2015 IEEE 29th International Parallel and Distributed Processing Symposium

Balanced Coloring for Parallel Computing Applications

Hao Lu[1], Mahantesh Halappanavar[2], Daniel Chavarría-Miranda[2], Assefaw Gebremedhin[1], and Ananth Kalyanaraman[1]

E-mail: luhowardmark@eecs.wsu.edu, {hala, daniel.chavarria}@pnnl.gov, {assefaw, ananth}@eecs.wsu.edu
[1] Washington State University [2] Pacific Northwest National Laboratory

***Abstract*—Graph coloring is used to identify subsets of independent tasks in parallel scientific computing applications. Traditional coloring heuristics aim to reduce the number of colors used as that number also corresponds to the number of parallel steps in the application. However, if the color classes produced have a skew in their sizes, utilization of hardware resources becomes inefficient, especially for the smaller color classes. Equitable coloring is a theoretical formulation of coloring that guarantees a perfect balance among color classes, and its practical relaxation is referred to as balanced coloring. In this paper, we revisit the problem of balanced coloring in the context of parallel computing. The goal is to achieve a balanced coloring of an input graph without increasing the number of colors that an algorithm oblivious to balance would have used. We propose and study multiple heuristics that aim to achieve such a balanced coloring, present parallelization approaches for multi-core and manycore architectures, and cross-evaluate their effectiveness with respect to the quality of balance achieved and performance. Furthermore, we study the impact of the proposed balanced coloring heuristics on a concrete application - viz. parallel community detection, which is an example of an irregular application. The thorough treatment of balanced coloring presented in this paper from algorithms to application is expected to serve as a valuable resource to parallel application developers who seek to improve parallel performance of their applications using coloring.**

***Keywords*-Balanced coloring; Tilera manycore architecture; community detection; graph algorithms.**

I. Introduction

Decomposing a computational task into constituent parts that can be executed simultaneously or identifying elements of composite data that can safely be updated simultaneously is a pervasive primitive in parallel computing. An immediately associated need is that of scheduling the identified subtasks (or data update operations) onto the processing units of a platform. In such a scenario, one would for performance reasons need to both maximize the amount of parallel execution (or data update) attained in a given step *and* minimize the total number of steps needed. In cases where the computational or data dependency between entities can be abstracted using a graph, this dual objective can be modeled and solved as a graph coloring problem.

The standard graph coloring problem aims at minimizing the number of colors used (that is, the number of independent subsets or color classes) without any requirement on the size of the color classes relative to each other. It therefore permits cases where the color classes can be highly unbalanced. In fact, by their nature, most practical algorithms for the standard graph coloring problem produce highly skewed color classes. This will be undesirable in the scenario sketched earlier because of load imbalance.

In this paper we consider the design, implementation and performance evaluation of algorithms for a variant of the graph coloring problem that also requires that color classes be *balanced* in their sizes. There is a body of work on *equitable colorings*—a theoretical formulation in which color classes are required to be *perfectly* balanced—in the graph theory literature, but little or no prior work exists on fast, practical balanced coloring algorithms and their parallelization on contemporary and emerging platforms. We seek to reduce this deficiency. The scope and contributions of the paper are as follows:

- ***(Algorithms)*** We investigate a variety of techniques, grouped in two main categories, for achieving balanced coloring. Algorithms in the first category aim to obtain a balanced coloring in a single attempt (*ab initio*). Those in the second category begin with an initial coloring oblivious to balance and use that information to produce a new coloring that is also balanced (*guided*). We propose three different types of guided balanced coloring algorithms (each of which could further be specialized to tune for performance) and examine several variants of *ab initio* approaches.
- ***(Parallelization)*** We parallelize all of the balanced coloring algorithms we explore targeting two different architectures: conventional multicore x86 architectures and a specialized many-core platform, Tilera TileGx36.
- ***(Application)*** We demonstrate the impact of balanced coloring on community detection — which is a widely used graph application. Our results show that using balanced coloring for this application could yield performance improvements while preserving quality.

The paper is organized as follows. We provide background and motivate our work in Sec. II. We describe the sequential versions of the various algorithms we explore for balanced coloring in Sec. III and discuss how they are parallelized in Sec. IV. We review essential features of the platforms for which the implementations are targeted in Sec. V. We present and discuss experimental results in Sec. VI. We conclude in Sec. VII.

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.113

IEEE computer society

II. Background

A. Basics of Equitable and Balanced Coloring

A coloring of a graph is an assignment of colors to vertices such that any two adjacent vertices get different colors. A coloring is said to be *equitable* if the sizes of any two color classes differ by at most one. The concept of equitable coloring was introduced by Meyer in a 1973 paper [1], but its history goes even further back to a conjecture by Erdös, a conjecture settled in 1970 by Hajnal and Szemerédi [2] forming their celebrated theorem: a graph with maximal degree Δ is equitably k-colorable if $k \geq \Delta+1$. This bound is sharp. The equitable coloring problem asks for an equitable k-coloring with the smallest possible k. The problem is NP-hard, as the classical coloring problem can be trivially reduced to it. Furmanczyk [3] provides a survey of work on equitable colorings until the early 2000's.

In equitable coloring, as stated earlier, the difference in size between any pair of color classes is required to be at most one. This ideal can for some practical needs be unnecessarily stringent and too costly to attain. In the closely related heuristic variant we refer to here as *balanced coloring*, the restriction is relaxed; the difference in color class size instead is allowed to be at most a "small" number greater than 1. One formal way to state this is to say that each color class is bounded by some parameter l. Bodleander and Fomin [5] study this problem and show that it, as well as the equitable coloring problem itself, can be solved in polynomial time for graphs with bounded treewidth.

Equitable coloring and balanced coloring find important applications in various areas. Examples include load-balanced partitioning for domain decomposition methods [6], parallel sparse matrix computations on irregular grids [7], and various types of scheduling and timetabling problems [8]. Tucker in a 1973 paper [9] discusses how equitable coloring theory has been used in helping out Operations Researchers at the Urban Science Department at Stony Brook, who were faced with a challenging routing problem that sought to optimize scheduling of garbage collecting trucks in the city.

Balanced coloring in the context of parallel scientific computing was studied by Gjertsen, Jones and Plassmann [10], where they developed a balanced parallel coloring heuristic building on their own earlier work on parallel graph coloring that was unconcerned with balancing color classes [11]. Their balanced coloring heuristic leverages a similarity with approximation algorithms for *bin packing* [12].

B. A Foundational Scheme

For the standard graph coloring problem, despite its NP-hardness, the greedy scheme outlined in Algorithm 1 is often found quite effective in practice, since the scheme gives usable solutions and can be implemented to run in linear-time for graphs that arise in practice.

Algorithm 1 Greedy

```
Greedy ($G = (V, E)$)
  for each $v \in V$ in some order do
      Assign $v$ a color not used by any of its neighbors
```

The scheme Greedy can be specialized in a variety of ways depending on a) the technique used to determine the order in which the vertices are processed and b) the strategy used to pick a color (among a set of permissible colors) for a vertex at a given step.

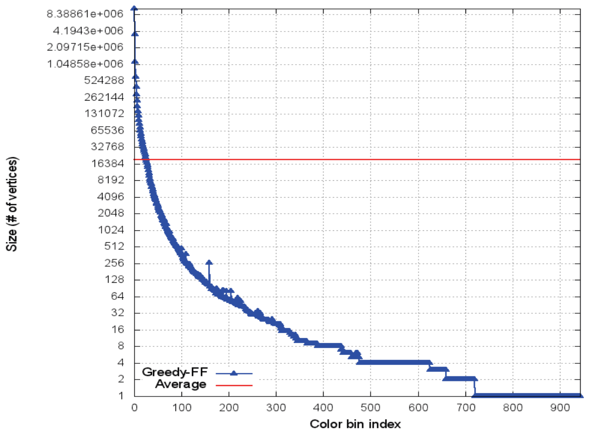
A common strategy used with regards to (b) is to pick the *smallest* (we assume colors are positive integers) permissible color for a vertex in each step. This strategy is sometimes referred to as First Fit (FF), since, considering the bin packing analogy mentioned earlier, it strives to place the vertex in the first bin (color) it could be placed in. The rationale behind choosing the smallest color is that one can then *guarantee* that the number of colors used by the scheme is bounded from above by $\Delta + 1$ (where Δ is the maximum degree in the graph) regardless of the order in which the vertices are processed and by $K + 1$ (where K is the *core number* of the graph) if the degeneracy order of the vertices is used. A degeneracy order, also known as Smallest Last ordering, can be obtained in linear-time.

The FF strategy is attractive for the bounds on the number of colors it assures. The color classes it produces, however, are highly skewed, with a vast majority containing significantly smaller number of vertices — an expected result out of selecting the first available bin for every vertex. The chart in Fig. 1a confirms this trend on a graph obtained by crawling an internet domain (uk-2002). Small-sized color classes can become scalability bottlenecks in an end-application, where typically the color classes are processed in different steps (to honor dependencies) and the smaller classes limit the degree of parallelism during those steps.

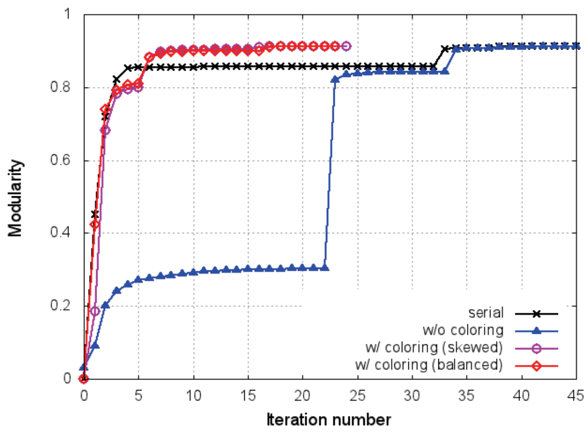
C. Motivating Application: Parallel Community Detection

Overcoming such scalability bottlenecks is in part what motivated our current work. We sought to investigate algorithms for achieving balanced coloring and their effective use in parallel computing applications. As a case-study we focus in this work on the use of balanced coloring in the context of a parallel community detection implementation, a suite called "*Grappolo*" that we developed for multi-core and manycore architectures [14], [13]. The parallel implementation is based on the sequential Louvain heuristic [15]. The Louvain method, which is one of the most widely used community detection algorithms, uses the modularity function [16] as the objective function to be maximized.

Grappolo consists of multiple phases, each in turn containing multiple iterations. Within each phase, the algorithm starts with every vertex placed in a community of its own. A series of iterations is then performed until a convergence criterion is met. Within each iteration, all vertices are scanned in parallel. For each vertex, a greedy decision is made as to whether the vertex should migrate to a different community (selected from one of its neighbors) or should



(a) Greedy Coloring, Input: uk-2002



(b) Community Detection, Input: cnr

Figure 1. a) The size distribution of the color classes obtained by the Greedy First Fit coloring heuristic for an input graph (*uk-2002*) obtained through a web crawl of the .uk domain. b) The evolution of modularity gain across the iterations of a parallel implementation of the Louvian method [13]. Four curves are depicted there. Two of the curves correspond to results obtained when coloring (skewed and balanced) is used in the parallel implementation, the third corresponds to results when coloring is *not* used, and the fourth corresponds to results on a serial implementation.

stay in its current community, so as to maximize the net modularity gain. This approach places multiple constraints on concurrent processing of neighboring vertices. In previous work, we had extensively explored the use of graph coloring in effectively addressing the challenges associated with these constraints [13]. Our findings showed that the use of coloring significantly accelerates convergence and, for many input cases, also improves the quality of communities output (as measured by the modularity function). However, since the color classes are processed in parallel one at a time, large skews in color class sizes reduced overall scalability, particularly for inputs such as the uk-2002 (see Fig. 1). Balanced coloring could potentially provide a significant performance boost here while preserving the quality of output (since we still respect the partial ordering of vertices imposed by coloring). The chart in Fig. 1b demonstrates the impact of coloring on Grappolo's output quality (modularity gain) and performance (convergence).

III. Algorithms for Balanced Coloring

In this section, we present multiple heuristics to compute a balanced coloring of an input graph. We explore two categories of approaches. Approaches in the first category aim at obtaining a balanced coloring in a single attempt. We refer to these as "*ab initio*" approaches. Those in the second category follow a two-step procedure, where an initial coloring obtained using a balance-oblivious procedure, is subsequently balanced in the second step. Since in all the approaches in this second category the information produced by the initial coloring is used to guide the respective balancing strategies, we refer to them as "*guided*" approaches.

A. Ab initio strategies

Within the *ab initio* category, we consider two well-known variants of the Greedy scheme outlined in Algorithm 1 that differ in how the choice of color to be assigned to a vertex in each step is done. In the first variant (Greedy-LU) a vertex is assigned the Least Used (LU) permissible color (not used by any neighbor) among all currently available colors. If no permissible color exists, then a new color is created and assigned to the vertex. In the second variant (Greedy-Random) a vertex is assigned a color picked at random from the set of permissible colors. The particular Greedy-Random variant we consider assumes the existence of a reasonable bound B on the number of colors needed. One such easy-to-compute bound is $B = \Delta + 1$. Then, a vertex v is assigned a randomly chosen color from the set of permissible colors $P(v) \subseteq \{1, 2, \ldots, B\}$. Manne and Boman analyze balanced greedy coloring using the strategies LU and Random in the context of sparse random graphs [17].

B. Guided strategies

In the guided category, we study different approaches for obtaining a balanced coloring given an initial coloring. We note here that *all* of the proposed guided approaches can be applied to an initial coloring produced by an arbitrary coloring method. However, a subset of these approaches is designed to exploit certain properties of an initial coloring produced by the Greedy coloring scheme that uses the FF color choice strategy (henceforth abbreviated as Greedy-FF).

Given an input graph $G = (V, E)$, let the number of colors used by the initial coloring step be C. In all our guided strategies, we make use of the quantity $\gamma = |V|/C$ to guide our methods. Note that in a strictly balanced setting, the size of each color class would be roughly γ. Consequently, we call the color classes (bins) whose sizes are greater than γ *over-full* and those whose sizes are less than γ *under-full*. (We use the terms bin and color class interchangeably throughout the paper.)

Broadly, we can classify our guided strategies into two

Table I

A COMPREHENSIVE LIST OF BALANCING STRATEGIES PRESENTED AND STUDIED IN THIS PAPER. THE INPUT GRAPH IS DENOTED BY $G = (V, E)$.

Strategy	**Category**	**Description**
Greedy-LU	*ab initio*	Run Algorithm 1 with LU color choice among permissible colors.
Greedy-Random	*ab initio*	Run Algorithm 1 with Random color choice among permissible colors.
Shuffling-unscheduled	guided	Run Algorithm 1 with FF color choice strategy. Based on the obtained coloring identify over-full and under-full bins. Move select vertices from over-full to under-full bins without changing the number of color classes. Further specializations include Vertex-centric FF (*VFF*) and Color-centric LU (*CLU*).
Shuffling-scheduled	guided	Run Algorithm 1 with FF color choice strategy. Based on the obtained coloring identify over-full and under-full bins. Move select vertices from over-full to under-full bins in a scheduled manner without changing the number of color classes.
Recoloring	guided	Run Algorithm 1 with FF color choice strategy. Let the number of colors used be C. Let $\gamma = \|V\|/C$. Construct an ordered set of vertices $W = \{V(C), V(C-1), \ldots, V(1)\}$, where $V(i)$ denotes the set of vertices having the color i. Re-color vertices in W in that order using Algorithm 1 such that in each step, a vertex v is assigned the smallest permissible color k such that the size of bin k is less than γ.

types. In the first type, a subset of vertices from each over-full bin is moved to under-full bins so that a better balance is attained. Since this is achieved *without* increasing the number of color classes, we refer to this type of methods *Shuffling*-based. In the second type, *all* vertices are colored afresh (moved to new bins), this time with a balance constraint imposed. We call this *Recoloring*.

The Shuffling methods in turn comprise two specializations: *unscheduled* and *scheduled* moves. The motivation for this distinction comes from parallel performance needs that will be explained in Section IV.

The Recoloring method takes advantage of an interesting property of the Greedy-FF scheme. Suppose a coloring of a graph $G = (V, E)$ is obtained using Greedy-FF in some vertex order. Let the number of colors used be C. Now suppose the vertices of G are ordered such that vertices in the same color class are listed consecutively. Then re-applying Greedy-FF using this new ordering will produce a new coloring of G using C or fewer colors. Culberson [18] applied this idea iteratively in his method called Iterated Greedy (IG) to successively reduce the number of colors and draw the number as close to the optimal as possible. There is a degree of freedom in how the color classes themselves could be ordered for IG to be successful. One of the better strategies is to list the color classes in reverse order — i.e., beginning from the vertices of the highest color index.

We build on this property to devise our Recoloring method for balancing. A key extension in our case is that we maintain the sizes of bins during the new coloring and use those to impose balance. In particular, in each step of the re-coloring, a vertex is assigned the smallest permissible color k such that the size of the bin k is less than γ.

Table I summarizes the various balancing methods we discussed in this section.

IV. PARALLEL ALGORITHMS

We parallelized all of the guided balanced coloring algorithms presented in Section III for the shared memory model. For each heuristic we developed two OpenMP-based implementations — one for conventional multicores and another for the Tilera manycore platform. To obtain the initial coloring we used a parallel implementation available for Greedy-FF from a previous effort [19]. In this section, we describe the parallel algorithms underlying the implementations of the balancing schemes.

To parallelize our shuffling-based approaches, we considered two ways of moving a vertex from an over-full bin to an under-full bin. The first type of move is "*unscheduled*". Here, the choice of the target bin for a given vertex is decided dynamically (using either the FF or LU strategy) based on the state of the color bins — encompassing both size and composition. This approach strives to achieve a good balance, if possible; as a trade-off, however, it entails the cost needed to keep each dynamic state up-to-date. More specifically, concurrent updates to the sizes of the same bin need to be synchronized.

To mitigate the cost of updates, we explored an alternative we call "*scheduled*" moves, where the target bin for a vertex in an over-full bin is statically determined using a heuristic, and the check to verify if such a move is permissible is deferred until the move is actually attempted. If a move attempt creates a "conflict", which is possible if a neighboring vertex is already in the same target bin, no further attempt is made and the vertex remains in its original bin. The advantage of this approach is the expected improvement in parallel performance, as no atomic operation or lock is needed to update bin sizes. However, this approach could potentially terminate without achieving a balance.

A. Parallelization using Unscheduled Moves

For obtaining guided balanced coloring using unscheduled moves, we considered two parallelization schemes. In the *vertex-centric* schemes, the loop-parallelization is around a set of vertices, and vertices from different color classes are allowed to be processed concurrently. In the *color-centric* schemes, vertices processed concurrently must belong to the same color class. In both schemes, only vertices in over-full bins are considered for color reassignment. Furthermore, once an over-full bin i reaches balance (i.e., $|V(i)| = \gamma$) at a certain point in the execution, then vertices from that bin are no longer considered for color reassignment. Hence, these schemes represent partial recoloring methods that proceed until either a balance is achieved or a balance is no longer possible (i.e., there exist no more permissible moves from any of the remaining over-full bins).

Vertex-centric parallelization scheme: Processing vertices from possibly different color classes exposes maximum concurrency. However, it could also cause conflicts. To handle such conflicts in parallel we adopt the *Speculation-and-Iteration* framework described in [19]. The basic idea in this framework is to maximize concurrency by temporarily tolerating inconsistencies. Consider a simple loop-parallelization over the set of vertices in the Greedy scheme (using FF or LU) outlined in Algorithm 1. Such a parallelization will not preclude the possibility of a pair of adjacent vertices from being colored at the same time and receiving the same color. In our adoption of the speculation-and-iteration framework, once vertices are moved to their target color classes, the idea is to detect conflicts (in parallel) in a separate phase in the same round and resolve them in a subsequent round. The algorithm proceeds iteratively in this fashion until all conflicts are resolved.

A template for the vertex-centric parallelization scheme is presented in Algorithm 2. This algorithm corresponds to the Vertex-centric First Fit (*VFF*) balancing method. It should be easy to see that the same algorithm can be easily adapted to the Vertex-centric Least Used (*VLU*) balancing method with a change to the target bin (k) selection criterion.

Algorithm 2 Vertex-centric parallel scheme for balanced coloring (using FF)

```
VertexParallelGuidedBalancing($G = (V, E)$)
  Obtain an initial coloring of $G$
  Let $U$ be the set of vertices from over-full bins
  while $U \neq 0$ do
      for $v \in U$ do in parallel
          Let $k$ be the smallest index of an under-full bin
              that is permissible                          ▷ FF
          if $k$ exists then
              $color[v] \leftarrow k$
              Update size of bin $k$          ▷ synchronized step
      $R \leftarrow \emptyset$
      for $v \in U$ do in parallel            ▷ check for conflicts
          for $w \in adj(v)$ do
              if ($color[w] = color[v]$ and $v > w$) then
                  $R \leftarrow R \cup \{v\}$
      $U \leftarrow R$
```

Note that the maximum number of conflicts per a vertex v in the above algorithm can be upper-bounded by $\min\{d(v), b\}$, where $d(v)$ is the number of vertices adjacent to v and b is the number of under-full bins. This upper-bound is rather weak. In practice, we observed that the closely related quantity – the actual number of iterations needed to clear all conflicts – is typically a small constant.

Color-centric parallelization scheme: In the color-centric scheme for parallelization, we allow only vertices from the same color class to be processed concurrently. This is achieved by processing one over-full color class at a time and performing the moves departing from that over-full bin in parallel until a balance is achieved or no more move is possible. This scheme, therefore, avoids conflicts, and the balancing procedure requires no more than a single pass of the over-full bins. However, the trade-off is in parallel performance of the balancing procedure, which requires as many parallel steps as there are number of over-full bins in the initial coloring. A template for the color-based parallelization scheme is shown in Algorithm 3.

Algorithm 3 Color-centric parallel balanced coloring

```
ColorParallelGuidedBalancing($G = (V, E)$)
  Obtain an initial coloring of $G$
  Let $Q$ be the set of over-full colors (bins)
  for each $j \in Q$ do
      Let $V(j)$ denote the set of vertices with color $j$
      for $v \in V(j)$ do in parallel
          Let $k$ be the smallest index of an under-full bin
              that is permissible to $v$                   ▷ FF
          if $k$ exists then
              $color[v] \leftarrow k$
              Update size of bin $k$          ▷ synchronized step
```

Initial coloring: We note here a special property emerging from the use of Greedy-FF for generating the initial coloring. Any initial coloring produced by Greedy-FF satisfies the following property: Assume a linear ordering of colors from $1, \ldots, C$. If a vertex v is assigned color j, where $j > 1$, then it implies that v contains at least one neighbor in each of the previous colors $1, \ldots, j-1$ (otherwise, v would have been assigned a smaller color). Therefore, if we follow the Greedy-FF initial coloring by another FF-based strategy during the subsequent *balancing* step (e.g., VFF or CFF), then the closest permissible bin, say k, we identify through that procedure would also correspond to a color that has a high incidence of edges on the source over-full color bin. Given that k represents a permissible bin despite its high incidence makes it intuitively an attractive target for this vertex. On the other hand, an LU-based strategy (VLU or CLU) operates oblivious to the ordering of the initial colors, and is therefore better suited for scenarios where the initial coloring was generated by schemes *other than* Greedy-FF.

It is for these reasons that we use the Greedy-FF strategy for initial coloring for VFF and CFF, while for VLU and CLU the use of any other initial coloring scheme is allowed.

B. Parallelization using Scheduled Moves

To parallelize guided balancing using scheduled moves we take advantage of both the incidence property (observed above) and another size-related property of the Greedy-FF initial coloring: owing to its First Fit strategy, Greedy-FF is expected (but not guaranteed) to assign more vertices to smaller-indexed color classes. In other words, color classes are expected (but not guaranteed) to be in non-increasing order of their sizes as one proceeds from color 1 through color C. This expectation agrees with the size distributions depicted in Fig. 1.

Our parallel algorithm with scheduled moves is outlined in Algorithm 4. Intuitively, we identify an arbitrary subset of surplus vertices from the sequence of over-full bins and mark each of them for assignment to a corresponding under-full color[1]. At this point, no explicit checks are made to identify conflicts. In the next step, all vertices from the over-full bins that were scheduled for recoloring are processed in parallel to check if any of them conflicts with the assigned target bin. A move is completed only if it generates no conflicts.

Algorithm 4 Parallel shuffling using scheduled moves

```
ScheduledBalancing($G = (V, E)$)
  Obtain an initial coloring of $G$ using Greedy-FF
  Let $Q_O$ be an ordered set of over-full colors in increasing
      order of color index
  Let $Q_U$ be an ordered set of under-full colors in decreasing
      order of color index
  Let $L$ (initially $\emptyset$) maintain a list of moves from over-full
      to under-full bins
  for each $j \in Q_O$ do
    Let $V(j)$ denote $\{u \in V \mid color[u] = j\}$
    Select $V'(j) \subseteq V(j)$ such that $|V'(j)| = |V(j)| - \gamma$
    for each $k \in Q_U$ AND $V'(j) \neq \emptyset$ do
      Let $V'_k(j) \subseteq V'(j)$ denote vertices that can be
          moved to $k$ such that $|V'_k(j)| + |V(k)| \leq \gamma$
      $L \leftarrow L \cup V'_k(j)$
      $V'(j) \leftarrow V'(j) \setminus V'_k(j)$
  for $V'_k(j) \in L$ do
    for $v \in V'_k(j)$ do in parallel
      if ($k$ is a permissible color for $v$) then
        $color[v] \leftarrow k$
```

This simple approach requires no synchronization on the bin sizes. However it could leave the bins imbalanced. To improve the chance of obtaining a better balance, we fill the under-full bins (set Q_U in Algorithm 4) in the *decreasing* order of color index (we refer to this approach as *Scheduled Reverse*, or more simply, *Sched-Rev*). Attempting to fill the under-full bins in this decreasing order tends to preserve, to the extent possible, color co-assignment of vertices — i.e., two vertices being moved from the same source over-full bin are likely to co-locate in the same target under-full bin, thus minimizing the chance of conflicts. This is a consequence of the aforementioned size-related property of the Greedy-FF initial coloring. In contrast, if the under-full bins were to be filled in the increasing order of color index, then the incidence property of Greedy-FF is likely to introduce more conflicts, preventing more vertices in the over-full bins from moving.

[1]This step is performed serially in our current implementation since it was very quick for most inputs; however, if required, this step can also be parallelized using parallel prefix (details omitted).

C. Parallel Recoloring

The parallelization we use for the balancing based on Recoloring is outlined in Algorithm 5. This is similar to the vertex-centric parallel scheme given in Algorithm 2 with the main difference being that we recolor *all* the vertices from scratch and that balance is *imposed* as the recoloring proceeds. The rationale for ordering the color classes in W in reverse order of their introduction in the initial coloring is that, as mentioned in section III-B, the re-coloring would then be highly likely to use fewer colors, since now the vertices that were "difficult" to color initially are processed earlier.

Algorithm 5 Parallel Recoloring for Balance

```
ParallelRecoloring($G = (V, E)$)
  Obtain an initial coloring of $G$ using Greedy-FF
  Let $C$ be the number of colors used, and let $\gamma = |V|/C$
  Let $V(j)$ denote $\{u \in V \mid color[u] = j\}$
  Construct ordered set $W = \{V(C), V(C-1), \ldots, V(1)\}$
  Initialize $bin[i] = 0$, for $i = 1, \ldots, C$
  $U \leftarrow W$
  while $U \neq \emptyset$ do                ▷ perform a fresh coloring
    for $v \in U$ do in parallel
      $color[v] \leftarrow$ smallest permissible color $k$ such that
          $bin[k] < \gamma$
      Increment $bin[k]$ by 1               ▷ synchronized step
    $R \leftarrow \emptyset$
    for $v \in U$ do in parallel
      for $w \in adj(v)$ do
        if ($color[w] = color[v]$ and $v > w$) then
          $R \leftarrow R \cup \{v\}$
    $U \leftarrow R$
```

D. Complexity

With careful choice of data structures, the sequential Greedy scheme (Algorithm 1) that underlies all of our parallel algorithms, can be implemented such that its runtime is upper-bounded by $O(|V| \cdot \Delta)$, where Δ is the maximum degree in the graph. In each of the templates outlined in Algorithms 2 through 5, the total "additional" work incurred due to parallelization is no more than the work involved in Algorithm 1. Furthermore, the number of rounds required by the iterative variants (Algorithm 2 and Algorithm 5), as argued earlier, is typically a small constant in practice. Therefore, we conclude that the net parallel work in any of our schemes can be upper-bounded by $O(|V| \cdot \Delta)$.

V. Implementation on the Tilera Platform

We have ported our parallel balanced coloring algorithms to the Tilera manycore platform. The Tilera TileGX36 system implements a manycore processor based on a two-dimensional mesh topology. Each core (called a "tile" in Tilera's terminology), consists of a 3-way VLIW processing unit, a private 32KB, 2-way set associative L1 data cache, a private 32KB, direct-mapped instruction cache and a 256KB, 8-way set associative unified L2 cache. The cache line granularity is 64 bytes across all three caches. Each tile is connected via multiple links to several networks-on-chip (NOCs) in a 2D mesh configuration[2].

Tilera's caching policies are the salient features that we exploit to optimize this application. For each individual memory page, the system can set the *home* tile of its data in the cache subsystem. There are two principal modes for setting the home tile of a memory page: **homed** (a particular tile is the home for the whole page) and **hashed** (individual cache lines on the page are distributed in a round-robin manner to the L2 caches of all tile).

For the balanced coloring algorithms and the community detection application we use a heap manager with a backing store of homed huge pages (16 MB/page) for all thread private data. The global shared data structures (Compressed Sparse Row Representation of the graph, arrays of colors and bin sizes) are allocated on default-sized pages (64 KB/page) using the hashed policy. Previous experience with basic coloring and community detection on Tilera [14] has shown this configuration to be the most performant one for all input data sets. The OpenMP threads created by the application are pinned to contiguous sets of cores on the manycore mesh architecture in order to avoid costly thread migration and subsequent cache misses.

VI. Experimental Results

A. Experimental Setup

Test Platforms: We used two platforms for testing: the Tilera TileGX36 presented in Section V and an Intel Xeon X7560 platform. The TileGX36 platform is equipped with 32 GB of DDR3 memory separated into two 16 GB banks, with the cores running at 1.2 GHz[3]. The community detection code has been parallelized using OpenMP and Tilera-specific extensions for memory management, synchronization and atomic operations.

The Intel multicore platform has four sockets and 256 GB of memory. Each socket is equipped with eight cores running at 2.266 GHz, leading to a total of 32 cores. The system is equipped with 32 KB of L1 and 256 KB L2 caches per core, and 24 MB of L3 cache per socket. Each socket has 64 GB of DDR3 memory with a peak bandwidth of 34.1 GB per second[4].

Test inputs: The test inputs used in different experiments are summarized in Table II. All inputs were downloaded from the University of Florida sparse matrix collection [20], with the exception of MG2 which is a custom-built biological network. These inputs were chosen to encompass a variety in graph sizes and color class properties such as the number of colors and color size distribution (Table III).

Table II
Statistics on structure of the real-world networks used in our study.

Input graph	Num. vertices (n)	Num. edges (m)	Degree stats max.	avg.
CNR	325,557	2,738,970	18,236	16.28
coPapersDBLP	540,486	15,245,729	3,299	56.41
Channel	4,802,000	42,681,372	18	17.77
MG2	11,005,829	674,142,381	5,466	122.50
uk-2002	18,520,486	261,787,258	194,955	28.27
Europe-osm	50,912,018	54,054,660	13	2.12

B. Balance Quality Assessment

In this section, we compare the quality of balance in the color class sizes produced by the different balancing schemes proposed in the paper. (Please refer to Table I for an overview of all the schemes.) To measure balance, we use the Relative Standard Deviation of the color class sizes (expressed in %), which is the ratio of the standard deviation to the average color size. The closer this value is to 0.00% the better is the balance. For the schemes {Recoloring, Greedy-LU and Greedy-Random} we also compared the number of colors they produce to the number of colors produced by the Greedy-FF scheme (initial coloring).

Table III shows the results of our quality assessment. First, we observe the very large skews in the color sizes produced by the Greedy-FF scheme (which was the primary motivation behind this work). With respect to balancing, we observe that schemes VFF and CLU generally outperform all other schemes in either category (guided or *ab initio*). We note here that if the initial coloring was generated by a scheme other than Greedy-FF, then CLU is expected to outperform VFF. The Sched-Rev scheme was also effective in reducing the skew although the degree of balance achieved was lower than VFF and CLU - as can be expected due to its scheduled strategy. One way to improve the performance of the scheduled strategy is to iterate the procedure a constant number of times; however the tradeoff is that it would increase run-time.

Among the schemes that do not guarantee the same number of colors as Greedy-FF (viz. Recoloring, Greedy-LU and Greedy-Random), we observed consistently that all those three schemes produced more colors than the Greedy-FF scheme. However, the number of colors produced by Recoloring was generally close to the number of colors produced by Greedy-FF and other guided schemes (VFF, CLU), and the balancing obtained was comparable to the Sched-Rev scheme. On the other hand, Greedy-LU and Greedy-Random produced significantly higher number of colors making them less desirable from the end-application perspective. As described in Section IV-C, the main advantage of the Recoloring scheme is that it processes the vertices with larger color indices earlier. Since these vertices have higher degree and consequently harder to color, there is

[2]These NOCs include one for coherence traffic, a user-programmable message passing NOC, and a dedicated I/O NOC

[3]The TileGX36 runs a custom version of Linux adapted for Tilera's hardware. The compiler and runtime environment are adapted from GCC 4.8.2 and retargeted for the TileGX's 64-bit VLIW cores.

[4]We use GCC 4.8.2 to compile an x86 version of the code that uses OpenMP for its parallelization. We use `numactl` to pin OpenMP threads to cores.

Table III
QUALITY OF BALANCE OBTAINED BY THE DIFFERENT HEURISTICS ON DIFFERENT INPUTS. ENTRIES IN EACH CELL SHOW THE RELATIVE STANDARD DEVIATION (IN %) OF COLOR CLASS SIZES OBTAINED BY A GIVEN HEURISTIC (THE LOWER THE VALUES, THE BETTER THE BALANCE). THE GUIDED SCHEMES VFF AND CLU PRODUCE THE SAME NUMBER OF COLORS AS THE INITIAL COLORING SCHEME (GREEDY-FF). THE NUMBER OF COLORS PRODUCED BY THE OTHER HEURISTICS IS PROVIDED IN PARANTHESIS (NEXT TO THEIR RESPECTIVE RSD VALUES).

Input graph	**Init. coloring** Greedy-FF		**Guided schemes** VFF	CLU	Sched-Rev	Recoloring		***Ab initio* schemes** Greedy-LU		Greedy-Random	
CNR	587.73%	(85)	**0.03%**	0.04%	16.44%	13.81%	(88)	12.03%	(211)	24.29%	(209)
coPapersDBLP	342.41%	(336)	0.69%	**0.15%**	12.01%	10.17%	(340)	**0.11%**	(337)	23.15%	(338)
Channel	128.99%	(12)	**0.00%**	7.16%	7.55%	35.75%	(14)	4.84%	(16)	20.05%	(17)
MG2	1272.31%	(2,143)	0.38%	**0.21%**	9.57%	25.34%	(2335)	18.09%	(2169)	94.37%	(2172)
uk-2002	1885.15%	(943)	0.08%	**0.01%**	4.88%	3.53%	(945)	2.94%	(1010)	28.34%	(1018)
Europe-osm	126.87%	(5)	**0.00%**	**0.00%**	6.70%	39.90%	(6)	0.00%	(7)	12.07%	(6)

potential benefit in processing them earlier in the Recoloring scheme. However, the balancing constraint imposed during the recoloring process coupled with parallel execution which disturbs the intended order of vertex processing explains the less-than-optimal performance displayed by this scheme.

For an illustration of the effect of the different balancing schemes, refer to Fig. 2, which shows the sizes of all the color classes produced by the different balancing schemes.

C. Performance Evaluation

The balancing schemes were also compared against one another for their parallel performance. We tested both our Tilera and x86 implementations on a range of inputs and thread counts. Tables IV and V show the run-times taken by the VFF balancing scheme[5]. The corresponding speedup charts are shown in Fig. 3.

The results show that the scaling in Tilera manycore is significantly superior to the scaling results in x86. For instance, a top speedup of 13× was observed on 16 Tilera cores. The improved scalability delivered by the Tilera manycore platform can be largely attributed to a scalable on-chip network interconnect, which reduces the costs of synchronization and latency for irregular memory accesses. On the other hand, we found synchronization overhead to be a significant factor impacting the parallel performance on the x86 architecture. We confirmed this by comparing the run-times between the VFF (that uses atomic operations to update bin sizes) and Sched-Rev (that does *not*). On the x86 architecture, we consistently observed Sched-Rev to be 8× or more faster than VFF on all inputs tested. The corresponding performance gain on the Tilera platform was a more modest 2×.

The speedup trends observed on both architectures across the three inputs also show the impact of the number of initial colors on parallel performance. In both speedup charts, MG2 (2K colors) is the best-performing, followed by uk-2002 (943 colors) and then by Channel (12 colors). Intuitively, fewer colors imply a higher probability for concurrent bin size updates.

With respect to absolute times, on a per-core basis the Tilera platform is much slower than the x86 system. The main reason is the relatively modest frequency and instruction level parallelism (ILP) of the Tilera cores (3 packed operations per VLIW instruction, statically scheduled by the compiler), in comparison to double the frequency on the x86 system and very aggresive & wide superscalar instruction scheduling. However, at full system scale the improved scalability of the Tilera platform makes up the difference with respect to x86 and even surpasses its absolute performance for some inputs.

Table IV
PARALLEL RUN-TIME (IN SECONDS) OF THE VFF SCHEME ON DIFFERENT NUMBER OF CORES OF THE TILERA PLATFORM. TIMES SHOWN ARE ONLY FOR THE BALANCING PROCEDURE (I.E., INITIAL COLORING TIME IS *not* INCLUDED).

Input graph	Number of threads 1	2	4	8	16	32	36
Channel	7.55	4.57	3.37	2.59	2.23	**2.06**	2.13
uk-2002	163.22	84.68	45.59	26.32	16.87	12.11	**11.66**
MG2	460.22	254.66	154.16	85.97	54.56	34.95	**33.29**

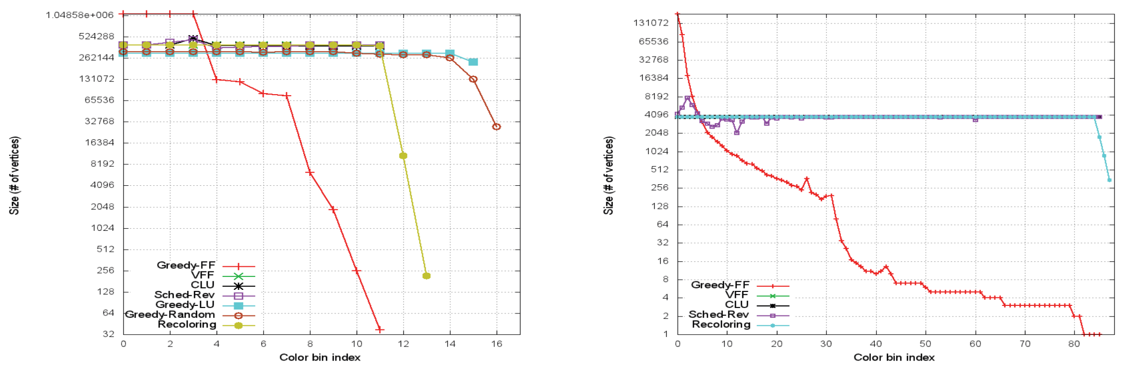
Table V
PARALLEL RUN-TIMES (IN SECONDS) OF THE VFF SCHEME ON DIFFERENT NUMBER OF CORES OF THE INTEL X86 PLATFORM. TIMES SHOWN ARE ONLY FOR THE BALANCING PROCEDURE (I.E., INITIAL COLORING TIME IS *not* INCLUDED).

Input graph	Number of threads 2	4	8	16	32
Channel	**0.45**	2.51	2.54	2.01	2.67
uk-2002	21.05	**15.71**	18.07	21.01	23.57
MG2	46.92	23.15	14.4	14.98	**10.08**

In Table VI, we compare the run-times of three of the most competitive balancing schemes {VFF, Sched-Rev and Recoloring} on the Tilera manycore platform. As expected the Sched-Rev scheme outperforms the other two schemes. More specifically, we observed Sched-Rev to be ∼2× faster than VFF[6]. Considering the fact that Sched-Rev also performed appreciably well in terms of balance quality (Section VI-B), we conclude that it provides the best trade-off between quality and performance among all the balancing schemes presented in this paper.

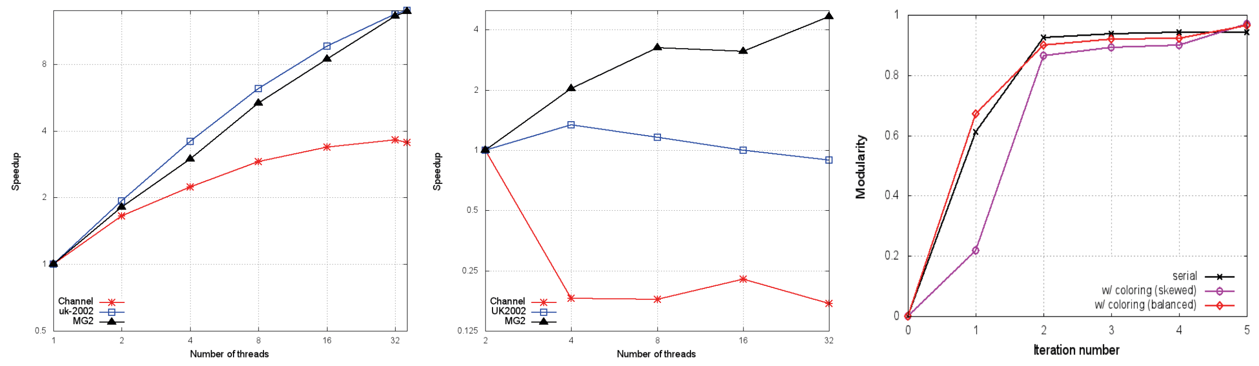
[5]We select VFF because it was one of the schemes that produced the best balancing results (as was discussed in SectionVI-B).

[6]This performance improvement was even more pronounced in x86 architecture as noted earlier.



(a) Channel (b) CNR

Figure 2. Distribution of color class sizes produced by the different balanced coloring schemes (horizontal axis corresponds to colors (bins) and vertical axis to sizes of color classes). Recall that smaller color class sizes correspond to reduced parallelism in the end-application, while higher number of colors corresponds to increased number of parallel steps within the application. For Channel, color class sizes from all balancing schemes are shown. For CNR, color class sizes from only the balancing schemes that produce less number of colors are shown.



(a) Speedup on Tilera (b) Speedup on x86 (c) Community detection

Figure 3. (a, b) Speedup obtained by our Tilera manycore and x86 multicore implementations of the VFF balancing scheme. Speedups are relative to one (Tilera) or two (x86) thread execution. (c) Application study: Evolution of modularity values within the first phase of a parallel community detection implementation (Grappolo) on uk-2002, performed with the use of VFF balanced coloring. The chart also shows the corresponding modularity curves for the runs made *without balanced* coloring and the best performing serial implementation [15].

Table VII
EVALUATION OF THE BALANCING HEURISTICS ON A PARALLEL COMMUNITY DETECTION APPLICATION, *Grappolo*. ALL TIMING RESULTS ARE IN SECONDS AND WERE OBTAINED ON 36 THREADS OF THE TILERA MANYCORE PLATFORM.

Input graph	**w/o balanced coloring**			**w/ balanced coloring**			
	Run-time		Modularity	Run-time			Modularity
	Init. coloring	Community detection		Init. coloring	*VFF balancing*	Community detection	
CNR	0.15	3.98	0.9124	0.15	*0.15*	4.16	0.9119
Channel	1.87	38.85	0.9348	1.87	*2.13*	**20.94**	0.9328
MG2	37.31	954.81	0.9984	37.31	*33.29*	**483.80**	0.9984
uk-2002	7.83	406.81	0.9895	7.83	*11.66*	**254.27**	0.9894
Europe-osm	17.95	358.26	0.9988	17.95	*20.98*	369.19	0.9988

Table VI
PARALLEL RUN-TIMES (IN SECONDS) OF THE THREE BALANCING SCHEMES {VFF, SCHED-REV AND RECOLORING} ON 16 TILERA CORES.

Input graph	VFF	Sched-Rev	Recoloring
Channel	2.23	**2.19**	3.28
uk-2002	16.87	**8.71**	36.97
MG2	54.56	**27.70**	185.19

D. Impact on the Community Detection Application

To evaluate the effectiveness of the proposed balanced coloring schemes in a real world application, we studied the parallel community detection code, Grappolo, described in Section II-C. Since VFF was one of the leading schemes for balance quality, we used VFF as our default balancing scheme on the Tilera platform. We ran Grappolo in two modes: i) using the original skewed coloring, and ii) using the balanced coloring produced by VFF.

Table VII shows the results of our evaluation in the context of community detection using Grappolo. In this table, we compare both end-to-end performance (run-time) and output quality (modularity). We note here that the our current implementation of Grappolo is configured to use coloring only during the first phase of its algorithm. However, the algorithm itself is multi-phase and configuring to use coloring in subsequent phases is one of our planned future extensions. However, for this paper, we used coloring only for the first phase, and therefore, the benefits of balanced coloring observed in Table VII are understated.

From the table, we can observe the following: The overhead introduced in balancing is compensated by the run-time gains achieved in the community detection. This is true for three of the five inputs tested — for instance, in the case of MG2, balancing yields a total end-to-end run-time savings of **44.11%**. Note that this is for a single execution of the community detection code. In practice, a user may run multiple instances of community detection under different parametric settings (while the coloring is a one-time preprocessing task). The CNR input is the smallest in the number of vertices and edges that we processed and the gains from parallelism (w/ or w/o balancing) is insignificant. As for Europe-osm, the first phase only consumed 6% of the total run-time and therefore the benefits of balanced coloring are not directly evident from Table VII.

The results in Table VII also demonstrate the ability of the VFF balanced scheme to preserve quality of output (in terms of modularity). In fact, we observed that introducing balancing has a positive impact on the progression of modularity in the first phase, as illustrated in Fig. 3(c) — which is a consequence of the revised ordering of vertices due to balanced coloring.

VII. Conclusions

In this paper, we provided a thorough treatment of the problem of balanced coloring, with our contributions spanning algorithm development, parallelization, and application. Specifically, we presented multiple balancing schemes, developed parallel implementations on conventional multicores and an emerging manycore platform (Tilera), and evaluated their effectiveness in achieving a balanced coloring and how such results translate to gains in an application's performance using community detection as a motivating case-study. Coloring is used in a number of parallel computing applications to identify independent tasks, and we expect the detailed study presented in this paper involving a family of balancing algorithms and their implementations on emerging architectures to serve as a valuable reference to application developers who seek to improve parallel performance of their applications using coloring.

Acknowledgements

This research was support in parts by U.S. Department of Energy grants DE-SC-0006516, DE-AC05-76RL01830, and DE-SC-0010205, and the U.S. Department of Defense under the Autotuning for Power, Energy & Resilience (ATPER) project.

References

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

[2] A. Hajnal and E. Szemerédi, *Proof of a conjecture of P. Erdös*. London: North-Holland, 1970, pp. 601–623.

[3] H. Furmańczyk, *Equitable coloring of graphs*. Providence, Rhode Island: American Mathematical Society, 2004, pp. 35–53.

[4] M. Kubale, Ed., *Graph Colorings*. Providence, Rhode Island: American Mathematical Society, 2004.

[5] H. Bodleander and F. Fomin, "Equitable colorings of bounded treewidth graphs," *Theoret. Comput. Sci.*, vol. 349, no. 1, pp. 22–30, 2005.

[6] B. Smith, P. Bjørstad, and W. Gropp, *Domain Decomposition; Parallel multilevel methods for elliptic Partial Differential Equations*. Cambiridge: Cambridge University Press, 1996.

[7] R. Melhem and V. Ramarao, "Multicolor reorderings of sparse matrices resulting from irregular grids," *ACM Transaction of Mathematical Software*, vol. 14, pp. 117–138, 1988.

[8] J. Blazewick, K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz, *Scheduling computer and manufacturing processes*. Berlin: Springer, 2001, 2nd edition.

[9] A. Tucker, "Perfect graphs and an application to optimizing municipal services," *SIAM Review*, vol. 15, pp. 585–590, 1973.

[10] J. Robert K. Gjertsen, M. T. Jones, and P. Plassmann, "Parallel heuristics for improved, balanced graph colorings," *Journal of Parallel and Distributed Computing*, vol. 37, pp. 171–186, 1996.

[11] M. T. Jones and P. Plassmann, "A parallel graph coloring heuristic," *SIAM Journal of Scientific Computing*, vol. 14, pp. 654–669, 1993.

[12] J. E.G. Coffman, M. Garey, and D. Johnson, *Approximation Algorithms for Bin Packing: A Survey*. PWS Publishing Company, 1997, pp. 46–86.

[13] H. Lu, M. Halappanavar, and A. Kalyanaraman, "Parallel heuristics for scalable community detection," *arXiv preprint arXiv:1410.1237*, 2014.

[14] D. Chavarría-Miranda, M. Halappanavar, and A. Kalyanaraman, "Scaling graph community detection on the tilera many-core architecture," in *HiPC 2014*, Goa, India, Dec. 2014, p. In Press., 00000.

[15] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, p. P10008, 2008, 02101.

[16] M. E. Newman, "Fast algorithm for detecting community structure in networks," *Physical review E*, vol. 69, no. 6, pp. 66–133, 2004, 02459.

[17] F. Manne and E. Boman, "Balanced greedy colorings of sparse random graphs," in *The Norwegian Informatics Conference, NIK'2005*, 2005, pp. 113–124.

[18] J. Culberson and F. Luo, "Exploring the k-colorable landscape with iterated greedy," in *Cliques, coloring and satisfiability: Second DIMACS implementation challenge*, D. Johnson and M. Trick, Eds. American Math. Society, 1996, pp. 245–284.

[19] U. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multi-core and massively multithreaded architectures," *Parallel Computing*, vol. 38, pp. 576–594, 2012, 00009.

[20] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, Dec. 2011, 01043.