adjacent vertices (fully connected adjacent vertices) within n different running transactions. In that scenario, the HTM mechanism detects read-write data conflicts on running transactions, because one (or more) running transaction attempts to write to the read-sets of another running transactions. Recall that the colors of the critical adjacent vertices are included in the read-set of each running transaction (lines 21 of Figure 13). Then, the HTM mechanism aborts $n-1$ running transactions, and commits only *one* of them. When the aborted $n-1$ transactions retry (each transaction can retry up to a predefined number of times), the validation step of lines 20-23 fails (lines 27 of Figure 13), since at that time the $n-1$ parallel threads observe that there is one critical adjacent vertex that has just been assigned to the color k (the committed transaction). Afterwards, since the validation failed, the $n-1$ parallel threads will retry to find new permissible colors for their current vertices (lines 27-29 of Figure 13).

Finally, we clarify that ColorTM provides forward progress and eventually terminates: each parallel thread retries to find a new permissible color for a current vertex v (line 29 of Figure 13) up to a limited number of retries. Specifically, a parallel thread retries to find a new color for a vertex v, when the validation step of lines 20-23 of Figure 13 fails. However, for each vertex v the validation step can fail up to a bounded number of times: the validation step fails when one (or more) critical adjacent vertex has been assigned to the same color k' with the speculative color k computed for the vertex v. Therefore, in the worst case, the validation step might fail up to $deg(v)$ times, where $deg(v)$ is the adjacency degree of the vertex v. When all v's adjacent vertices have obtained a color, there are *no* critical adjacent vertices to be validated (line 15 of Figure 13), and thus, the speculative color k is directly assigned to the vertex v (line 16 of Figure 13), and the validation step is omitted. As a result, each parallel thread retries to find a color for each vertex v of the graph at most $deg(v)$ times. However, in our evaluation, we find that the validation step fails only for *a few* times: across all our evaluated large real-world graphs (Table 1) and using a large number of parallel threads (up to 56 threads) the validation step failures are less than 0.01%. Overall, we conclude that ColorTM *correctly* handles all the race conditions that may arise in multithreaded executions of the graph coloring kernel, and *effectively* terminates with a valid coloring.

4.5 The BalColorTM Algorithm

Figure 14 presents the *balanced* counterpart of ColorTM, named as BalColorTM. Similarly to CLU and VFF, in BalColorTM (i) only the vertices of the over-full color classes are considered for re-coloring, i.e., to be moved from over-full to under-full color classes in order to achieve high vertex-balance across color classes, and (ii) graph coloring balance is achieved *without increasing* the number of color classes produced by the initial graph coloring (e.g., using ColorTM).

Similarly to ColorTM, BalColorTM (Figure 14) has one single parallel step (lines 7-42): multiple parallel threads repeatedly iterate over each vertex of the over-full color classes until either a valid re-coloring to an under-full class is performed, or there is no permissible re-coloring for this vertex to an under-full color class (line 42). For each vertex of an over-full color class c, there are two sub-steps. In the first sub-step (lines 8-20), the parallel thread keeps track the forbidden set of colors assigned to the adjacent vertices of the vertex v (line 16), and the set of the critical adjacent vertices (lines 17-18) of the vertex v. In BalColorTM, note that the *critical* adjacent vertices of a vertex v (line 17) are the adjacent vertices that (i) belong to *over-full* color classes (recall that the vertices assigned under-full color classes are *not* considered to be re-colored/moved, and thus they do not cause any coloring inconsistency during runtime), and (ii) are assigned to different threads compared to the parallel thread in which the vertex v is assigned to. Then, the parallel thread speculatively computes a color k of an under-full color class that is permissible to be assigned to the vertex v (lines 19-20). If a permissible color k exists (without increasing the number of color classes produced by the initial graph coloring), the parallel thread attempts to assign the speculative color k to the vertex v in the second sub-step (lines 21-42). If there is *no* permissible color k of an under-full color class (line 41), the parallel threads continue to process the next vertices (line 42). In the second sub-step, if there are critical adjacent vertices that need to be validated, the parallel thread validates the speculative color k over the colors of the critical adjacent vertices within an HTM transaction (lines 27-39). If the validation succeeds (line 33), the parallel thread moves the vertex v from the color class c to the color class k by re-coloring it (line 34), and atomically updates the sizes of the color classes c and k (lines 36-37) accordingly. If the validation step fails due to a coloring inconsistency appeared during runtime (line 38), the parallel thread *eagerly* retries to find a new permissible color of an under-full color class for the vertex v (line 40).

```
 1 Input: Graph G=(V,E)
 2 Let $N(v)$ be the adjacent vertices of the vertex $v$
 3 Obtain an initial coloring on G
 4 Let $C$ be the number of colors produced
 5 Let $b = V/C$ be the perfect balance
 6 Let $Q$ be the set of vertices of the over-full color classes
 7 for each $v \in Q$ do in parallel
 8   Let $c$ be the current color of the vertex $v$
 9   if (the size of the color class $c$ <= $b$)
10       continue// Color Class is Balanced
11   RETRY:
12   // Speculative Computation
13   R = $\emptyset$ // Track Forbidden Colors
14   C = $\emptyset$ // Track Critical Adjacent Vertices
15   for each $u \in N(v)$ do
16     R = R $\cup$ $u$.color
17     if ((isOverFull($u$.color) == true) && (get_threadID($u$) != tid))
18       C = C $\cup$ $u$ // Critical Adjacent Vertices Are the Vertices of Over-Full Color
        Classes That Are Assigned to Another Thread
19   $k$ = compute_speculative_color(R)
20   Let $k$ be the index of the minimum under-full color class that is
        permissible to the vertex $v$
21   if ($k$ exists) // Validate Coloring
22     if (C == $\emptyset$) // Skip the Validation Step, If There Are No Critical Adjacent
        Vertices
23       $v$.color = $k$
24       Atomically decrease the size of the color class $c$
25       Atomically increase the size of the color class $k$
26     else
27       begin_transaction
28       bool valid = true
29       for each $u \in C$ do // Validate the Colors of the Critical Adjacent Vertices Over
        the Speculative Color
30         if ($u$.color == $k$)
31           valid = false
32           break
33       if (valid == true) // If the Validation Succeeded, Set the Speculative Color to
        the Vertex $v$
34         $v$.color = $k$
35         end_transaction
36         Atomically decrease the size of the color class $c$
37         Atomically increase the size of the color class $k$
38       else // If the Validation Failed, Immediately Retry to Find a New Color for the
        Vertex $v$
39         end_transaction
40         goto RETRY // Eager Resolution
41   else
42     continue
```

Fig. 14 The BalColorTM algorithm.

Finally, note that BalColorTM iterates over the vertices of each over-full color class until that particular over-full class becomes balanced at a certain point in the execution (lines 9-10), i.e., until the size of the particular color class becomes smaller or equal to $b = V/C$. Then, the vertices belonging to that color class are no longer considered for re-coloring (line 10). Overall, BalColorTM terminates when either vertex-balance across color classes is achieved or vertex-balance across color classes is no longer available, i.e., there are no more permissible re-colorings for any vertex belonging to an over-full color class.

Similarly to ColorTM, BalColorTM *completely* avoids barrier synchronization, since it includes only one single parallel step. This way BalColorTM significantly minimizes synchronization costs compared to prior balanced graph coloring algorithms (e.g., CLU, VFF, Recoloring) that employ barrier synchronization. Moreover, it also integrates an *eager* approach to detect and resolve coloring conflicts appearing during runtime among parallel threads, that concurrently move vertices from over-full to under-full color classes. With the *eager* coloring policy, BalColorTM provides high performance by minimizing access latency costs to application data compared to that of prior balanced graph coloring algorithms (e.g., CLU, VFF, Recoloring), which integrate a *lazy* approach to detect and resolve coloring conflicts. Finally, BalColorTM effectively implements short critical sections (short running transactions with small transaction footprints) by (i) speculatively performing the computations to find permissible colors for the vertices of the over-full color classes *outside* the critical section (lines 9-13), and (ii) accessing inside the critical sections *only* the necessary data to ensure correctness, i.e., for each vertex v BalColorTM *only* accesses the colors of a small subset of v's adjacent vertices (critical adjacent vertices). Via short running transactions, BalColorTM achieves low synchronization costs and provides high amount of parallelism.

5 Evaluation

5.1 Evaluation Methodology

We conduct our evaluation using a 2-socket Intel Haswell server with an Intel Xeon E5-2697 v3 processor with 28 physical cores and 56 hardware threads. The processor runs at 2.6 GHz and each physical core has its own L1 and L2 caches of sizes 32 KB and 256 KB, respectively. Each socket includes a shared 35 MB L3 cache. We statically pin each software thread to a hardware thread, and enable hyperthreading only on 56-thread executions, unless otherwise stated. In our evaluation (Section 5), the numbers reported are averaged across 5 runs of each experiment.

Table 1 shows the characteristics of the large real-world graphs used in our evaluation. We select 18 representative graphs from the Suite Matrix Collection that vary in vertex and graph degrees, and are used in different application domains. For each graph, Table 1 presents the number of vertices (#vertices), the number of edges (#edges), the maximum (deg_{max}) degree, the average (deg_{avg}) degree and the standard deviation of the vertices' degrees (deg_{std}), and the last column of this table shows the ratio of the standard deviation of the vertices' degrees to the average degree ($\frac{deg_{std}}{deg_{avg}}$).

This section evaluates the proposed ColorTM and BalColorTM algorithms. First, we compare the coloring quality and the performance over prior state-of-the-art graph coloring algorithms (Section 5.2). Second, we compare the color balancing quality and the performance of BalColorTM over prior state-of-the-art balanced graph coloring algorithms (Section 5.3). Finally, we evaluate the performance of Community Detection [34] by parallelizing it using ColorTM and BalColorTM (Section 5.4) via chromatic scheduling.

5.2 Analysis of Parallel Graph Coloring Algorithms

We compare the following parallel graph coloring implementations:

Graph Name	**#Vertices**	**#Edges**	$\mathbf{deg_{max}}$	$\mathbf{deg_{avg}}$	$\mathbf{deg_{std}}$	$\mathbf{\frac{deg_{std}}{deg_{avg}}}$
Queen_4147 (**qun**)	4147110	329499284	81	79.45	6.34	0.080
Geo_1438 (**geo**)	1437960	63156690	57	43.92	4.39	0.100
Flan_1565 (**fln**)	1564794	117406044	81	75.03	11.43	0.152
Bump_2911 (**bum**)	2911419	127729899	195	43.87	6.96	0.159
Serena (**ser**)	1391349	64531701	249	46.38	9.24	0.199
delaunay_n24 (**del**)	16777216	100663202	26	5.99	1.34	0.222
rgg_n_2_23_s0 (**rgg**)	8388608	127002786	40	15.14	3.89	0.257
kmer_A2a (**kmr**)	170728175	360585172	40	2.11	0.57	0.267
cage15 (**cag**)	5154859	99199551	47	19.24	5.73	0.298
road_usa (**usa**)	23947347	57708624	9	2.41	0.93	0.386
dielFilterV3real (**dlf**)	1102824	89306020	270	80.98	36.56	0.451
audikw_1 (**aud**)	943695	77651847	345	82.29	42.44	0.516
vas_stokes_2M (**vas**)	2146677	65129037	637	30.34	37.18	1.226
stokes (**stk**)	11449533	349321980	720	30.51	41.44	1.358
uk-2002 (**uk**)	18520486	298113762	2450	16.10	27.53	1.710
soc-LiveJournal1 (**soc**)	4847571	68993773	20293	14.23	36.08	2.535
arabic-2005 (**arb**)	22744080	639999458	9905	28.14	78.84	2.802
FullChip (**fch**)	2987012	26621990	2312481	8.91	1806.80	202.725

Table 1 Large Real-World Graph Dataset.

- The sequential Greedy algorithm presented in Figure 1.
- The SeqSolve algorithm presented in Figure 2.
- The IterSolve algorithm presented in Figure 3.
- The IterSolveR algorithm presented in Figure 4.
- A variant of our proposed algorithm (Figure 13) that uses fine-grained locking instead of HTM, henceforth referred to as ColorLock. Specifically, each vertex of the graph is associated with a software-based lock. In the beginning of the critical section (line 18 in Figure 13), parallel threads acquire the corresponding locks of both the current vertex v and the *critical* adjacent vertices of the vertex v. Then, when the critical section ends (lines 26 and 28 in Figure 13), parallel threads release the acquired locks. To avoid deadlocks, we impose a global order when acquiring/releasing locks based on the vertices' id: parallel threads acquire/release locks of multiple vertices starting from the lock associated with the vertex with the smallest vertex id, iterating via an increasing order of the vertices' ids, and finishing to the lock associated with the vertex with the highest vertex id.
- Our proposed ColorTM algorithm (Figure 13) that leverages HTM. Each transaction can retry up to 50 times, before resorting to a non-transactional fallback path. The non-transactional path is a coarse-grained locking solution for the critical section (lines 18-28 in Figure 13).

For a fair comparison, in all graph coloring schemes we color the vertices in the order they appear in the input graph representation (first-fit ordering heuristic [21]).

5.2.1 Analysis of the Coloring Quality

Table 2 compares the coloring quality of all parallel graph coloring implementations in single-threaded and multithreaded executions.

Coloring Scheme	**1 thread**	**14 threads**	**28 threads**	**56 threads**
Greedy	42.58	-	-	-
SeqSolve	42.58	42.34	42.33	42.18
IterSolve	42.58	44.05	43.94	44.04
IterSolveR	42.58	43.61	43.88	44.58
ColorLock	42.58	45.75	45.67	46.14
ColorTM	42.58	46.20	45.77	46.28

Table 2 The geometric mean on the number of colors produced across all large real-world graphs (lower is better) for each parallel graph coloring implementation using one core (1 thread), all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads).

We make two key observations. First, there is low variability on the number of colors used across the different graph coloring schemes. The parallel graph coloring schemes provide similar graph coloring quality, because the number of colors produced is primarily determined by the order in which the vertices are colored [20, 21]. In this work, we use the *first-fit* ordering heuristic in all schemes, i.e., coloring the vertices in the order they appear in the input graph representation, and we leave the experimentation of other ordering heuristics for future work. Second, we find that in most schemes the coloring quality becomes slightly worse as the number of threads increases. As the number of threads increases, the number of coloring conflicts that arise during runtime typically increases, and thus parallel threads might resolve coloring inconsistencies by introducing a few additional color classes. The SeqSolve scheme does not typically increase the number of colors used in multithreaded executions, because the coloring inconsistencies are resolved using one single thread. Overall, we conclude that since all graph coloring schemes employ the same ordering heuristic, they provide similar coloring quality.

5.2.2 Performance Comparison

Figure 15 evaluates the scalability achieved by all parallel graph coloring implementations in our large real-world graphs, when increasing the number of threads from 1 to 56, i.e., the maximum available hardware thread capacity of our machine.

We draw three findings. First, ColorTM and ColorLock achieve the lowest execution time across all schemes in single-threaded executions. Using one single thread, ColorTM and ColorLock on average outperform SeqSolve by 1.55× and 1.42×, respectively, and they on average outperform IterSolve by 1.17× and 1.06×, respectively. With only one thread, ColorTM and ColorLock have identical executions to the sequential Greedy algorithm (Figure 1): thanks to the optimizations proposed in Section 4.2, the list of critical adjacent vertices that need to be validated inside the critical section is empty, and thus ColorTM and ColorLock *completely* eliminate using synchronization (either HTM of fine-grained locking). Second, we find that IterSolveR exhibits the lowest scalability across all schemes. IterSolveR merges two parallel for-loops into a

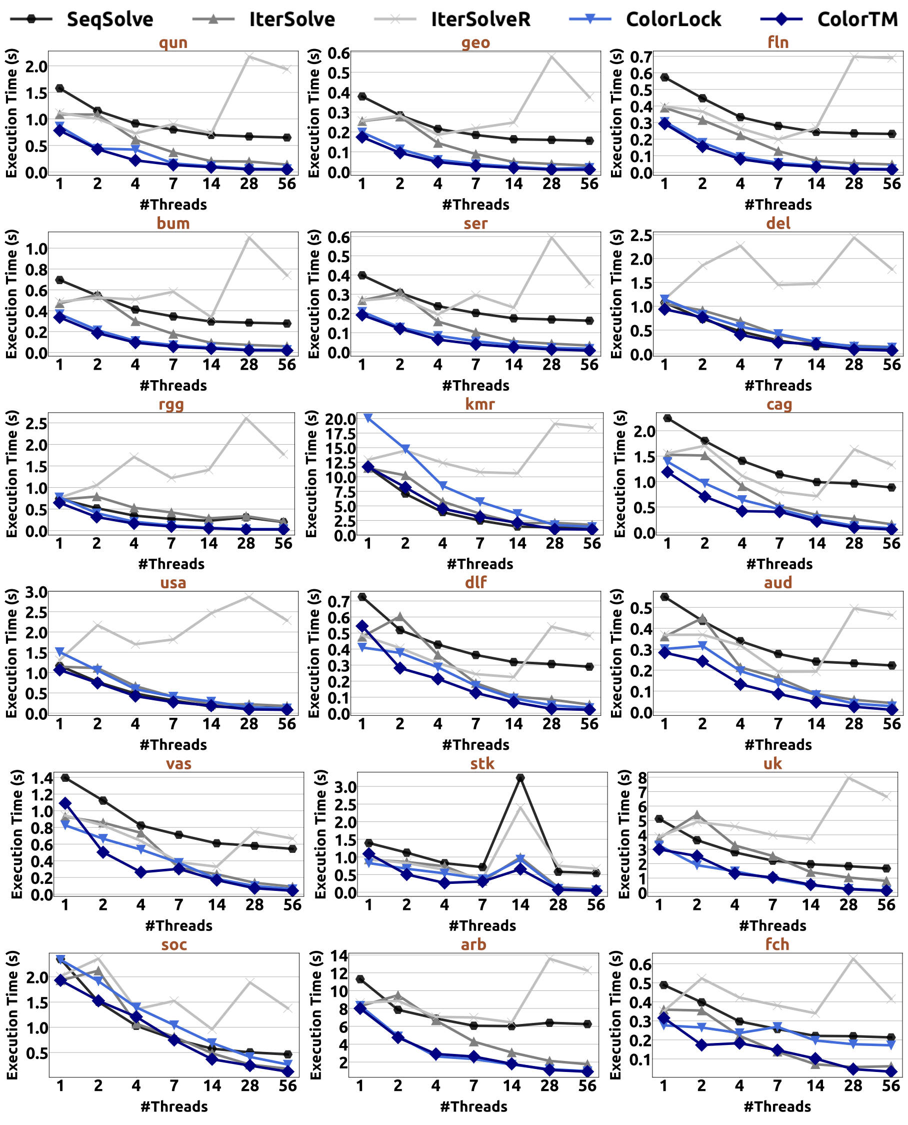

Fig. 15 Scalability achieved by all parallel graph coloring implementations in large real-world graphs.

single parallel for-loop in order to eliminate one of the two barriers used in IterSolve. Even though IterSolveR reduces the barrier synchronization costs, it increases the load imbalance among parallel threads, thus causing significant performance overheads. Third, we observe that the scalability of SeqSolve, IterSolve, and IterSolveR is highly affected by the NUMA effect, i.e., the non-uniform memory access latencies to the application data. For example, when increasing the number of threads from 7 to 14 (only one NUMA socket is used) the performance of SeqSolve, IterSolve, IterSolveR, ColorLock and ColorTM improves by 1.24×, 1.75×, 1.06×, 1.62× and 1.65×, respectively, averaged across all large graphs. However, when increasing the number of threads from 14 to 28, i.e., using both NUMA sockets of our machine, the performance of SeqSolve and IterSolve *only* improves by 1.03× and 1.26×, respectively, while the performance of and IterSolveR decreases by 2.13×, averaged across all large graphs. In contrast, when increasing the number of threads from 14 to 28, the performance of ColorLock and ColorTM significantly improves by 1.77× and 1.97×, respectively, averaged across all graphs. This is because our proposed algorithmic design implemented in ColorLock and ColorTM leverages better the deep memory hierarchy of commodity multicore platforms thanks to its *eager* conflict detection and resolution policy, thus achieving lower data access costs. Overall, we conclude that our proposed algorithmic design achieves the best scalability in modern multicore platforms.

Figure 16 compares the speedup achieved by all schemes over the sequential Greedy scheme, when varying the number of hardware threads used in all large real-world graphs.

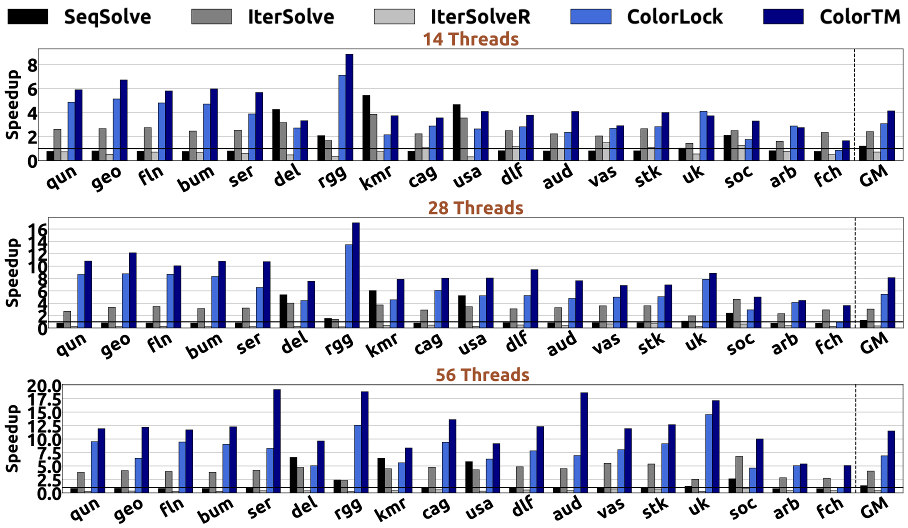

Fig. 16 Speedup achieved by all parallel graph coloring implementations over the sequential Greedy scheme in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads).

We make two key observations. First, all parallel graph coloring schemes achieve lower speedup in very irregular graphs including the `soc`, `arb` and `fch` graphs, compared to all the remaining real-world graphs. In very irregular graphs, the number of edges per vertex significantly vary across vertices [41–43]: typically only a few vertices have a much larger number of edges over the vast majority of the remaining vertices of the graph. Therefore, in irregular graphs parallel threads typically cause more coloring inconsistencies than regular graphs, which are resolved during runtime, increasing the execution time. Second, we find that ColorTM achieves significant performance improvements over all the prior state-of-the-art graph coloring schemes. ColorTM outperforms SeqSolve, IterSolve, and IterSolveR by 3.43×, 1.71× and 5.83× respectively, when using 14 threads, and by 8.46×, 2.84× and 27.66× respectively, when using the maximum hardware thread capacity of our machine (56 threads). This is because SeqSolve, IterSolve, and IterSolveR traverse *all* the vertices of the graph at least twice, and employ a *lazy* conflict resolution policy, thus incurring high data access costs. Instead, ColorTM traverses more than once *only* the conflicted vertices, and resolves coloring inconsistencies with an *eager* approach, thus better leveraging the deep memory hierarchy of multicore platforms and reducing data access costs. In addition, ColorTM outperforms ColorLock by 1.34× and 1.67× when using 14 and 56 threads, respectively. As explained, HTM is a speculative hardware-based synchronization mechanism, and thus ColorTM provides high performance improvements over ColorLock thanks to significantly minimizing data access and synchronization costs. Note that in the fine-grained locking approach of ColorLock, for each adjacent vertex accessed inside the critical section, the parallel thread needs to acquire and release the corresponding software-based lock, thus performing additional memory accesses in the memory hierarchy for accessing the lock variable. Overall, we conclude that ColorTM significantly outperforms all prior state-of-the-art parallel graph coloring algorithms across a wide variety of large real-world graphs.

To confirm the performance benefits of ColorTM across multiple computing platforms, we evaluate all schemes on a 2-socket Intel Broadwell server with an Intel Xeon E5-2699 v4 processor at 2.2 GHz having 44 physical cores and 88 hardware threads. Figure 17 compares the speedup achieved by all schemes over the sequential Greedy scheme in all large real-world graphs using 88 threads, i.e., the maximum hardware thread capacity of the Intel Broadwell server. We find that ColorTM provides significant performance benefits over prior state-of-the-art graph coloring algorithms, achieving 11.98×, 4.33× and 22.06× better performance over SeqSolve, IterSolve, and IterSolveR, respectively.

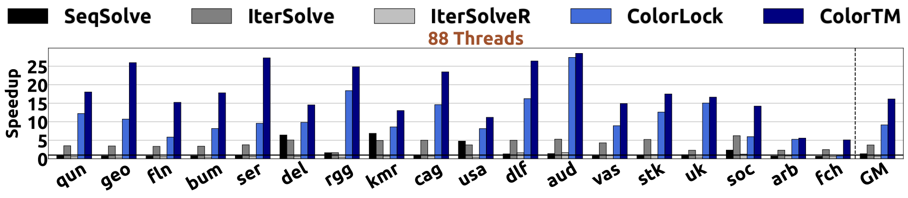

Fig. 17 Speedup achieved by all parallel graph coloring implementations over the sequential Greedy scheme in large real-world graphs using the maximum hardware thread capacity of an Intel Broadwell server with hyperthreading enabled (88 threads).

5.3 Analysis of Balanced Graph Coloring Algorithms

We compare the following balanced graph coloring implementations:

- The CLU algorithm presented in Figure 5.
- The VFF algorithm presented in Figure 6.
- The Recoloring algorithm presented in Figure 7.

- Our proposed BalColorTM algorithm (Figure 14) that leverages HTM. Each transaction is retried up to 50 times, before resorting to a non-transactional fallback path. The non-transactional path is a coarse-grained lock scheme for the critical section (lines 27-39 in Figure 14).

For a fair comparison, in all graph coloring schemes we color the vertices in the order they appear in the color classes produced by the initial coloring.

5.3.1 Analysis of Color Balancing Quality

Table 3 compares the quality of balance in the color class sizes produced by the balanced-oblivious ColorTM and all our evaluated balanced graph coloring implementations. Similarly to [31], we evaluate the color balancing quality using the relative standard deviation of the color class sizes expressed in %, which is defined as the ratio of the standard deviation of the color class sizes to the average color class size. The closer the value of this metric is to 0.00, the better is the color balance. For the ColorTM and Recoloring schemes, we also include in parentheses the number of color classes produced. As already explained in Section 2.3, the CLU, VFF, and BalColorTM schemes produce the same number of color classes with the initial coloring. In this experiment, we evaluate all algorithms using the maximum hardware thread capacity of our machine, i.e., 56 threads, in order to evaluate the color balancing quality of all schemes using the maximum available parallelism provided by the underlying hardware platform.

Input Graph	**Initial Coloring ColorTM**		**Balanced Graph Coloring Schemes**				
			CLU	**VFF**	**Recoloring**		**BalColorTM**
qun	63.62	(48)	0.212	1.669	14.739	(48)	0.009
geo	70.28	(36)	0.321	0.635	17.664	(34)	0.020
fln	65.42	(45)	0.576	0.611	20.384	(51)	0.044
bum	64.32	(36)	0.179	0.647	17.950	(33)	0.009
ser	73.64	(39)	0.405	0.751	16.651	(38)	0.024
del	100.06	(9)	0.002	0.013	35.136	(10)	0.001
rgg	115.30	(22)	0.018	3.783	21.799	(23)	0.003
kmr	189.79	(11)	0.0003	0.0002	31.492	(12)	0.0004
cag	122.89	(19)	0.014	0.649	34.197	(20)	0.005
usa	105.09	(5)	0.001	0.024	0.0005	(5)	0.0005
dlf	57.95	(54)	2.58	2.53	22.551	(57)	3.01
aud	84.02	(60)	5.243	2.780	19.498	(54)	3.575
vas	144.18	(38)	0.084	18.527	25.373	(34)	0.016
stk	141.41	(35)	0.016	17.684	25.375	(34)	0.003
uk	1882.66	(944)	0.437	0.237	65.994	(1355)	1.732
soc	945.35	(324)	1.136	1.466	58.190	(459)	1.886
arb	3351.79	(3248)	0.681	1.499	68.521	(4772)	3.410
fch	125.70	(9)	0.012	0.271	33.854	(10)	0.451

Table 3 Color balancing quality achieved by ColorTM and all balanced graph coloring implementations in the large real-world graphs. We present the relative standard deviation (in %) on the sizes of the color classes obtained by each scheme (lower is better). In ColorTM and Recoloring, we provide inside the parentheses the number of color classes produced. The CLU, VFF and BalColorTM produce the same number of color classes with the initial coloring scheme.

We draw three findings from Table 3. First, we observe that the balanced-oblivious ColorTM scheme incurs very high disparity in the sizes of the color classes produced. Specifically, the color balancing quality of ColorTM is 1887.01×, 287.70×, 10.32×, and 4266.03× worse than that of CLU, VFF, Recoloring and BalColorTM, respectively. Second, even though Recoloring is effective over ColorTM by providing better color balancing quality, its color balancing quality is the worst compared to all the remaining balanced graph coloring schemes. In addition, in highly irregular graphs (graphs with high maximum degree and high standard deviation in the vertices' degrees) such as `uk`, `soc` and `arb`, Recoloring significantly increases the number of color classes produced over the initial coloring. Recoloring re-colors the vertices of the graph with a different order compared to that used in the initial graph coloring scheme, which in turn may introduce new additional color classes. Third, we find that BalColorTM provides the best color balancing quality compared to all prior state-of-the-art balanced graph coloring schemes. Specifically, the color balancing quality of BalColorTM is 2.26×, 14.82× and 413.31× better compared to that of CLU, VFF and Recoloring, respectively. Overall, we conclude that our proposed BalColorTM provides the best color balancing quality over prior state-of-the-art schemes in all large real-world graphs.

To better illustrate the effect of balancing the vertices across color classes, we present in Figure 18 the sizes of all the color classes produced by ColorTM, CLU, VFF, Recoloring and BalColorTM for a representative subset of our evaluated real-world graphs. Specifically, the x axis represents the color index i of each color class produced in the input graph, and the y axis shows how many of the vertices of the graph have been colored with the color i. The `uk`, `soc` and `arb` graphs are web social networks [44] with a highly power-law distribution [41–43]: only a *few* vertices have a very *high* degree, while the vast majority of the remaining vertices of the graph has very low degree. In such graphs, ColorTM inserts the vast majority of the vertices in the first few color classes, and the remaining *few* vertices are assigned to different separate color classes. Moreover, as already explained, Recoloring introduces a large number of *new* additional color classes in such real-world graphs.

5.3.2 Performance Comparison

Figure 19 evaluates the scalability achieved by all balanced graph coloring implementations in a representative subset of our evaluated large real-world graphs, as the number of threads increases from 1 to 56, i.e., up to the maximum available hardware thread capacity of our machine. We present the execution time of *only* the kernel that balances the vertices across color classes (excluding the execution time of the initial graph coloring).

We draw three findings. First, we observe that Recoloring achieves the worst performance over all balanced graph coloring schemes. Even in the single-threaded executions, Recoloring performs by 3.21×, 2.26× and 3.69× worse than CLU, VFF and BalColorTM, respectively, because it executes a much larger amount of computation, memory accesses and synchronization. Recall that Recoloring processes and re-colors *all* the vertices of the graph, while

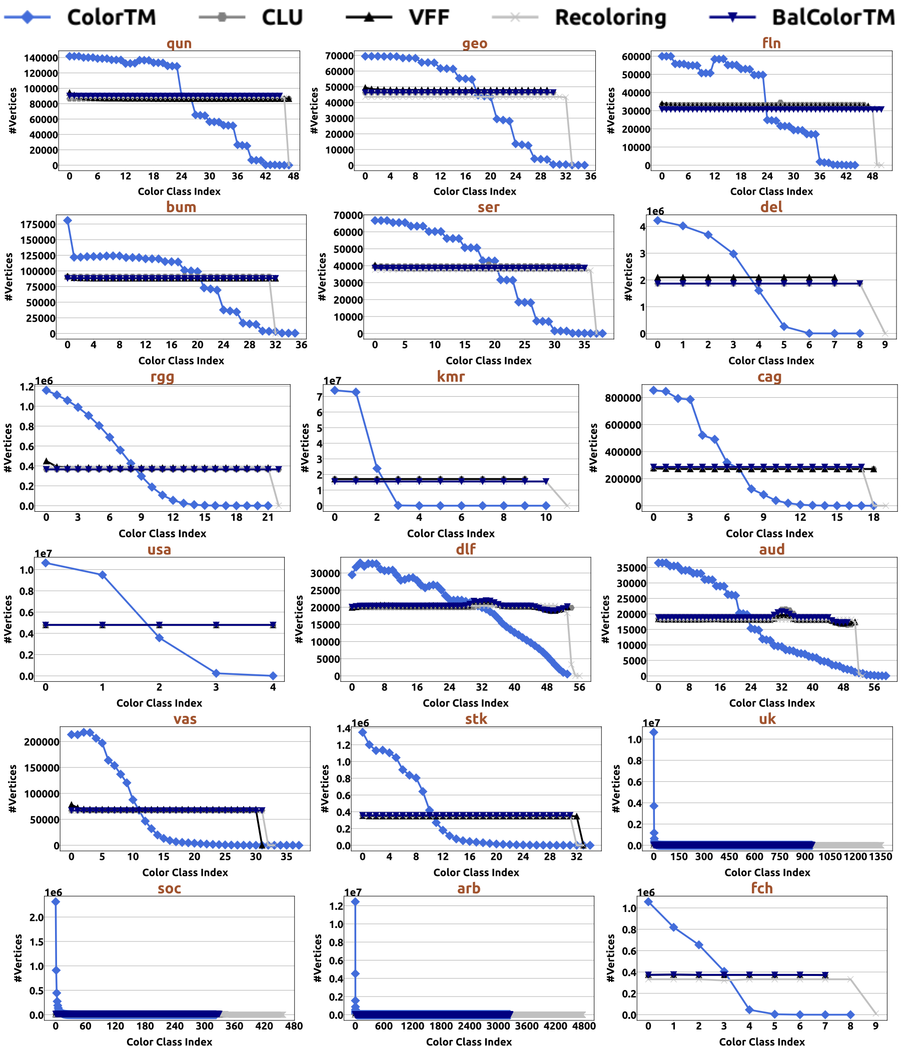

Fig. 18 Distribution of color class sizes produced by ColorTM and all our evaluated balanced graph coloring schemes. Note that small color class sizes result in reduced parallelism in the real-world end-application.

the remaining balanced graph coloring schemes re-color only a *subset* of the vertices of the graph. Note that in `uk` and `arb` graphs, all balanced graph coloring schemes need to re-color a *large* portion of the graph's vertices, thus performing closely to each other. Second, we find that the scalability of all schemes is affected by the NUMA effect, however BalColorTM on average scales well even when using all available hardware threads and both NUMA sockets of our machine. When increasing the number of threads from 28 to 56, the performance of BalColorTM improves by 1.55× averaged across all large graphs. Third, we find that in contrast to the graph coloring kernel, in many real-world graphs the performance of the balanced graph coloring

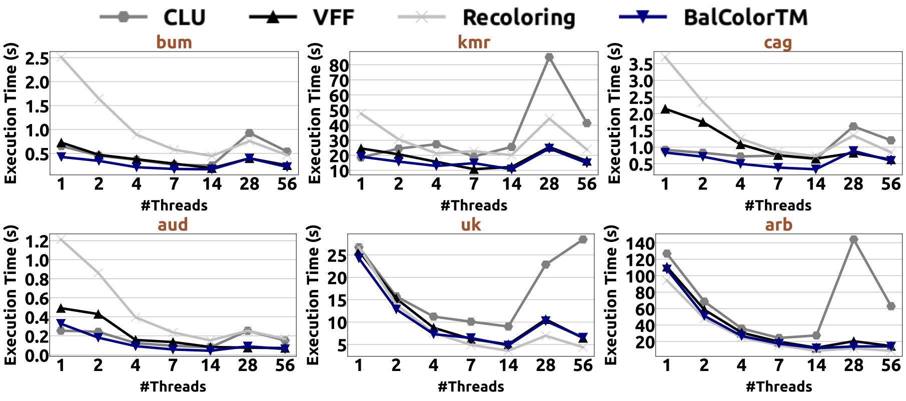

Fig. 19 Scalability achieved by all balanced graph coloring implementations in large real-world graphs.

kernel scales up to 14 threads, and degrades when using 56 threads. This is because the balanced graph coloring kernel has a lower amount of parallelism (a small subset of the vertices of the graph are re-colored by parallel threads) than the graph coloring kernel. Thus, our analysis demonstrates that when a kernel has low levels of parallelism, the best performance is achieved using a smaller number of parallel threads than the available hardware threads on the multicore platform. To this end, we suggest software designers of real-world end-applications to *on-the-fly* adjust the number of parallel threads used to parallelize each different sub-kernel of the end-application based on the parallelization needs of each particular sub-kernel.

Figure 20 compares the speedup achieved by all balanced graph coloring schemes normalized to the CLU scheme in all large real-world graphs. We compare the actual kernel time that balances the vertices across color classes.

We observe that BalColorTM outperforms all prior state-of-the-art balanced graph coloring schemes across all various large real-world graphs with a large number of parallel threads used. BalColorTM outperforms CLU, VFF and Recoloring by on average 1.89×, 1.33× and 2.06× respectively, when using 14 threads. Moreover, BalColorTM outperforms CLU, VFF and Recoloring by on average 2.61×, 1.05× and 1.68× respectively, when using 56 threads, i.e., the maximum hardware thread capacity of our machine. Overall, BalColorTM performs best over all prior schemes in all large real-world graphs. Therefore, considering the fact that BalColorTM also provides the best color balancing quality over prior schemes, we conclude that our proposed algorithmic design is a highly efficient and effective parallel graph coloring algorithm for modern mutlicore platforms.

To confirm the performance benefits of BalColorTM across multiple computing platforms, we evaluate all schemes on a 2-socket Intel Broadwell server with an Intel Xeon E5-2699 v4 processor at 2.2 GHz having 44 physical cores and 88 hardware threads. Figure 21 compares the speedup achieved by all balanced graph coloring schemes normalized to the CLU scheme in all large

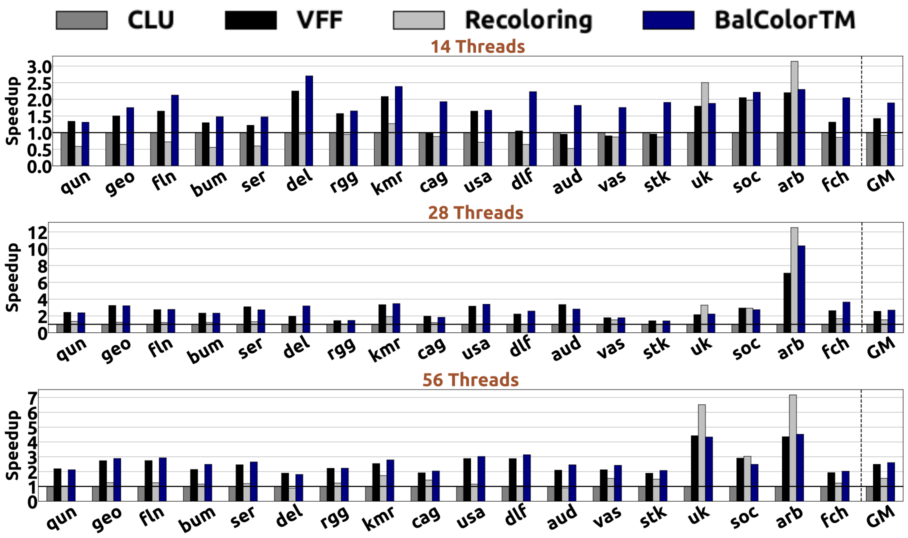

Fig. 20 Speedup achieved by all balanced graph coloring implementations over the CLU scheme in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads).

real-world graphs using 88 threads, i.e., the maximum hardware thread capacity of the Intel Broadwell server. We find that BalColorTM provides significant performance benefits over prior state-of-the-art graph coloring algorithms, achieving 1.82×, 1.22× and 1.84× better performance over CLU, VFF, and Recoloring, respectively.

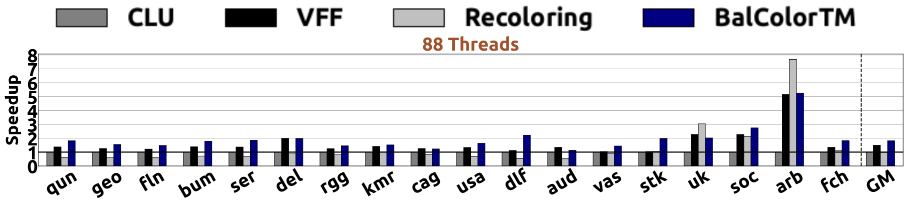

Fig. 21 Speedup achieved by all balanced graph coloring implementations over the CLU scheme in large real-world graphs using the maximum hardware thread capacity of an Intel Broadwell server with hyperthreading enabled (88 threads).

5.4 Analysis of a Real-World Scenario

In this section, we study the performance benefits of our proposed graph coloring schemes, i.e., ColorTM and BalColorTM, when parallelizing a widely used real-world end-application, i.e., Community Detection, via chromatic scheduling. Specifically, we compare the following parallel implementations to execute the Community Detection application:

- The parallelization scheme for the Louvain method [45–47] provided by Grappolo suite [48], henceforth referred to as SimpleCD, in which the vertices are processed as they appear in the input graph representation.

The algorithm consists of multiple iterations. First, each vertex is placed in a community of its own. Then, multiple iterations are performed until a convergence criterion is met. Within each iteration, all vertices are processed concurrently by multiple parallel threads, and a greedy decision is made to decide whether each vertex should be moved to a different community (selected from one of its adjacent vertices) or should remain in its current community, targeting to maximize the net modularity gain. For more details, we refer the reader to [45, 49–51].

- The chromatic scheduling parallelization approach using ColorTM to color the vertices of the graph, henceforth referred to as ColorTMCD, in which the vertices are processed in the order they are distributed in the color classes. The end-to-end Community Detection execution can be broken down in two steps: (i) the time to color the vertices of the graph with ColorTM, and (ii) the time to classify the vertices of the graph into communities via chromatic scheduling parallelization approach. The (ii) step processes the color classes produced by the (i) step sequentially, and all vertices of the same color class are processed in parallel.
- The chromatic scheduling parallelization approach using ColorTM to color the vertices of the graph and BalColorTM to balance the vertices across color classes produced, henceforth referred to as BalColorTMCD, in which the vertices are processed in the order they are distributed in the color classes. The end-to-end Community Detection execution can be broken down in three steps: (i) the time to color the vertices of the graph with ColorTM, (ii) the time to balance the vertices of the graph across color classes, and (iii) the time to classify the vertices of the graph into communities via chromatic scheduling parallelization approach. The (iii) step processes the color classes produced by the (ii) step sequentially, and all vertices of the same color class are processed in parallel.

Figure 22 evaluates the scalability of all the end-to-end Community Detection parallel implementations in a representative subset of large real-world graphs, as the number of parallel threads increases. We present the *total* end-to-end execution time, i.e., in ColorTMCD we account for the time to color the vertices of the graph (coloring step), and in BalColorTMCD we account for the time to color the vertices of the graph (coloring step), and the time to balance the vertices across color classes (balancing step).

We draw two findings. First, we find that ColorTMCD and BalColorTMCD scale well in large real-world graphs. For example, when increasing the number of threads from 1 to 56, ColorTMCD improves performance by 12.34× and 3.44× in `bum` and `arb` graphs, respectively. Similarly, when increasing the number of threads from 1 to 56, BalColorTMCD improves performance by 11.38× and 3.63× in `bum` and `arb` graphs, respectively. However, we observe that in `uk` and `arb` graphs, SimpleCD outperforms both ColorTMCD and BalColorTMCD. In these two graphs, ColorTM and BalColorTM produce the largest number of color classes compared to all the remaining real-world graphs (See Table 3), i.e., they produce 944 and 3248 colors for the `uk` and `arb` graphs,

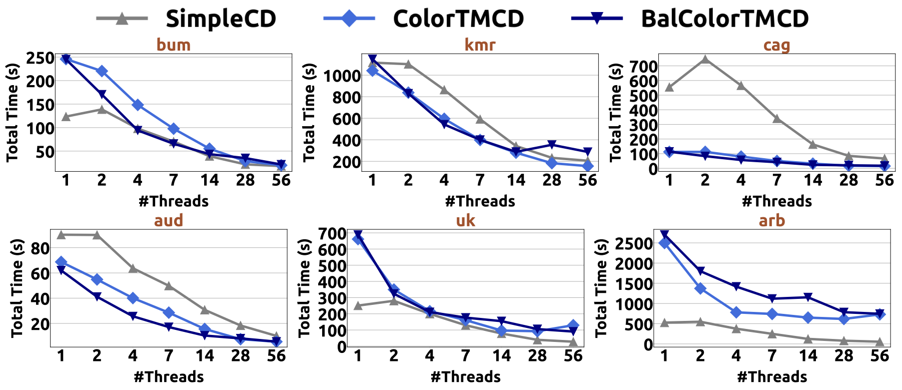

Fig. 22 Scalability of the end-to-end Community Detection execution achieved by (i) the Grappolo [48] parallelization approach of the Louvain method (SimplCD) and (ii) the chromatic scheduling parallelization approach with ColorTM (ColorTMCD) and (iii) the chromatic scheduling parallelization approach with both ColorTM and BalColorTM (BalColorTMCD) in large real-world graphs.

respectively. As a result, in `uk` and `arb` graphs the chromatic scheduling parallelization approach of ColorTMCD and BalColorTMCD executes 944 and 3248 times of *barrier* synchronization among parallel threads, respectively, thus incurring higher synchronization costs over SimpleCD. Second, the scalability of BalColorTMCD is affected more by the NUMA effect compared to that of ColorTMCD. Specifically, when increasing the number of threads from 14 to 28, the performance of ColorTMCD improves by 1.63× averaged across all real-world graphs, while the performance of BalColorTMCD only improves by 1.22×. Similarly, when increasing the number of threads from 14 to 56, the performance of ColorTMCD improves by 1.98×, while the performance of BalColorTMCD improves by 1.50×. We find that even though balancing the sizes of color classes provides higher load balance across parallel threads of real-world end-applications, it might because more remote expensive memory accesses across NUMA sockets of modern multicore machines.

Figure 23 shows the actual kernel time (without accounting for performance overheads introduced by the coloring and balancing steps) of Community Detection by comparing the speedup of ColorTMCD and BalColorTMCD over SimpleCD in all our evaluated large real-world graphs.

We draw two key findings. First, BalColorTM can on average outperform ColorTM, when considering only the actual kernel time of Community Detection, by providing better load balance among parallel threads. When only the actual kernel time of Community Detection is considered (excluding the performance overheads introduced by the coloring and balancing steps), BalColorTMCD on average outperforms ColorTMCD by 1.27×, 1.01× and 1.12× when using 14, 28, and 56 threads, respectively. Second, parallelizing the Community Detection using ColorTM and BalColorTM provides significant performance speedups over SimpleCD, the state-of-the-art paralellization approach of Louvain method of Community Detection [45–48]. Specifically,

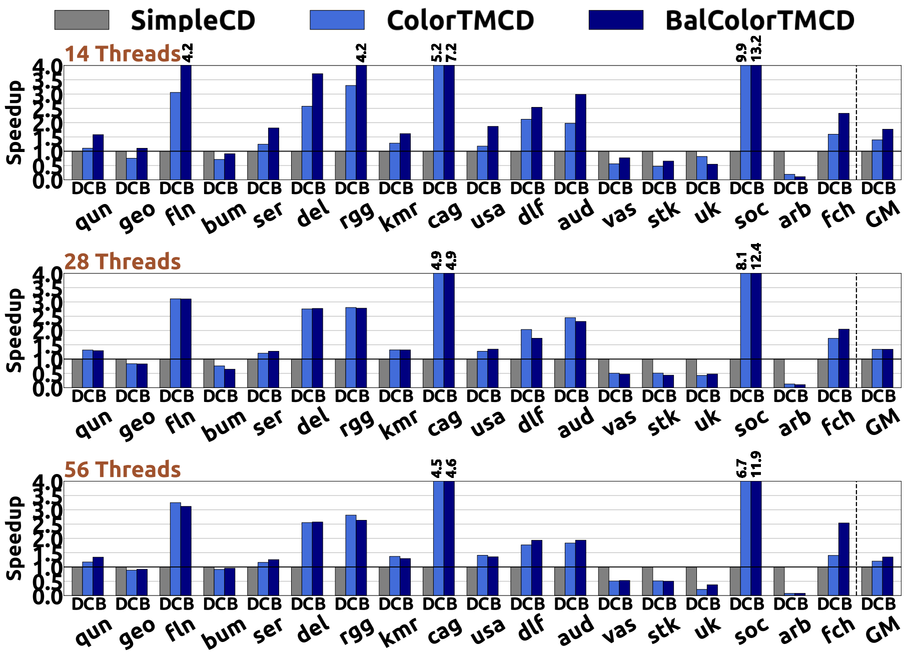

Fig. 23 Speedup of the actual kernel of the Community Detection execution achieved by (i) SimpleCD (D), (ii) ColorTMCD (C) and (iii) BalColorTMCD (B) in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads).

ColorTMCD improves the performance of the actual kernel time of Community Detection compared to SimpleCD by 1.40×, 1.34×, and 1.20×, when using 14, 28, and 56 threads, respectively. In addition, BalColorTMCD improves the performance of the actual kernel time of Community Detection compared to SimpleCD by 1.77×, 1.34×, and 1.34×, when using 14, 28, and 56 threads, respectively. We conclude that our proposed graph coloring algorithmic designs can provide high performance benefits in real-world end-applications which are parallelized using coloring.

Figure 24 presents the speedup breakdown of ColorTMCD and BalColorTMCD over SimpleCD in all our evaluated large real-world graphs. The performance is broken down in three steps: (i) the coloring step to color the vertices of the graph (**Coloring**), (ii) the balancing step to balance the vertices across color classes (**Balancing**), and (iii) the actual Community Detection kernel time (**CommunityDetection**).

We make two key observations. First, BalColorTMCD on average outperforms ColorTMCD when using up to 14 threads (using one single NUMA socket). When considering the end-to-end execution including the performance overheads introduced by the coloring and balancing steps, BalColorTMCD outperforms ColorTMCD by 1.19× when using 14 threads, while it performs on average 1.18× and 1.10× worse over ColorTMCD, when using 28 and 56

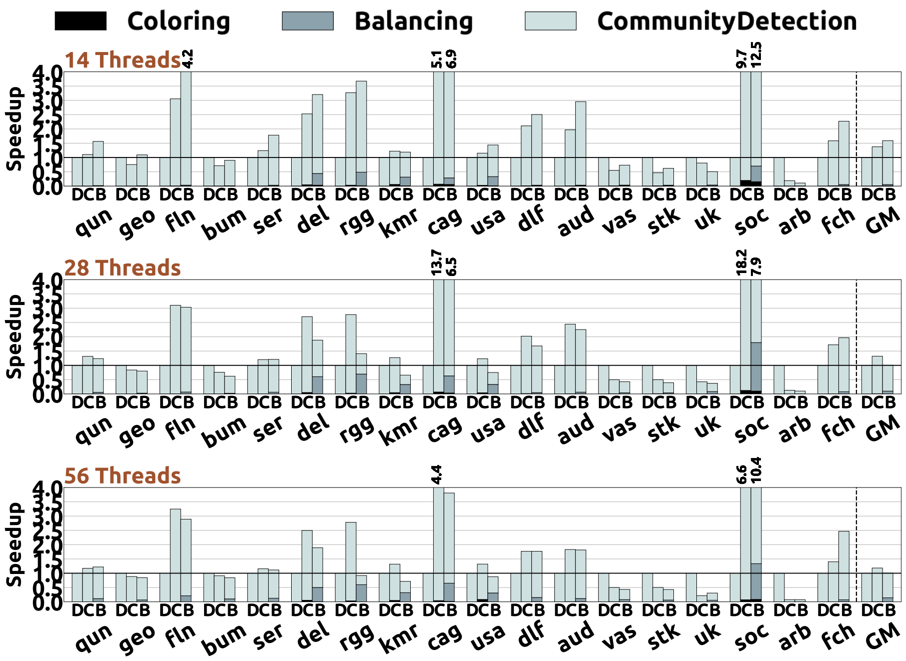

Fig. 24 Speedup breakdown of the end-to-end Community Detection execution achieved by (i) SimpleCD (D), (ii) ColorTMCD (C) and (iii) BalColorTMCD (B) in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads).

threads, respectively. We find that the performance overhead introduced in the balancing step of BalColorTMCD is not compensated in the runtime of the actual kernel time of Community Detection when using both NUMA sockets of our machine. Second, we observe that both ColorTMCD and BalColorTMCD can provide high performance in Community Detection. ColorTMCD on average outperforms SimpleCD by 1.38×, 1.33× and 1.19×, when using 14, 28 and 56 threads, respectively. BalColorTMCD on average outperforms SimpleCD by 1.64×, 1.10× and 1.08×, when using 14, 28 and 56 threads, respectively. In addition, we observe that BalColorTMCD provides significant performance speedups over Simple CD in many graphs such as `fln`, `del`, `cag`, `aud`, `soc` and `fch`, reaching up to 10.36× with 56 threads. Overall, we conclude that our proposed parallel graph coloring algorithms can provide significant performance improvements in real-world end-applications, e.g., parallelizing Community Detection with chromatic scheduling, across a wide variety of input data sets with diverse characteristics.

6 Related Work

A handful of prior works [1, 20, 21, 27–31, 33, 52, 52–54] has examined the graph coloring kernel in modern multicore platforms. Welsh and Powell [1] propose the original sequential Greedy algorithm that colors the vertices of the graph using the *first-fit* heuristic. Recent prior works [27–30] parallelize Greedy by proposing the SeqSolve, IterSolve and IterSolveR schemes described in Section 2.2. We compare ColorTM with these prior schemes in Section 5.2, and demonstrate that our proposed ColorTM outperforms these state-of-the-art schemes across a wide variety of real-world graphs. Jones and Plassmann [53] design an algorithm, named JP, that colors the vertices of the graph by identifying independent sets of vertices: in each iteration, the algorithm finds and selects an independent set of vertices that can be colored concurrently. However, JP is a recursive algorithm that typically runs longer than the original Greedy [20, 21, 33], since it performs more computations and needs more synchronization points, i.e., parallel threads need to synchronize at each iteration of processing independent sets of vertices. Moreover, the original paper [53] shows that JP provides good performance mostly in $\mathcal{O}(1)$-degree graphs. In contrast, our work efficiently parallelizes the original and widely used Greedy algorithm for graph coloring, and our proposed parallel algorithms achieve significant performance improvements across a wide variety of real-world graphs and using a large number of parallel threads.

Deveci et al. [54] present an edge-centric parallelization scheme for graph coloring which is better suited for GPUs. ColorTM and BalColorTM can be straightforwardly extended to color the vertices of a graph by equally distributing the edges of the graph among parallel threads. We leave the exploration of edge-centric graph coloring schemes for future work. Future work also comprises the experimentation of the graph coloring kernel on multicore computing platforms such as modern GPUs [55–58] and Processing-In-Memory systems [18, 41, 42, 59–65]. Maciej et al. [20] and Hasenplaugh et al. [21] propose new vertex *ordering* heuristics for graph coloring. Ordering heuristics define the order in which Greedy colors the vertices of the graph in order to improve the coloring quality by minimizing the number of colors used. Instead, our work aims to improve system performance by proposing efficient parallelization schemes. For a fair comparison, we employ the *first-fit* ordering heuristic (the vertices of the graph are colored in the order they appear in the input graph representation) in all parallel algorithms evaluated in Sections 5.2 and 5.3. ColorTM and BalColorTM can support various ordering heuristics [3, 4, 11, 20–26, 66] by assigning the vertices of the graph to parallel threads with a particular order. We leave the evaluation of various vertex ordering heuristics for future work.

Lu et al. [31] design *balanced* graph coloring algorithms to efficiently balance the vertices across the color classes. We compare BalColorTM with their proposed algorithms, i.e., CLU, VFF, Recoloring, in Section 2.3, and demonstrate that our proposed BalColorTM scheme on average performs best across all large real-world graphs. Tas et al. [52] propose balanced graph coloring algorithms for bitpartie graphs, i.e., graphs whose vertices can be divided into two disjoint and independent sets U and V, and every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In contrast, ColorTM and BalColorTM are designed to be general, and efficiently color any *arbitrary* real-world graph using a large number of parallel threads. In addition, Tas et al. [52] also explore the distance-2 graph coloring kernel on multicore architectures, in which any two vertices u and v with an edge-distance at most 2 are assigned with different colors. Instead, our work efficiently parallelizes the distance-1 graph coloring kernel on multicore platforms, in which any two adjacent vertices of the graph connected with a *direct* edge are assigned with different colors. Finally, prior works propose algorithms for edge coloring [67], dynamic or streaming coloring [68–74], k-distance coloring [75, 76] and sequential exact coloring [77–80]. All these works are not closely related to our work, since we focus on designing high-performance parallel algorithms for the distance-1 vertex graph coloring kernel.

7 Conclusion

In this work, we explore the graph coloring kernel on multicore platforms, and propose ColorTM and BalColorTM, two novel algorithmic designs for high performance and balanced graph coloring on modern computing platforms. ColorTM and BalColorTM achieve high system performance through two key techniques: (i) *eager* conflict detection and resolution of the coloring inconsistencies that arise when adjacent vertices are concurrently processed by different parallel threads, and (ii) *speculative* computation and synchronization among parallel threads by leveraging Hardware Transactional Memory. Via the eager coloring conflict detection and resolution policy, ColorTM and BalColorTM effectively leverage the deep memory hierarchy of modern multicore platforms and minimize access costs to application data. Via the speculative computation and synchronization approach, ColorTM and BalColorTM minimize synchronization costs among parallel threads and provide high amount of parallelism. Our evaluations demonstrate that our proposed parallel graph coloring algorithms outperform prior state-of-the-art approaches across a wide range of large real-world graphs. ColorTM and BalColorTM can also provide significant performance improvements in real-world scenarios. We conclude that ColorTM and BalColorTM are highly efficient graph coloring algorithms for modern multicore systems, and hope that this work encourages further studies of the graph coloring kernel in modern computing platforms.

8 Acknowledgments

We thank the CSLAB Research Group members of ECE NTUA for continued and undivided support, insightful comments and valuable feedback. Christina Giannoula is funded for her postgraduate studies from the Foundation for Education and European Culture. The ColorTM and BalColorTM algorithms are publicly available [32] at https://github.com/cgiannoula/ColorTM.git.

9 Appendix

9.1 Analysis of ColorTM and BalColorTM Execution

We further analyze the HTM-related execution behavior of our proposed ColorTM and BalColorTM algorithms. Figure 25 presents the abort ratio of

ColorTM, i.e., the number of transactional aborts divided by the number of attempted transactions, in all real-world graphs, as the number of threads increases. In the 14-thread execution, we pin all thread on one single NUMA socket. In the 28-thread execution, we pin threads on both NUMA sockets of our machine with hyperthreading disabled. In the (14+14)-thread execution, we pin all 28 threads on the same *single* socket with hyperthreading enabled. In the 56-thread execution, we use the maximum hardware thread capacity of our machine.

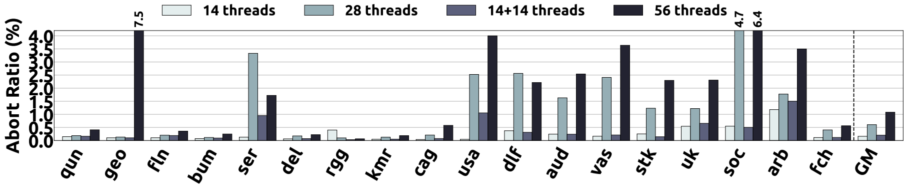

Fig. 25 Abort ratio exhibited by ColorTM in all large real-world graphs.

We make three key observations. First, we find that the abort ratio becomes high in real-world graphs which have high maximum degree and high standard deviation of the vertices' degrees, e.g., `dlf`, `aud`, `vas`, `stk`, `uk`, `soc` and `arb` graphs. In graphs with high vertex degree, the transaction data access footprint is large and parallel threads compete for the same adjacent vertices with a high probability, thus causing aborts in HTM. Second, we observe that when using both sockets of our machine, the transactional aborts in ColorTM significantly increase due to the NUMA effect. Specifically, averaged across all graphs the (14+14)-thread execution of ColorTM exhibits 2.97× lower abort ratio compared to the 28-thread execution of ColorTM. Due to the NUMA effect, the memory accesses to the application data are very expensive. As a result, the duration of the transactions increases, thus increasing the probability of conflict aborts among running transactions (See more details in the next experiment). Third, we observe that ColorTM exhibits a very low abort ratio. ColorTM has *only* 1.08% abort ratio on average across all real-world graphs, when using the maximum hardware thread capacity (56 threads) of our machine. Our proposed *speculative* algorithmic design effectively reduces the amount of computations and data accesses performed inside the critical section (inside the HTM transaction), thus effectively decreasing the transaction's footprint and duration. As a result, ColorTM provides high amount of parallelism and low interference among parallel threads. We conclude that ColorTM has low synchronization and interference costs among a large number of parallel threads, even in real-world graphs with high vertex degree.

Figure 26 presents the breakdown of different types of aborts exhibited by ColorTM in a representative subset of real-world graphs. We break down the transactional aborts into four types: (i) *conflict* aborts: they appear when a running transaction executed by a parallel thread attempts to write the read-set of another running transaction executed by a different thread, (ii) *capacity* aborts: they appear when the memory footprint of a running transaction exceeds the size of the hardware transactional buffers, (iii) *lock* aborts: current HTM implementations [37–40] provide no guarantee that any transaction will eventually commit inside the transactional path, and thus the programmer provides an alternative non-transactional fallback path, i.e., falling back to the acquisition of coarse-grained lock that allows only a single thread to enter the critical section, and forces aborts to the transactions of all the remaining threads [3], and (iv) *other* aborts: they appear when a transaction fails due to other reasons such as cache line evictions, interrupts and/or when the duration of a transaction exceeds the scheduling quantum and the OS scheduler schedules out the software thread from the hardware thread, aborting the transaction. Note that since the fallback path lock is just a variable in the source code, some conflict aborts are caused by the writes in this lock variable. Thus, a part of the lock aborts is counted as conflict aborts in our measurements.

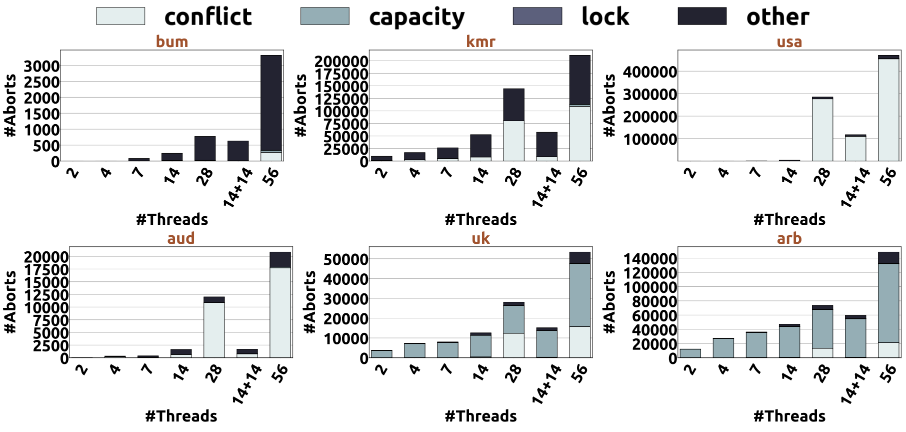

Fig. 26 Breakdown of different types of aborts exhibited by ColorTM in real-world graphs.

We draw three findings. First, we find that the *conflict* aborts significantly increase across all graphs when using both sockets of our machine due to the NUMA effect. For example, the number of conflicts aborts in the 28-thread executions is 3.32× higher compared to that in the 14-thread executions. As already mentioned, the NUMA effect significantly increases the duration of the running transactions, and thus the probability of causing conflict aborts among running transactions is high. Second, as number of threads increases, e.g., when comparing the 56-thread execution over the 28-thread execution, the number of conflict aborts increases by 1.05×. This is because partitioning the graph to a higher number of threads results in a higher number of crossing edges among parallel threads, which in turn results in a larger list of critical adjacent vertices that is validated inside the HTM transactions. Therefore, the transaction footprint increases, thus increasing the probability of causing

[3]To achieve this, the lock is added to each transaction's read set, so that when the lock is acquired by a thread (write to the lock variable), the remaining threads are aborted and wait until the lock is released.

conflict aborts. Third, we find that in graphs with very high maximum degree, e.g., `uk` and `arb` graphs, the capacity aborts constitute a large portion of total aborts. In such graphs, the data access footprint of the transactions is large, resulting in a high probability of exceeding the hardware buffers. Overall, our analysis demonstrates that current HTM implementations are severely limited by the NUMA effect [81], and incur high performance costs when using more than one NUMA socket on the machine. To this end, we recommend hardware designers to improve the HTM implementations in NUMA machines, and suggest software designers to propose intelligent algorithmic schemes and data partitioning approaches that minimize the expensive memory accesses to remote NUMA sockets inside the HTM transactions.

Figure 27 presents the abort ratio of BalColorTM, i.e., the number of transactional aborts divided by the number of attempted transactions, in all real-world graphs, as the number of threads increases. In the 14-thread execution, we pin all thread on one single socket. In the 28-thread execution, we pin threads on both NUMA sockets of our machine with hyperthreading disabled. In the (14+14)-thread execution, we pin all 28 threads on the same single socket with hyperthreading enabled. In the 56-thread execution, we use the maximum hardware thread capacity of our machine.

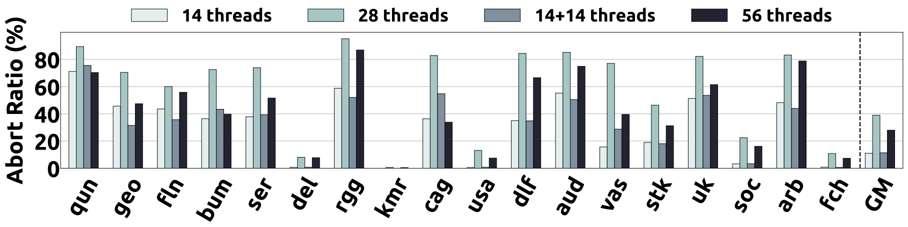

Fig. 27 Abort ratio exhibited by BalColorTM in all large real-world graphs.

We make two key observations. First, we observe that BalColorTM on average incurs higher abort ratio over ColorTM, reaching up to 80% abort ratio in some multithreaded executions. Specifically, BalColorTM incurs 68.55×, 64.35×, 55.83× and 25.91× higher abort ratio (averaged across all real-world graphs) over ColorTM, when using 14, 28, (14+14), and 56 threads, respectively. This is because BalColorTM processes and re-colors a much smaller number of vertices (a small subset of the vertices of the graph) compared to ColorTM, which instead processes and colors *all* the vertices of the graph. As a result, parallel threads compete for the same data and memory locations with a much higher probability in BalColorTM compared to ColorTM, thus incurring higher abort ratio and synchronization costs. Second, we find that in *all* real-world graphs the vast majority of transactional aborts are *conflict* aborts. Specifically, the portion of conflict aborts is more than 95% in all real-world graphs for all multithreaded executions. Typically, the lower parallelization needs a parallel kernel has, the higher data contention among parallel threads it incurs. Overall, our analysis demonstrates that using a high number of parallel threads results in high contention on shared data due to low amount of parallelism of the balanced graph coloring kernel. The aforementioned high contention causes high synchronization overheads. To this end, we recommend software designers of real-world end-applications to design adaptive parallelization schemes that trade off the amount of parallelism provided for lower synchronization costs.

10 Declarations

Ethical Approval.
Not applicable.
Competing interests.
Not applicable.
Authors' contributions.
Christina Giannoula contributed to the conception and design of the study, the technical implementation and the experimental characterization of the proposed algorithms and the writing and the review of the manuscript.

Athanasios Peppas contributed to the technical implementation, the experimental characterization and analysis of the proposed algorithms.

Georgios Goumas contributed to the design of the study, the review of the manuscript, the supervision and the financial support for the research leading to this publication.

Nectarios Koziris contributed to the review of the manuscript, the supervision and the financial support for the research leading to this publication.

Funding.
Christina Giannoula receives a PhD award from the Foundation for Education and European Culture.

Availability of data and materials.
Not applicable.

References

[1] Welsh, D.J.A., Powell, M.B.: An Upper Bound for the Chromatic Number of a Graph and its Application to Timetabling Problems. The Computer Journal **10**(1), 85–86 (1967)

[2] Marx, D.: Graph Coloring Problems and Their Applications in Scheduling. In: Proc. John Von Neumann PhD Students Conference, vol. 48, pp. 11–16 (2004)

[3] Arkin, E.M., Silverberg, E.B.: Scheduling Jobs with Fixed Start and End Times. Discrete Applied Mathematics **18**(1), 1–8 (1987)

[4] Marx, D.: Graph Colouring Problems and their Applications in Scheduling. Periodica Polytechnica Electrical Engineering **48**, 11–16 (2004)

[5] Ramaswami, R., Parhi, K.K.: Distributed Scheduling of Broadcasts in a Radio Network. In: IEEE INFOCOM, pp. 497–5042 (1989)

[6] Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register Allocation via Coloring. Computer Languages

6(1), 47–57 (1981)

[7] Chaitin, G.J.: Register Allocation & Spilling via Graph Coloring. In: SIGPLAN Symposium on Compiler Construction, vol. 17, pp. 98–101 (1982)

[8] Briggs, P., Cooper, K.D., Torczon, L.: Improvements to Graph Coloring Register Allocation. TOPLAS **16**(3), 428–455 (1994)

[9] Chen, W.-Y., Lueh, G.-Y., Ashar, P., Chen, K., Cheng, B.: Register Allocation for Intel Processor Graphics. In: CGO, pp. 352–364 (2018)

[10] Cohen, A., Rohou, E.: Processor Virtualization and Split Compilation for Heterogeneous Multicore Embedded Systems. In: DAC, pp. 102–107 (2010)

[11] Coleman, T.F., Moré, J.J.: Estimation of Sparse Jacobian Matrices and Graph Coloring Problems. SIAM Journal on Numerical Analysis **20**(1), 187–209 (1983)

[12] Saad, Y.: SPARSKIT: a basic tool kit for sparse matrix computations - Version 2 (1994)

[13] Jones, M.T., Plassmann, P.E.: The Efficient Parallel Iterative Solution of Large Sparse Linear Systems. In: George, A., Gilbert, J.R., Liu, J.W.H. (eds.) Graph Theory and Sparse Matrix Computation, pp. 229–245 (1993)

[14] Gebremedhin, A.H., Manne, F., Pothen, A.: What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. SIAM Review **47**(4), 629–705 (2005)

[15] Kaler, T., Hasenplaugh, W., Schardl, T.B., Leiserson, C.E.: Executing Dynamic Data-Graph Computations Deterministically Using Chromatic Scheduling. ACM TOPC **3**(1) (2016)

[16] Kaler, T., Hasenplaugh, W., Schardl, T.B., Leiserson, C.E.: Executing Dynamic Data-graph Computations Deterministically Using Chromatic Scheduling. In: SPAA, pp. 154–165 (2014)

[17] Strati, F., Giannoula, C., Siakavaras, D., Goumas, G., Koziris, N.: An Adaptive Concurrent Priority Queue for NUMA Architectures. In: CF, pp. 135–144 (2019)

[18] Giannoula, C., Vijaykumar, N., Papadopoulou, N., Karakostas, V., Fernandez, I., Gómez-Luna, J., Orosa, L., Koziris, N., Goumas, G., Mutlu, O.: *SynCron*: Efficient Synchronization Support for Near-Data-Processing Architectures. In: HPCA, pp. 263–276 (2021)

[19] Garey, M.R., Johnson, D.S., Stockmeyer, L.: Some Simplified NP-Complete Problems. In: STOC, pp. 47–63 (1974)

[20] Besta, M., Carigiet, A., Janda, K., Vonarburg-Shmaria, Z., Gianinazzi, L., Hoefler, T.: High-Performance Parallel Graph Coloring with Strong Guarantees on Work, Depth, and Quality. In: SC, pp. 1–17 (2020)

[21] Hasenplaugh, W., Kaler, T., Schardl, T.B., Leiserson, C.E.: Ordering Heuristics for Parallel Graph Coloring. In: SPAA, pp. 166–177 (2014)

[22] Brélaz, D.: New Methods to Color the Vertices of a Graph. Communications of ACM **22**(4), 251–256 (1979)

[23] Matula, D.W., Beck, L.L.: Smallest-Last Ordering and Clustering and Graph Coloring Algorithms. J. ACM **30**(3), 417–427 (1983)

[24] Karp, R.M., Wigderson, A.: A Fast Parallel Algorithm for the Maximal Independent Set Problem. Journal of the ACM **32**(4), 762–773 (1985)

[25] Luby, M.: A Simple Parallel Algorithm for the Maximal Independent Set Problem. In: STOC, vol. 7, pp. 567–583 (1985)

[26] Goldberg, M., Spencer, T.: A New Parallel Algorithm for the Maximal Independent Set Problem. In: SFCS, pp. 161–165 (1987)

[27] Çatalyürek, Ü.V., Feo, J., Gebremedhin, A.H., Halappanavar, M., Pothen, A.: Graph Coloring Algorithms for Muti-core and Massively Multithreaded Architectures. Parallel Computing **38**(10), 576–594 (2012)

[28] Gebremedhin, A.H., Manne, F.: Scalable Parallel Graph Coloring Algorithms. Concurrency: Practice and Experience **12**(12), 1131–1146 (2000)

[29] Rokos, G., Gorman, G., Kelly, P.H.J.: A Fast and Scalable Graph Coloring Algorithm for Multi-core and Many-core Architectures. Euro-Par, 414–425 (2015)

[30] Boman, E.G., Bozdağ, D., Catalyurek, U., Gebremedhin, A.H., Manne, F.: A Scalable Parallel Graph Coloring Algorithm for Distributed Memory Computers. EuroPar, 241–251 (2005)

[31] Lu, H., Halappanavar, M., Chavarría-Miranda, D., Gebremedhin, A., Kalyanaraman, A.: Balanced Coloring for Parallel Computing Applications. In: IEEE IPDPS, pp. 7–16 (2015)

[32] Christina Giannoula: ColorTM: A High-Performance Graph Coloring Algorithm. https://github.com/cgiannoula/ColorTM.git

[33] Giannoula, C., Goumas, G., Koziris, N.: Combining HTM with RCU to Speed up Graph Coloring on Multicore Platforms. In: ISC HPC, pp. 350–369 (2018)

[34] Fortunato, S.: Community Detection in Graphs. Physics Reports **486**(3-5), 75–174 (2010)

[35] Mitchem, J.: On Various Algorithms for Estimating the Chromatic Number of a Graph. The Computer Journal **19**(2), 182–183 (1976)

[36] Lovász, L.M., Saks, M.E., Trotter, W.T.: An On-Line Graph Coloring Algorithm with Sublinear Performance Ratio. Discrete Mathematics **75**(1), 319–325 (1989)

[37] Herlihy, M., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: ISCA, pp. 289–300 (1993)

[38] Yoo, R.M., Hughes, C.J., Lai, K., Rajwar, R.: Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing. In: SC (2013)

[39] Cain, H.W., Michael, M.M., Frey, B., May, C., Williams, D., Le, H.: Robust Architectural Support for Transactional Memory in the Power Architecture. In: ISCA, pp. 225–236 (2013)

[40] Wang, A., Gaudet, M., Wu, P., Amaral, J.N., Ohmacht, M., Barton, C., Silvera, R., Michael, M.: Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In: PACT, pp. 127–136 (2012)

[41] Giannoula, C., Fernandez, I., Gómez-Luna, J., Koziris, N., Goumas, G., Mutlu, O.: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. In: SIGMETRICS, pp. 33–34 (2022)

[42] Giannoula, C., Fernandez, I., Luna, J.G., Koziris, N., Goumas, G., Mutlu, O.: SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. Proc. ACM Meas. Anal. Comput. Syst. **6**(1) (2022)

[43] Tang, W.T., Zhao, R., Lu, M., Liang, Y., Huyng, H.P., Li, X., Goh, R.S.M.: Optimizing and Auto-Tuning Scale-Free Sparse Matrix-Vector Multiplication on Intel Xeon Phi. In: CGO, pp. 136–145 (2015)

[44] Boldi, P., Vigna, S.: The WebGraph Framework I: Compression Techniques. In: WWW 2004, pp. 595–602 (2004)

[45] Lu, H., Halappanavar, M., Kalyanaraman, A.: Parallel Heuristics for Scalable Community Detection. Parallel Computing (2015)

[46] Chavarria-Miranda, D., Halappanavar, M., Kalyanaraman, A.: Scaling Graph Community Detection on the Tilera Many-Core Architecture. In: HiPC, vol. 47, pp. 19–37 (2014)

[47] Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E.: Fast Unfolding of Communities in Large Networks. JSTAT **2008**(10), 10008 (2008)

[48] ExaGraph: Grappolo: Parallel Clustering Using the Louvain Method as the Serial Template. https://github.com/Exa-Graph/grappolo

[49] Ghosh, S., Halappanavar, M., Tumeo, A., Kalyanaraman, A., Lu, H., Chavarrià-Miranda, D., Khan, A., Gebremedhin, A.: Distributed Louvain Algorithm for Graph Community Detection. In: IPDPS, pp. 885–895 (2018)

[50] Naim, M., Manne, F., Halappanavar, M., Tumeo, A.: Community Detection on the GPU. In: IPDPS, pp. 625–634 (2017)

[51] Halappanavar, M., Lu, H., Kalyanaraman, A., Tumeo, A.: Scalable Static and Dynamic Community Detection using Grappolo. In: HPEC, pp. 1–6 (2017)

[52] Tas, M.K., Kaya, K., Saule, E.: Greed Is Good: Parallel Algorithms for Bipartite-Graph Partial Coloring on Multicore Architectures. In: ICPP, pp. 503–512 (2017)

[53] Jones, M.T., Plassmann, P.E.: A Parallel Graph Coloring Heuristic. SIAM Journal on Scientific Computing **14**(3), 654–669 (1993)

[54] Deveci, M., Boman, E.G., Devine, K.D., Rajamanickam, S.: Parallel Graph Coloring for Manycore Architectures. In: IPDPS, pp. 892–901 (2016)

[55] Grosset, A.V.P., Zhu, P., Liu, S., Venkatasubramanian, S., Hall, M.: Evaluating Graph Coloring on GPUs. In: PPoPP, pp. 297–298 (2011)

[56] Osama, M., Truong, M., Yang, C., Buluç, A., Owens, J.: Graph Coloring on the GPU. In: IPDPSW, pp. 231–240 (2019)

[57] Chen, X., Li, P., Fang, J., Tang, T., Wang, Z., Yang, C.: Efficient and high-quality sparse graph coloring on gpus. Concurrency and Computation: Practice and Experience **29**(10), 4064 (2017)

[58] Che, S., Rodgers, G., Beckmann, B., Reinhardt, S.: Graph Coloring on the GPU and Some Techniques to Improve Load Imbalance. In: IPDPS, pp. 610–617 (2015)

[59] Fernandez, I., Quislant, R., Gutiérrez, E., Plata, O., Giannoula, C., Alser, M., Gómez-Luna, J., Mutlu, O.: NATSA: A Near-Data Processing Accelerator for Time Series Analysis. In: ICCD, pp. 120–129 (2020)

[60] Gómez-Luna, J., El Hajj, I., Fernandez, I., Giannoula, C., Oliveira, G.F., Mutlu, O.: Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-In-Memory Hardware. In: IGSC, pp. 1–7 (2021)

[61] Gómez-Luna, J., El Hajj, I., Fernandez, I., Giannoula, C., Oliveira, G.F., Mutlu, O.: Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. In: IEEE Access, vol. 10, pp. 52565–52608 (2022)

[62] Gao, M., Ayers, G., Kozyrakis, C.: Practical Near-Data Processing for In-Memory Analytics Frameworks. In: PACT, pp. 113–124 (2015)

[63] Ahn, J., Hong, S., Yoo, S., Mutlu, O.: A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In: ISCA, pp. 105–117 (2015)

[64] Nai, L., Hadidi, R., Sim, J., Kim, H., Kumar, P., Kim, H.: Graph-PIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In: HPCA, pp. 457–468 (2017)

[65] Zhuo, Y., Wang, C., Zhang, M., Wang, R., Niu, D., Wang, Y., Qian, X.: GraphQ: Scalable PIM-Based Graph Processing. In: MICRO, pp. 712–725 (2019)

[66] Alabandi, G., Powers, E., Burtscher, M.: Increasing the Parallelism of Graph Coloring via Shortcutting. In: PpopP, pp. 262–275 (2020)

[67] Holyer, I.: The NP-Completeness of Edge-Coloring. SIAM Journal on Computing **10**(4), 718–720 (1981)

[68] Sallinen, S., Iwabuchi, K., Poudel, S., Gokhale, M., Ripeanu, M., Pearce, R.: Graph Colouring as a Challenge Problem for Dynamic Graph Processing on Distributed Systems. In: SC, pp. 347–358 (2016)

[69] Yuan, L., Qin, L., Lin, X., Chang, L., Zhang, W.: Effective and Efficient Dynamic Graph Coloring. VLDB **11**(3), 338–351 (2017)

[70] Bossek, J., Neumann, F., Peng, P., Sudholt, D.: Runtime Analysis of Randomized Search Heuristics for Dynamic Graph Coloring. In: GECCO, pp. 1443–1451 (2019)

[71] Barba, L., Cardinal, J., Korman, M., Langerman, S., Renssen, A., Roeloffzen, M., Verdonschot, S.: Dynamic Graph Coloring. Algorithmica, 97–108 (2019)

[72] Bhattacharya, S., Chakrabarty, D., Henzinger, M., Nanongkai, D.: Dynamic Algorithms for Graph Coloring. In: ACM SIAM, pp. 1–20 (2018)

[73] Solomon, S., Wein, N.: Improved Dynamic Graph Coloring. TALG **16**(3), 1–24 (2020)

[74] Chakrabarti, A., Ghosh, P., Stoeckl, M.: Adversarially Robust Coloring for Graph Streams. arXiv preprint arXiv:2109.11130 (2021)

[75] Bozdağ, D., Çatalyürek, U.V., Gebremedhin, A.H., Manne, F., Boman, E.G., Özgüner, F.: Distributed-Memory Parallel Algorithms for Distance-2 Coloring and Related Problems in Derivative Computation. SIAM Journal on Scientific Computing **32**(4), 2418–2446 (2010)

[76] Bozdag, D., Çatalyürek, Ü.V., Gebremedhin, A.H., Manne, F., Boman, E.G., Özgüner, F.: A Parallel Distance-2 Graph Coloring Algorithm for Distributed Memory Computers. In: HPCC, pp. 796–806 (2005)

[77] Lin, J., Cai, S., Luo, C., Su, K.: A Reduction based Method for Coloring Very Large Graphs. In: IJCAI, pp. 517–523 (2017)

[78] Verma, A., Buchanan, A., Butenko, S.: Solving the Maximum Clique and Vertex Coloring Problems on Very Large Sparse Networks. INFORMS Journal on Computing **27**(1), 164–177 (2015)

[79] Hebrard, E., Katsirelos, G.: A Hybrid Approach for Exact Coloring of Massive Graphs. In: CPAIOR, pp. 374–390 (2019)

[80] Improving Probability Learning based Local Search for Graph Coloring. Applied Soft Computing **65**, 542–553 (2018)

[81] Brown, T., Kogan, A., Lev, Y., Luchangco, V.: Investigating the Performance of Hardware Transactions on a Multi-Socket Machine. In: SPAA, pp. 121–132 (2016)