SCALABLE PARALLEL ALGORITHMS AND IMPLEMENTATIONS FOR LARGE-SCALE GRAPH ANALYSES

By

HAO LU

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2017

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of HAO LU find it satisfactory and recommend that it be accepted.

Ananth Kalyanaraman, Ph.D., Chair

Carl Hauser, Ph.D.

Assefaw Gebremedhin, Ph.D.

ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor Prof. Ananth Kalyanaraman for the continuous support of my Ph.D study, for his patience, motivation, and knowledge. His guidance helped me throughout the course of my Ph.D. With a 100% confidence, I can say that without his support, I could not have finished my Ph.D.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Carl Hauser and Prof. Assefaw Gebremedhin, for their insightful comments and encouragement.

My sincere thanks also goes to my collaborators Dr. Mahantesh Halappanavar and Dr. Daniel Chavarría-Miranda, who provided me with an opportunity to join their team as an intern, gave access to their laboratory space and research facilities, and acted as my mentors on my research. Without their precious support it would not have been possible to conduct this research. In particular, I would like to thank Dr. Mahantesh Halappanavar for effectively serving as my research mentor from PNNL throughout the course of my Ph.D., and for his extended support.

My dissertation research was supported by U.S. Department of Energy award DE-SC-0006516 and the U.S. National Science Foundation award IIS 0916463.

SCALABLE PARALLEL ALGORITHMS AND IMPLEMENTATIONS FOR LARGE-SCALE GRAPH ANALYSES

Abstract

by Hao Lu, Ph.D.
Washington State University
May 2017

Chair: Ananth Kalyanaraman

Scientific fields nowadays have adopted graph analytics into their discovery process. Different graphs have been generated from many different applied domains, such as social sciences, life sciences and business. Many of these application domains have become data-intensive, thereby emphasizing the need for scalability in computation. Addressing scalability is particularly a significant challenge for graph computations due to their inherent irregularity. In this dissertation, we address two graph problems—viz. *community detection* and *balanced graph coloring*.

Community detection is a well-studied problem that has multiple applications in several domains. Given a $G(V, E)$, the problem of community detection is one of identifying closely-knit subgroups of vertices ("communities"). For the most widely-used formulation, which is one of maximizing a metric called "modularity".

The *Louvain* method is one of the most widely used iterative community detection heuristic for modularity optimization, developed by Blondel et al. in 2008. Here, we observe certain key properties of this method that present challenges for its parallelization, and consequently propose heuristics that are designed to break those sequential barriers. Our approach, which we call *Grappolo*, demonstrates scaling on standard multicore platforms and emerging manycore platforms.

The other problem we address is balanced graph coloring. The goal for graph coloring is to assign colors to vertices such that no two adjacent vertices are assigned the same color (distance-1 coloring). Coloring can be used to identify independent tasks in parallel computing. Classical coloring heuristics attempt to reduce the number of colors used but the colorings generated in practice tend to have a skewed distribution in size, which could negatively impact parallel performance. Here, we propose multiple classes of heuristics to generate distance-1 colorings and partial distance-2 colorings (for bipartite graphs) with the dual objective of minimizing the number of color classes while ensuring that the color classes are balanced in size. We demonstrate the effectiveness of these heuristics on improving the performance of parallel community detection.

Different heuristics and design techniques presented in this dissertation can potentially be adapted into the broader context of parallelizing other graph operations that also have a similar irregular, and/or iterative structure to their computation.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENT . . . . . **iii**

ABSTRACT . . . . . **iv**

LIST OF FIGURES . . . . . **x**

LIST OF TABLES . . . . . **xv**

CHAPTER

1 INTRODUCTION **1**

1.1 Contributions of this dissertation 4

1.1.1 Key Publications 5

1.2 Organization of the dissertation 7

2 BACKGROUND AND NOTATION **8**

2.1 Background of Graph Theory 8

2.2 Basic Notation 10

3 COMMUNITY DETECTION **12**

3.1 Problem Overview 12

3.2 Problem statement and notation 13

3.3 The Louvain algorithm 15
3.4 Challenges in parallelization 17
3.4.1 Negative gain scenario 18
3.4.2 Swap and local maxima scenarios 20
3.5 Parallel heuristics 21
3.5.1 The minimum label heuristic 21
3.5.2 Coloring 23
3.5.3 The vertex following heuristic 24
3.5.4 Parallel algorithm 27
3.5.5 Implementation 28
3.5.6 Analysis 29
3.6 Experimental evaluation 30
3.6.1 Experimental setup 30
3.6.2 Performance evaluation 32
3.6.3 Effect of multiphase coloring 47
3.6.4 Effect of varying the modularity gain threshold 48
3.7 Extensions to Community Detection 49
3.7.1 Synchronization-based Extensions 49
3.7.2 Extension to Dynamic Graphs 52

4 BALANCED COLORING 55

4.1 Problem Overview 55
4.2 Problem Statement and Background 57
4.2.1 Related Work 58
4.2.2 A Foundational Scheme 60
4.2.3 Community Detection: A Motivating Application 62

4.3 Algorithms for Balanced Distance-1 Coloring 63
4.3.1 *Ab initio* balancing strategies 63
4.3.2 Guided balancing strategies 64
4.4 Parallel Algorithms 66
4.4.1 Parallelization using Unscheduled Moves 67
4.4.2 Parallelization using Scheduled Moves 70
4.4.3 Parallel Recoloring 71
4.4.4 Complexity 74
4.5 Partial Distance-2 Coloring 74
4.5.1 Preliminaries 74
4.5.2 Algorithms 77
4.5.3 Another example of an application 78
4.6 Implementation on the Tilera Platform 79
4.7 Experimental Results 80
4.7.1 Experimental Setup 80
4.7.2 Balance Quality Assessment 82
4.7.3 Performance Evaluation 85
4.7.4 Impact on the Community Detection Application 87
4.7.5 Results on Partial Distance-2 Coloring 89
4.8 Extensions to Balanced Coloring 91
4.8.1 Weighted Balancing 91
4.8.2 Lower Bound-based Coloring 91
4.8.3 Experimental Results 92

5 CONCLUSION 102

5.1 Community Detection 102

5.2 Balanced Coloring 103
5.3 Future Works 103

BIBLIOGRAPHY 106

LIST OF TABLES

2.1 Illustration of reduction of Königsberg Bridge Problem to graph problem. 9

3.1 Illustration of the negative gain scenario using an example of three vertices (Lemma 1). 18

3.2 Examples of cases which can be handled by using the minimum labeling heuristic. The dotted arrows point to the direction of the vertex migration. Case 1 shows a scenario of vertex swap between two communities. Case 2) shows the evolution of two different communities $\{i_1, i_2, i_3\}$ and $\{i_4, i_5, i_6, i_7\}$. Without the application of any heuristic (Case 2b), the algorithm may either form partial communities (e.g., $\{i_1\}, \{i_2, i_3\}$) or may settle on a local maxima (e.g., $\{i_4, i_6\}, \{i_5, i_7\}$). Whereas the use of a minimum label heuristic could help the communities converge to the final solutions faster (as shown in Case 2b). 21

3.3 Charts showing the evolution of modularity (left column) and the parallel runtime performance (right column) for each test input. The steep climbs in modularity visible in the modularity curves correspond to phase transitions. Also shown for comparison are the corresponding performance of the serial algorithm. 34

3.4 Charts showing the evolution of modularity (left column) and the parallel runtime performance (right column) for each test input. 35
3.5 Charts showing the evolution of modularity (left column) and the parallel runtime performance (right column) for each test input. 36
3.6 Charts showing the evolution of modularity (left column) and the parallel runtime performance (right column) for each test input. 37
3.7 Speedup charts for our parallel implementation, *Grappolo*. The chart on left shows the relative speedup of the parallel implementation using the 2-thread run as the reference. The chart on the right shows the absolute speedup — i.e., relative to the serial Louvain implementation *findcommunities*. All speedups are calculated using the *baseline+VF+Color* implementation of *Grappolo*. Note that in the absolute speedup chart, curves for Europe-osm and friendster are not shown because the serial Louvain implementation failed to complete on these two inputs. 40
3.8 Breakdown of the parallel run-times by the different steps of the algorithm - viz. coloring, time to perform the graph transformations between phases, and the time spent in the iterations. The runs correspond to the *baseline+VF+Color* implementation. 41
3.9 Chart showing the speedup curves for the graph rebuilding phase of our parallel algorithm. 41

3.10 Relative profile of performance for three combinations of heuristics: The relative performance of different heuristics and serial implementation for the test problems with respect to the best algorithm for a given problem. Europe-osm and friendster are not included in the comparison because the serial Louvain implementation crashes on those inputs. Final modularity scores are shown in the figure on left (part a), and run-times are shown on the right (part b). Run-time results from 32 thread runs were used to plot curves for the parallel heuristics. It is to be noted that the longer a heuristic's curve stays near the Y-axis the more superior its performance relative to the other schemes over a wider range of inputs. 45

3.11 Charts showing the evolution of modularity for the different versions (viz. baseline, Early Termination and Fully-Synchronized) of our community detection method on 16 threads. 51

4.1 a) The size distribution of the color classes obtained by the Greedy First Fit heuristic for distance-1 coloring on 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 Louvain method (Lu, Halappanavar, and Kalyanaraman, 2015). 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. 61

4.2 Illustration of equivalence among structurally orthogonal partition of a matrix A (a), partial distance-2 coloring of the vertices in V_2 in bipartite graph $G_b = (V_1, V_2, E)$ of A (b), and distance-1 coloring of the subgraph of the square graph induced by V_2, that is ${G_b}^2[V_2]$ (c). 75

4.3 **Distance-**1 **coloring:** 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 and random2, color class sizes from all balancing schemes are shown. For NLP-KKT200 and CNR, color class sizes only from the balancing schemes that produce same or comparable number of colors to the Greedy-FF scheme are shown. 82

4.4 **Recoloring:** The figure illustrates the impact of different bounds on bin sizes for the recoloring scheme (Algorithm 6). The default recoloring scheme which uses $\tau = 0.0$ (identifed in the chart by the "0.0" label) fixes the average size for each bin based on the numbers from the initial (unbalanced) coloring scheme. The average size is then varied from 10%, 20% and 30% (identified by labels "0.1", "0.2" and "0.3" respectively), which results in a smaller number of colors used and possibly a higher imbalance in bin sizes than the default recoloring scheme. The percentages inside paranthesis against each τ setting indicates the imbalance, measured by the Relative Standard Deviation of the resulting color class sizes. 83

4.5 (a, b) Speedup obtained by our Tilera manycore and x86 multi-core implementations of the VFF balancing scheme. Speedups are relative to one thread executions on both systems. (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 (Blondel et al., 2008). 86

4.6 **Partial distance-2 coloring:** The distributions of color class sizes produced by the different balanced coloring schemes (horizontal axis corresponds to colors (bins) and vertical axis (in $\log_2$ scale) to sizes of color classes). 90

4.7 **Balanced coloring extensions:** Distribution of color class sizes produced by the two distance-1 balanced coloring extensions (Weighted, LowerBound) compared to VFF balancing (without weights or lower-bounds) and and initial coloring (Greedy-FF). Horizontal axis corresponds to colors (bins) and vertical axis to sizes of color classes (measured in the *number* of vertices). 92

LIST OF FIGURES

3.1 Input statistics for the real world networks used in our experimental study. "RSD" represents the relative standard deviation of vertex degrees for each graph. It is given by the ratio between the standard deviation of the degree and its mean. 31

3.2 Comparison of the modularities and run-times achieved by our parallel implementation *baseline+VF+Color* (using 8 threads) against the corresponding values achieved by the serial Louvain implementation *findcommunities*. All runs were performed on the same test platform described under Experimental Setup. The "N/A' entries denote cases where the serial Louvain implementation did *not* complete (i.e., crashed). It is to be noted that the serial Louvain implementation is a 32-bit implementation. 42

3.3 Qualitative comparison between the parallel and serial community outputs by their composition. 46

3.4 Comparative results showing the effect of using coloring for only the first phase input vs. for multiple phases of the parallel algorithm. The multi-phase coloring scheme is same as the *baseline+VF+Color* scheme. All run-times are reported in seconds for runs corresponding to two threads. 47

3.5 Table showing the effect of varying the modularity gain threshold. Two sets of experiments were performed, each running the *baseline+VF+Color* implementation, while one using 10^{-2} and another 10^{-4} as the value for the modularity gain threshold used within the colored phases. . . 48
3.6 Runtime statistics with 16 threads. 52

4.1 A comprehensive list of balancing strategies for distance-1 coloring presented and studied in this paper. The input graph is denoted by $G = (V, E)$. The same set of strategies are also extended to obtain a balanced partial distance-2 coloring on bipartite graph inputs. 94
4.2 Input statistics for the graphs used in our distance-1 coloring study. 95
4.3 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 Greedy-FF, Recoloring and the two *ab initio* schemes is provided in parenthesis (next to their respective RSD values). 96
4.4 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). 97
4.5 Parallel run-times (in seconds) of the VFF scheme on different number of cores of the AMD x86 platform. Times shown are only for the balancing procedure (i.e., initial coloring time is *not* included). . . . 97

4.6 Parallel run-times (in seconds) of the three balancing schemes {VFF, Sched-Rev and Recoloring} on 16 Tilera cores. 97

4.7 Runtime (in seconds) comparison for the Guided VFF scheme vs. *Ab initio* (Greedy-LU) balancing scheme. All runs were performed on 32 threads of the AMD x86 platform. 98

4.8 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. 99

4.9 Statistics on structure of the bipartite real-world graphs $G = (V_1, V_2, E)$ used in our study. Recall that Δ corresponds to maximum degree. . 100

4.10 Parallel run-times (in seconds) of unbalanced and balanced {VFF, Sched-Rev} partial distance-2 schemes on the AMD platform. 32 cores were used for the KDD inputs, and 16 cores were used for the remaining inputs (due to smaller size). All runs were performed to color vertices in V_2. 100

4.11 Run-time evaluation of our extensions (weighted and LB-based) in comparison to the VFF balancing scheme and the Greedy-FF (initial coloring) scheme. All executions were performed on 16 threads of our x86 platform. 101

CHAPTER 1

INTRODUCTION

Graph analytics have become an integral part of the discovery pipeline in many application domains. The inherent connectivity and interrelationships of data, obtained from both naturally-built and synthetic systems, render themselves into graph representations. In simple terms, a graph ($G(V, E)$) consists of a set of vertices V and a set of edges E, such that vertices represent the entities and edges represent the pairwise relationship between any two vertices. For instance, in social networks, vertices represent the users and edges represent their social interactions. Similarly, in the life sciences, a protein-protein interaction network has vertices representing the proteins and edges representing their pairwise molecular interactions.

Given a graph, there are multiple operations that can be defined to mine for useful information from an application perspective. These include fundamental operations, such as finding shortest paths, detecting connected components, and performing breadth-first (depth-first) searches, or computing the number of triangles. In addition, there are more advanced operations that aid in mining for complex, higher order structural information from graphs, or to provide nontrivial insights into the connectivity properties of graphs.

In this dissertation, we address two such advanced operations—viz. community

detection and balanced graph coloring.

Community detection: Given a graph $G(V, E)$, the problem of *community detection* is one of identifying tightly-knit subgroups ("communities") of vertices, such that the strength of edge-based connections among vertices of the same community is significantly more than the strength of connections across communities (Fortunato, 2010). The strength of a pairwise relationship is represented by the weight of an edge.

The output of community detection is the set of communities, which represent a partitioning of the vertices. While, in principle, this output property ties community detection to the classical graph partitioning problem (Kernighan and Lin, 1970), there is a fundamental difference between the two problems. Unlike graph partitioning, community detection does *not* require any prior knowledge of the number of output communities or their sizes; instead the method is expected to discover/infer the number of communities based on the input. The idea is to find a hidden community-wise organization that may exist in a real world network. As a result, community detection is often used as a discovering tool to extract information about how a network is structurally organized. For instance, an analysis of a social network such as Facebook could reveal communities of friends, which could in turn be used for suggesting new friends sharing common interests (Newman and Girvan, 2004),(Newman, 2004a).

Balanced graph coloring: The other problem we address in this dissertation is *balanced graph coloring*. Given a graph $G(V, E)$, graph coloring is the problem of assigning every vertex a color, such that no two neighboring vertices (i.e., connected by an edge in the graph) are assigned the same color. This formulation is referred as the *distance-1 coloring* problem.

In parallel computation, graph coloring is often used in the scheduling of parallel tasks (Dániel, 2004). More specifically, a coloring represents maximal independent sets of vertices such that vertices that are assigned the same color are not directly

dependent on one another and therefore can be processed concurently. For example, given a graph where vertices represent classes and edges represent the time conflicts, a distance-1 graph coloring of the graph would represent a conflict-free final exam schedule.

Classical formulations of the graph coloring problem focus on minimizing the number of colors (Jensen and Toft, 2011). As a result, most current methods that implement graph coloring (Gebremedhin and Manne, 2000) solely focus on reducing the number of classes, however, without heeding attention to the individual *sizes* of each color class. In fact, due to the nature of the heuristics used, these methods often produce highly skewed distributions in the size of color classes. The size distribution of these color classes could have an impact on parallel performance of the end-application that uses these color classes to determine their parallel schedule. More specifically, skewed color classes could negatively impact the parallel efficiency (or thread utilization), while executing the steps corresponding to small color classes. In the interest of maintaining parallel efficiency across the color steps of computation, it not only becomes necessary to minimize the number of color classes but also to keep the color classes balanced in their size distribution.

Parallelizing graph computations: Scaling up graph computations has become an important area of research. Over the last two decades, multiple application domains have experienced an explosion of data due to advancements in technology and a growing interest in obtaining a deeper understanding of the system through the lens of the observations made (i.e., data). As a result, real world networks that have millions of vertices and billions of edges have become commonplace. For instance, the size of a web graph in 2014 covers 1.7 billion web pages and connected by 64 billion hyperlinks (*Web Data Commons - Hyperlink Graphs*). To enable the processing of such large graphs, parallelization has become essential.

Besides the challenges associated with large size, graph applications also face a unique set of challenges that are due to certain inherent characteristics of their computation. More specifically, graph computations are known to generate highly irregular data access patterns, which in turn cause complications when it comes to preserving data locality or keeping memory-related overheads minimal. These challenges become more pronounced in the context of parallelization.

1.1 Contributions of this dissertation

In this dissertation, we address two graph-theoretic problems—viz. community detection and balanced graph coloring. More specifically, we present the design and development of efficient parallel techniques and implementations for conducting both these operations on modern day multi-core and many-core architectures. The contributions are as follows:

In the context of community detection:

i) We introduce novel and effective heuristics for parallelizing the Louvain algorithm (Blondel et al., 2008) on multithreaded architectures;

ii) Experimental studies using numerous real-world networks obtained from varied sources demonstrate the effectiveness of our parallel approach both in terms of performance and quality;

iii) A thorough comparative study of the performance and related trade-offs among the different parallel heuristics compared to the serial heuristic.

In the context of balanced graph coloring:

i) We address two problems—a) to identify a balanced distance-1 graph coloring

for general graphs; and b) to identify a balanced partial distance-2 coloring for bipartite graphs;

ii) We present two classes of heuristics (viz. *ab initio* and guided) for achieving a balanced coloring under both problem formulations;

iii) We present the design of different parallelization strategies for generating a balanced coloring on conventional multicore (x86) architectures and a specialized many-core architecture (the EZchip Tilera platform (Bell et al., 2008)[1]).

iv) Our experimental results demonstrate that effectiveness of the proposed heuristics in balancing the color classes, and also in improving the performance of an end-application.

We believe that many of the techniques and heuristics that we propose in this dissertation for these two graph operations to carry over to other graph problems with similar computation characteristics—viz. vertex-centric computations that have a greedy iterative structure and/or relies on querying information from local neighborhoods.

1.1.1 Key Publications

Components of this dissertation have been published in the following peer-reviewed venues:

- The community detection work was first published at the 2014 IEEE International Workshop on Multithreaded Architectures and Applications (MTAAP) (Lu et al., 2014), and later expanded into a journal article and published in the Parallel Computing journal (Lu, Halappanavar, and Kalyanaraman, 2015).

[1]Henceforth, this platform will be identified as the "Tilera" platform.

This work was a collaborative effort among Dr. Ananth Kalyanaraman (AK), Dr. Mahantesh Halappanavar (MH) and myself (HL). All three contributed to the problem formulation and algorithm design. MH and HL implemented the parallel algorithms and conducted the experiments, while most of the writing was managed by AK and MH.

- The balanced graph coloring work was first published at the 2015 IEEE International Parallel and Distributed Processing Symposium (Lu et al., 2015), and later expanded into a journal article and published in IEEE Transactions on Parallel & Distributed Systems (Lu et al., 2016). This work was a collaborative effort among Dr. Ananth Kalyanaraman, Dr. Mahantesh Halappanavar, Dr. Assefaw Gebremedhin (AG), Dr. Daniel Chavarría-Miranda (DC), Dr. Ajay Panyala (AP) and myself. HL, AK, MH, AG and DM contributed to different aspects of the problem formulation and algorithm design. HL and MH implemented the algorithms and generated the experimental results for x86 multicore platforms. DM and AP implemented the algorithms and generated the experimental results for the Tilera manycore platform. AK, MH, and AG performed most of the paper writing with contributions from all the authors.

Related bodies of work originating from this dissertation were also published in other venues. More specifically, we designed variants of our community detection and coloring algorithms for Network-on-Chip (NOC)-based architectures. This work was conducted in collaboration with Dr. Partha Pande, Mr. Karthi Duraisamy and Dr. Ananth Kalyanaraman. The results of these collaborative efforts were published at the 2015 International Conference on Compilers, Architecture and Synthesis of Embedded Systems (Duraisamy et al., 2015) and the ACM Transactions on Embedded Computing Systems (Duraisamy et al., 2016). Furthermore, an extended review

article on parallel community detection was also published at the Foundations and Trends in Electronic Design Automation (Kalyanaraman et al., 2016).

1.2 Organization of the dissertation

The rest of the dissertation is organized as follows: In Chapter 2, we provide a background for graph theory and introduce basic notation that will be used in the remainder of the dissertation. In Chapter 3, we present our heuristics and related results for parallel community detection. This chapter is mostly a reproduction of our journal paper on parallel community detection (Lu, Halappanavar, and Kalyanaraman, 2015), with a few additions that describe certain extensions of that work. In Chapter 4, we present our heuristics and related results for balanced graph coloring. This chapter is mostly a reproduction of our journal paper on balanced graph coloring (Lu et al., 2016). Finally, in Chapter 5, we summarize our key findings and provide a preview for potential future extensions.

CHAPTER 2

BACKGROUND AND NOTATION

2.1 Background of Graph Theory

Graph theory is the study of graphs, which are abstract representations used to model relationships among objects. The origin of graph theory dates back to 1736 in the form of the Königsberg Bridge problem. A Swiss mathematician, Leonhard Euler, was presented with a problem to traverse through each of the seven bridges, which connected the four separated lands in the Prussian town of Königsberg, exactly once. Euler reduced this to a graph problem by representing each land as a "vertex" and each bridge as an "edges". Consequently, he showed that in order to traverse each edge once, all vertices, except the starting and ending vertices, must have an even degree, as shown in Figure 2.1.

However, the term "graph" was not formally introduced until hundred and fifty years later. In 1878, an English mathematician James Joseph Sylvester publish a paper in Nature, defined the term graph (Sylvester, 1878). Since then, numerous graph related problems have been studied.

The branch of graph theory that studies the structural properties of graphs is called *network topology*. In 1951, Rapoport and Solomonoff proposed that naturally-

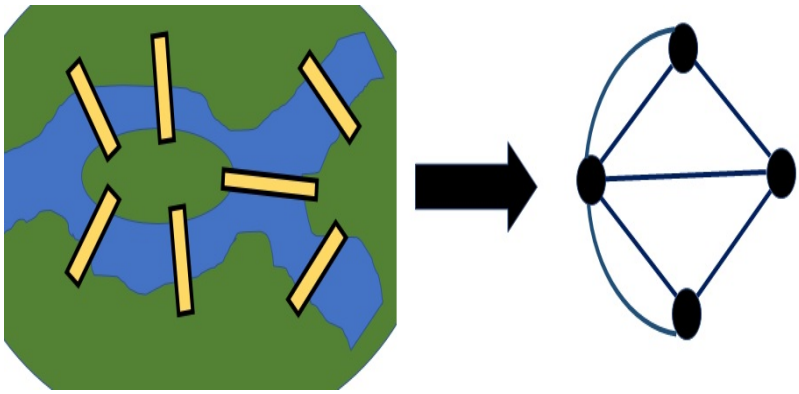


Figure 2.1 Illustration of reduction of Königsberg Bridge Problem to graph problem.

built systems such as neural net and disease epidemiology can be presented in the form of random graphs (Solomonoff and Rapoport, 1951). In 1960, Paul Erdös and Alfred Rènyi developed Erdös-Rènyi model to generate random networks; according to the model, all graphs generated with same set of vertices and a fixed number of edges are equally likely (Erdös and Rényi, 1960). However, a significant deviation from the random graph model was observed by a British statistician, Udny Yule, in 1925. He observed that the growth rate of each individual species depends on the current size of the species; this was formulated using preferential attachment (Yule, 1925). This notion of rich getting richer was later used to model the degree distribution of real world networks, which effectively laid the foundation for the scale-free network model (Barabási, 2009). Another influential study in graph topology was done in the domain of social science. In 1967, Stanley Milgram concluded that the social network that connected people has a "six degree of separation" (Milgram, 1967). This result would later lead to the extensive study of small-world networks (Watts and Strogatz, 1998).

Despite these early advances, more studies about the structural organization for non-random real world graphs emerged only in recent years. In 1973, Granovetter

suggested that social networks, built by friendship or relationship, consist of strongly connected groups (i.e., "communities") and loosely connected groups (Granovetter, 1973). This observation was one of the first studies to identify a higher order of structure from a real world graph. Since then, multiple efforts have focused on identifying the community-level organization of a wide variety of real world networks built from both natural and engineered systems. These include the Internet, social networks (Papadopoulos et al., 2012), *C. Elegans* neural system (Watts and Strogatz, 1998), protein interaction networks (Girvan and Newman, 2002), electric power grid (Newman, 2003), dolphin communication network (Connor, Heithaus, and Barre, 2001), collaboration networks (Newman, 2003), customer preference databases (Reddy et al., 2002), and climate variability networks (Steinhaeuser, Chawla, and Ganguly, 2011).

Consequently, community detection has proven to be of critical value in multiple application domains, to gain important structural insights into real world networks. Graph coloring also represents a structural property of the graph—one that relates to identification of mutually independent sets of vertices. A more detailed related work on community detection and graph coloring are in Sec 3.1 and 4.2

2.2 Basic Notation

We denote an undirected graph as $G(V, E)$, where V is the set of vertices, and E is the set of edges in the form of $\{(i, j)|i, j \in V\}$. If the edges have numerical weights, then we denote a weighted undirected graph as $G(V, E, \omega)$, where $\omega(.)$ is a weighting function that maps every edge in E to a nonzero positive weight.

As a special consideration in this dissertation, we allow edges that connect a vertex to itself—i.e., (i, i) can be a valid edge. However, multi-edges are not allowed—e.g., (i, j, k) is not a valid edge.

For a given vertex i, it's adjacency list is denoted by $\Gamma(i) = \{j|(i,j) \in E\}$. Also, the weighted degree of vertex i is given by: $k_i = \sum_{\forall j \in \Gamma(i)} \omega(i,j)$.

We will use n to denote the number of vertices in G; M to denote the *number* of edges in G; and m to denote the sum of the edge weights of all edges in G—i.e., $m = \frac{1}{2} \sum_{\forall i \in V} k_i$.

CHAPTER 3

COMMUNITY DETECTION

3.1 Problem Overview

Community detection, or graph clustering, is becoming pervasive in the data analytics of various fields including (but not limited to) scientific computing, life sciences, social network analysis, and internet applications (Fortunato, 2010). As data grows at explosive rates, the need for scalable tools to support fast implementations of complex network analytical functions such as community detection is critical. Given a graph, the problem of community detection is to compute a partitioning of vertices into communities that are closely related within and weakly across communities. Modularity is a metric that can be used to measure the quality of communities detected (Newman and Girvan, 2004). Community detection with maximum modularity is an NP-Complete problem, but fast heuristics exist. One such heuristic is the Louvain method (Blondel et al., 2008).

Our basis for selecting the Louvain heuristic for parallelization hinges on its increasing popularity within the user community and owing to its strengths in algorithmic and qualitative robustness. With well over 1,000 citations to the original paper (as of this writing), the user base for this method has been rapidly expanding

in the last few years. Yet, there is no scalable parallel implementation available for this heuristic. As network sizes continue to grow rapidly into a scale of tens or even hundreds of billions of edges (*The 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering*), the memory and runtime limits of the serial implementation are likely to be tested. However, parallelization of this inherently serial algorithm can be challenging (as discussed in Sections 3.2 and 3.4).

The parallel solutions presented in this dissertation (Section 3.5) provide a way to overcome key scalability challenges. In devising our algorithm, we factored in the need to parallelize without compromising the quality of the original serial heuristic and yet be capable of achieving substantial scalability. We also factored in the need for stable solutions across different platforms and programming models. The resulting algorithm, presented in Section 3.5.4, is a combination of heuristics that can be implemented on both shared and distributed memory machines. As demonstrated in our experimental section (Section 3.6), our implementations provide outputs that have either a higher or comparable modularity to that of the serial method, and is able to reduce the time to solution by factors of up to 16×. These observations are supported over a number of real-world networks.

3.2 Problem statement and notation

A *community* within graph G represents a (possibly empty[1]) subset of V. In practice, for community detection, we are interested in partitioning the vertex set V into an arbitrary number of *disjoint* non-empty communities, each with an arbitrary size (> 0 and $\leq n$). We call a community with just one element as a *singlet* community. We will

[1]The notion of empty communities do not have a practical relevance. We have intentionally defined it this way so as to make our later algorithmic descriptions easier. It is guaranteed, however, that all output communities at the end of our algorithm will be non-empty subsets.

use $C(i)$ to denote the community that contains vertex i in a given partitioning of V. We use the term *intra-community edge* to refer to an edge that connects two vertices of the same community. All other edges are referred to as *inter-community edges*. Let $E_{i \to C}$ refer to the set of all edges connecting vertex i to vertices in community C. And let $e_{i \to C}$ denote the sum of the weights of the edges in $E_{i \to C}$ (also referred to as the degree of a community).

$$e_{i \to C} \;=\; \sum_{\forall (i,j) \in E_{i \to C}} \omega(i,j) \tag{3.1}$$

Let a_C denote the sum of the degrees of all the vertices in community C.

$$a_c \;=\; \sum_{\forall i \in C} k_i \tag{3.2}$$

Modularity: Let $P = \{C_1, C_2, \ldots C_k\}$ denote the set of all communities in a given partitioning of the vertex set V in $G(V, E, \omega)$, where $1 \leq k \leq n$. Consequently, the *modularity* (denoted by Q) of the partitioning P is given by the following expression (Newman and Girvan, 2004):

$$Q \;=\; \frac{1}{2m} \sum_{\forall i \in V} e_{i \to C(i)} \;-\; \sum_{\forall C \in P} \left(\frac{a_C}{2m} \cdot \frac{a_C}{2m}\right) \tag{3.3}$$

Intuitively, modularity is a statistical measure for assessing the quality of a given community-wise partitioning (or equivalently, "clustering"). A "good" clustering method is one that clusters closely related elements (vertices) as part of the same community (or "cluster") while separating weakly related elements into different clusters. In other words, the goal becomes one of maximizing intra-community links while keeping the number of inter-community edges low. This explains the first term in Eqn. (3.3). However, if the goal is simply to maximize the contribution from intra-community edges, then one could potentially assign all vertices into one community. But such a solution is likely to be meaningless in practice. To overcome this problem,

the second term in the Eqn. (3.3) was introduced. This term represents the fraction of intra-community edges one would expect in an "equivalent" graph (i.e., another graph with the same numbers of vertices and edges, and the same vertex degrees) but with just the edges randomly reconnected.

Modularity is not the ideal metric for community detection and issues such as resolution limit have been identified (Fortunato, 2010; Traag, Van Dooren, and Nesterov, 2011), and consequently, a few variants of modularity definitions have been devised (Traag, Van Dooren, and Nesterov, 2011; Bader and McCloskey, 2009; Berry et al., 2011) are available. However, the definition provided in Eqn. (3.3) continues to be the more widely adopted version in practice, including in the Louvain method (Blondel et al., 2008), and therefore, we will use that definition for this dissertation.

Community detection: Given $G(V, E, \omega)$, the problem of community detection is to compute a partitioning P of communities that maximizes modularity.

This problem has been shown to be NP-Complete (Brandes et al., 2008). Note that this problem is different from graph partitioning problem and its variants (Hendrickson and Kolda, 2000), where the number of clusters and the rough size distribution of those target clusters are known *a priori*. In the case of community detection, both quantities are unknown prior to computation. In fact they encapsulate the input properties that one seeks to discover out of the community detection exercise.

3.3 The Louvain algorithm

In 2008, Blondel *et al.* presented an algorithm for community detection (Blondel et al., 2008). The method, called the *Louvain* method, is a multi-phase, iterative, greedy heuristic capable of producing a hierarchy of communities. The main idea of the algorithm can be summarized as follows: The algorithm has multiple *phases*, and

within each phase it carries out multiple *iterations* until a convergence criterion is met.

At the beginning of the first phase, each vertex is assigned to a separate community. Subsequently, the algorithm progresses from one iteration to another until the net modularity *gain* becomes negligible (as defined by a predefined threshold). Within each *iteration*, the algorithm linearly scans the vertices in an arbitrary but predefined order. For every vertex i, all its neighboring communities (i.e., the communities containing $i's$ neighbors) are examined and the modularity gain that would result if i were to move to each of those neighboring communities from its current community is calculated. Once the gains are calculated, the algorithm assigns a neighboring community that would yield the maximum modularity gain, as the new community for i (i.e., new $C(i)$), and updates the corresponding data structures that it maintains for the source and target communities. Alternatively, if all gains turn out to be negative, the vertex stays in its current community. An iteration ends once all vertices are linearly scanned in this fashion. Consequently, the modularity is a monotonically increasing function across iterations of a phase.

Once the algorithm converges within a phase, it proceeds to the next *phase* by collapsing all vertices of a community to a single "meta-vertex"; placing an edge from that meta-vertex to itself with an edge weight that is the sum of weights of all the intra-community edges within that community; and placing an edge between two meta-vertices with a weight that is equal to the sum of the weights of all the inter-community edges between the corresponding two communities. The result is a condensed graph $G'(V', E', \omega')$, which then becomes the input to the next phase. Subsequently, multiple phases are carried out until the modularity score converges. Note that each phase represents a coarser level of hierarchy in the community detection process.

At any given iteration, let $\Delta Q_{i \to C(j)}$ denote the modularity gain that would result from moving a vertex i from its current community $C(i)$ to a different community $C(j)$. This term is given by:

$$\begin{aligned} \Delta Q_{i \to C(j)} &= \frac{e_{i \to C(j)} - e_{i \to C(i) \setminus \{i\}}}{m} \\ &\quad + \frac{2 \cdot k_i \cdot a_{C(i) \setminus \{i\}} - 2 \cdot k_i \cdot a_{C(j)}}{(2m)^2} \end{aligned} \tag{3.4}$$

Consequently, the new community assignment for i at an iteration is determined as follows. For $j \in \Gamma(i) \cup \{i\}$:

$$C(i) = \arg\max_{C(j)} \Delta Q_{i \to C(j)} \tag{3.5}$$

In the implementation (*findcommunities*), several data structures are maintained such that each instance of $\Delta Q_{i \to C(j)}$ can be computed in $O(1)$ time. Consequently, the algorithm's time complexity *per* iteration is $O(M)$. While no upper bound has been established on the number of iterations or on the number of phases, it should be evident that the algorithm is guaranteed to terminate with the use of a cutoff for the modularity gain (because of the modularity being a monotonically increasing function until termination). In practice, the method needs only tens of iterations and fewer phases to terminate on most real world inputs.

3.4 Challenges in parallelization

Any attempt at parallelizing the Louvain method should factor in the sequential nature in which the vertices are visited within each iteration and the impact it has on convergence. Visiting the vertices sequentially gives the advantage of working with the latest information available from all the preceding vertices in this greedy procedure. Furthermore, in the serial algorithm, when a vertex computes its new community assignment (using Eqn.(3.5)), it does so with the guarantee that no other

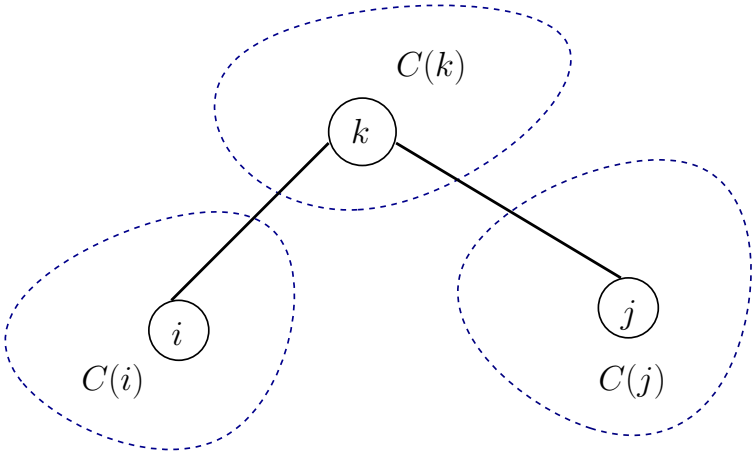


Figure 3.1 Illustration of the negative gain scenario using an example of three vertices (Lemma 1).

part of the community structure is concurrently being altered. These guarantees may *not* hold in *parallel*. In other words, if communities are updated in parallel, it could lead to some interesting situations with an impact on the convergence process as described below.

3.4.1 Negative gain scenario

To illustrate the case in point, consider the example scenario illustrated in Figure 3.1, where two vertices i and j are both connected to a third vertex k with all three of them in different communities initially — i.e., $i \in C(i)$, $j \in C(j)$, $k \in C(k)$ s.t. $C(i) \neq C(j) \neq C(k)$. If both vertices i and j evaluate the possibility of moving to $C(k)$ independently, using Eqn.(3.4), then from each of their perspectives, their *predicted* value for the new modularity is $Q_{old} + \Delta Q_{i \to C(k)}$ and $Q_{old} + \Delta Q_{j \to C(k)}$, respectively. However, if both i and j decide to move to $C(k)$ in parallel, then the *actual* value for the new modularity will be $Q_{old} + \Delta Q_{\{i,j\} \to C(k)}$, where:

$$\begin{aligned} \Delta Q_{\{i,j\} \to C(k)} &= \Delta Q_{i \to C(k)} + \Delta Q_{j \to C(k)} \\ &\quad + \frac{\omega(i,j)}{m} - \frac{2 \cdot k_i \cdot k_j}{(2m)^2} \end{aligned} \tag{3.6}$$

If $(i,j) \notin E$, $\omega(i,j) = 0$, implying:

$$\begin{aligned} \Delta Q_{\{i,j\} \to C(k)} &= \Delta Q_{i \to C(k)} + \Delta Q_{j \to C(k)} \\ &\quad - \frac{2 \cdot k_i \cdot k_j}{(2m)^2} \\ &\leq \Delta Q_{i \to C(k)} + \Delta Q_{j \to C(k)} \end{aligned} \tag{3.7}$$

Furthermore, if $\Delta Q_{i \to C(k)} + \Delta Q_{j \to C(k)} < \frac{2 \cdot k_i \cdot k_j}{(2m)^2}$

$$\Rightarrow \quad \Delta Q_{\{i,j\} \to C(k)} < 0 \tag{3.8}$$

On the other hand, if $\frac{\omega(i,j)}{m} > \frac{2 \cdot k_i \cdot k_j}{(2m)^2}$ (can be true only if $(i,j) \in E$), then:

$$\Delta Q_{\{i,j\} \to C(k)} > \Delta Q_{i \to C(k)} + \Delta Q_{j \to C(k)} \tag{3.9}$$

This is because $\Delta Q_{i \to C(k)} > 0$ and $\Delta Q_{j \to C(k)} > 0$; the latter two inequalities follow from the fact that i and j chose to move to $C(k)$. Note that if this happens, then parallel version could potentially surpass the serial version toward modularity convergence.

Lemma 1. *At any given iteration of the Louvain algorithm, if community updates for vertices are performed in parallel, then the net modularity gain achieved cannot be guaranteed to be always positive.*

Proof. Follows directly from inequality (3.7). □

The above lemma has a direct implication on the convergence property of the Louvain method, one way or another. Pessimistically speaking, if the net modularity gain can become negative between consecutive iterations of the algorithm, then there is no theoretical guarantee that the algorithm will terminate. Even if the chances of non-termination turn out to be bleak, it could potentially slow down the rate at which the algorithm progresses toward a solution, causing more number of iterations. For

this reason, the *number of iterations* that the algorithm takes to converge toward the solution and the *quality of the solution* relative to the serial algorithm's can be good indicators of the effectiveness of a parallel strategy. Note that the above example with three vertices can be extended to scenarios where multiple unrelated vertices are trying to enter a community at its periphery without mutual knowledge.

3.4.2 Swap and local maxima scenarios

There exists another scenario that could impede the progression of the parallel algorithm toward a solution. Consider a simple example where two vertices i and j connected by an edge $(i, j) \in E$ s.t., $C(i) = \{i\}$ and $C(j) = \{j\}$. In the interest of increasing modularity, if the two vertices make a decision to move to each other's community concurrently, then such an update could potentially result in both vertices simply swapping their community assignments without achieving any modularity gain. This could also happen in a more generalized setting, where subsets of vertices between two different communities swap their community assignments, each unaware of the other's intent to also migrate.

A parallel algorithm also runs the risk of settling on locally optimal decisions. This could happen even in serial; in parallel such scenarios may arise if a single community gets partitioned into equally weighted sub-communities, in which there is no incentive for any individual vertex to merge with any of the other sub-communities; and yet, if all vertices from each of the sub-communities were to merge together to form a single community the net modularity gain could be positive. An example of this case will be shown later in Section 3.5.1. Getting stuck in a locally optimal solution, however, can be resolved when the algorithm progresses to subsequent phases.

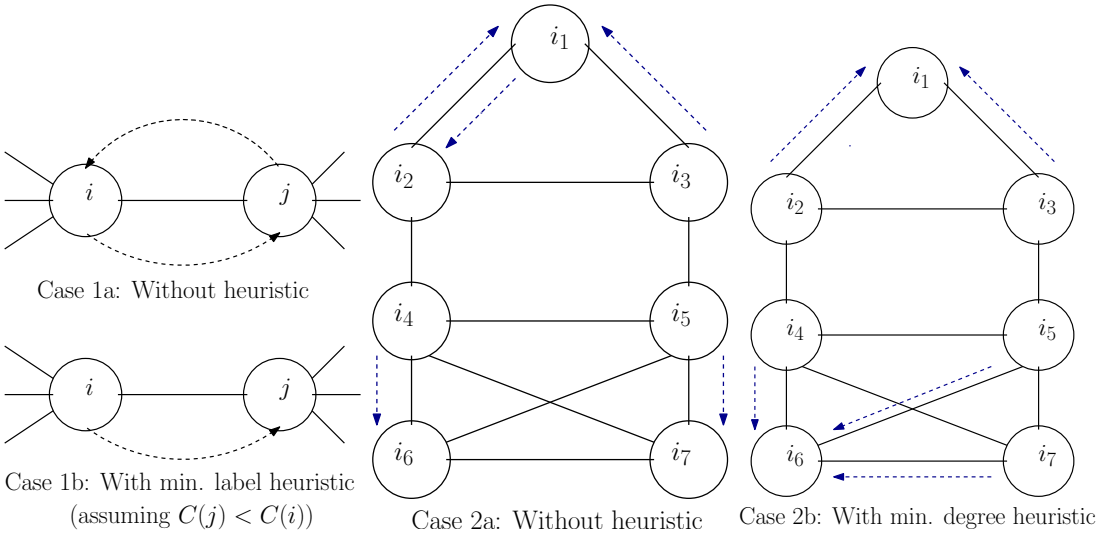


Figure 3.2 Examples of cases which can be handled by using the minimum labeling heuristic. The dotted arrows point to the direction of the vertex migration. Case 1 shows a scenario of vertex swap between two communities. Case 2) shows the evolution of two different communities $\{i_1, i_2, i_3\}$ and $\{i_4, i_5, i_6, i_7\}$. Without the application of any heuristic (Case 2b), the algorithm may either form partial communities (e.g., $\{i_1\}, \{i_2, i_3\}$) or may settle on a local maxima (e.g., $\{i_4, i_6\}, \{i_5, i_7\}$). Whereas the use of a minimum label heuristic could help the communities converge to the final solutions faster (as shown in Case 2b).

3.5 Parallel heuristics

In this section, we present our ideas to tackle the challenges outlined above in parallelizing the Louvain heuristic community detection.

3.5.1 The minimum label heuristic

Section 3.4.2 elaborated on the possibilities of swapping conditions that may delay the parallel algorithm's convergence to a solution. In this section we present a heuristic designed to address some of these cases. Let us consider the simple case of two vertices i and j outlined in Section 3.4.2. Here both vertices are initially in communities of size one, and a decision in favor of merging at any given iteration will lead them to

simply swap their respective communities without resulting in any net modularity gain. This is outlined in the Case 1a of Figure 3.2. Such a swap can be easily prevented by introducing a labeling scheme where it can be enforced that only one of them move to other's community. More specifically, let the communities at any given stage of the algorithm be labeled numerically (in an arbitrary order). We will use the notation $\ell(C)$ to denote the label of a community C. Then the heuristic is as follows:

The singlet minimum label heuristic: In the parallel algorithm, at any given iteration, if a vertex i which is in a community by itself (i.e., $C(i) = \{i\}$), decides (in the interest of modularity gain) to move to another community $C(j)$ which also contains only one vertex j, then that move will be performed *only if* $\ell(C(j)) < \ell(C(i))$.

The above heuristic can be generalized to other cases of swapping or local maxima. For instance, let us consider the 4-clique of $\{i_4, i_5, i_6, i_7\}$ shown in Figure 3.2: case 2, assuming that each vertex is in its own individual community to start with. Here, in the absence of an appropriate heuristic there is a chance that the algorithm would settle on a local maxima. For instance, maximum modularity gains can be achieved at vertex i_4 by either moving to $C(i_6)$ or $C(i_7)$, and similarly for vertex i_5. However, if i_4 moves to $C(i_6)$ and i_5 to $C(i_7)$, then the resulting solution $\{i_4, i_6\}, \{i_5, i_7\}$ (shown in case 2a of Figure 3.2) will represent a local maxima from which the algorithm may not proceed in the current phase. This is because, once these partial communities form, there is no incentive for i_4 or i_6 to individually move to the community containing $\{i_5, i_7\}$, without each other's company. This is a limitation imposed by the Louvain heuristic, which makes decisions at the vertex level. However, if we label and treat the communities in a certain way then such local maxima situations can be avoided.

The generalized minimum label heuristic: In the parallel algorithm, at

any given iteration, if a vertex i has *multiple* neighboring communities yielding the maximum modularity gain, then the community which has the minimum label among them will be selected as i's destination community.

In the example for Figure 3.2:case 2, vertices i_6 and i_7 will both yield the maximum modularity gain for vertices i_4 and i_5. However, using the above minimum label heuristic, all three vertices $\{i_4, i_5, i_7\}$ will migrate to $C(i_6)$, while i_6 stays in $C(i_6)$ — i.e., assuming $\ell(C(i_4)) < \ell(C(i_5)) < \ell(C(i_6)) < \ell(C(i_7))$.

While swap situations may delay convergence, they can never lead to nontermination of the algorithm due to the use of a minimum required net modularity gain threshold to continue a phase. As for local maxima, a general proof that effects of elimination of local maxima cases progressively as the algorithm progresses is not possible due to the heuristic nature of algorithm. However, many situations, similar to those explained earlier in Section 3.4.2, typically get resolved in subsequent phases; this is because the representation of the individual sub-communities as meta-vertices is likely to lead them to merge with one another forming the containing communities eventually in the output.

3.5.2 Coloring

In this section, we explore the idea of graph coloring to address some of the parallelization challenges outlined in Section 3.4. A distance-k coloring of a graph is an assignment of colors to vertices such that no two vertices separated by a distance of at most k are assigned the same color. It should be easy to see that using distance-1 coloring to partition the vertices into color sets prior to the processing would prevent vertex-to-vertex swap scenarios. In this scheme, vertices of the same color are processed in parallel, and this is equivalent of guaranteeing that no two adjacent vertices will be processed concurrently. However, distance-1 coloring may not be adequate

to address other potential complications that may arise during parallelization (see Section 3.4.1).

Corollary 2. *Applying and processing the vertices in parallel by distance-1 coloring does* not *necessarily preclude the possibility of negative modularity gains between iterations.*

Proof. Follows directly from the three vertex example case presented for Lemma 1. □

In fact the same result can be extended for application of a distance-k coloring scheme, where $k > 1$, as was shown in (Lu et al., 2014).

Despite these lack of guarantees for a positive modularity gain between iterations, coloring still could be effective as a heuristic in practice, as we will demonstrate in Section 3.6. The performance trade-off presented by coloring is a potential reduction in the degree of parallelism versus faster convergence to higher modularity. Coloring also presents an added advantage of being able to use higher modularity gain thresholds during the earlier phases of the algorithm, as will be explored in Section 3.6. The run-time cost of coloring is expected to be dominated by the time spent within iterations; furthermore, for scalability in preprocessing, we use a parallel implementation to perform coloring (Catalyurek et al., 2012).

3.5.3 The vertex following heuristic

In this section, we will layout a particular property of the *serial* Louvain algorithm in the way it treats vertices with single neighbors, and devise a heuristic around it. For the purpose of the lemma below, we will assume a version of Louvain algorithm which continues with iterations within a phase, until the communities stop changing. We also distinguish between vertex i being a *single degree* vertex and a *single neighbor*

vertex — the former is when the only edge incident on i is (i, j), whereas the latter is when i could have up to two edges incident with (i, j) being mandatory and (i, i) being optional.

Lemma 3. *Given an input graph $G(V, E, \omega)$, let i and j be two different vertices such that i is a single degree vertex with only one incident edge $(i, j) \in E$. Then, in the final solution $C(i) = C(j)$ — i.e., i should be part of the same community as j.*

Proof. Consider any iteration r in which vertices i and j are in two different communities — i.e., $C(i) \neq C(j)$. During iteration r, the value of $\Delta Q_{i \to C(j)}$ will evaluate to the following:

$$\begin{aligned} \Delta Q_{i \to C(j)} &= \frac{\omega(i,j)}{m} + \frac{2 \cdot k_i \cdot a_{C(i)\setminus\{i\}} - 2 \cdot k_i \cdot a_{C(j)}}{(2m)^2} \\ &\geq \frac{\omega(i,j)}{m} - \frac{2 \cdot k_i \cdot a_{C(j)}}{(2m)^2} \ (\because a_{C(i)\setminus\{i\}} \geq 0) \\ &= \frac{\omega(i,j)}{2m^2} \left(2m - \frac{k_i \cdot a_{C(j)}}{\omega(i,j)} \right) \end{aligned} \tag{3.10}$$

Since vertex i is a single degree vertex, $k_i = \omega(i, j)$. Therefore,

$$\Delta Q_{i \to C(j)} \geq \frac{\omega(i,j)}{2m^2} \left(2m - a_{C(j)} \right) \tag{3.11}$$

Now, if i were to decide *against* moving to $C(j)$, $\Delta Q_{i \to C(j)} \leq 0$. Given that the above inequality (3.11) is a lower bound for $\Delta Q_{i \to C(j)}$, and also that all edge weights are non-negative:

$$\begin{aligned} \Rightarrow 2m - a_{C(j)} &\leq 0 \\ \Rightarrow 2m &\leq a_{C(j)} \end{aligned} \tag{3.12}$$

But inequality (3.12) is *not* possible because $a_{C(j)} \leq 2m$ for any community (by the definition in Eqn.3.2) and in this case, since $i \notin C(j)$, $a_{C(j)} \leq (2m - \omega(i, j)) < 2m$. This implies that i will have no choice but to move to $C(j)$ in iteration r. □

We refer to the guarantee provided by the above lemma as the *vertex following (VF) rule*. Note that it is guaranteed to hold only for single degree vertices in the input graph. The implication of this rule is that there is no need to explicitly make decisions on single degree vertices during the Louvain algorithm's iterations. Instead, we can preprocess the input such that all single degree vertices are merged *a priori* into their respective neighboring vertex. More specifically, let i be a single degree vertex with j as its neighbor. Then, we remove vertex i from the graph, and replace j with a new vertex j', such that $\Gamma(j') = \{\Gamma(j) \setminus \{i\}\} \cup \{j'\}$ and $\omega(j', j') = \omega(i, j)$ if $(j, j) \notin E$; and $\omega(j', j') = \omega(j, j) + \omega(i, j)$ otherwise.

This preprocessing not only could help reduce the number of vertices that need to be considered during each iteration, but it also allows the vertices that contain multiple neighbors (that tend to be the hubs in the networks) be the main drivers of community migration decisions. This is more important under a parallel setting because if the single degree vertices were retained in the network the hub nodes could potentially gravitate temporarily toward one of their single degree mates, thereby delaying progression of solution or getting stuck in a local maxima.

We could also extend the result of the Lemma 3 to benefit cases where vertex i is a single *neighbor* vertex. The idea is similar to that of a k-core decomposition of the graph (Batagelj and Zaveršnik, 2002). Intuitively, during preprocessing, single neighbor vertices can be collapsed into their only neighboring vertex recursively until the negative component of the inequality (3.10) starts to dominate its positive counterpart. Termination of this recursive merging can be implemented either by explicitly calculating both sides of the inequality (3.10) or by estimating through other means via lower bounds or statistical thresholds. The idea is to lead to fast compression of chains within the input graph prior to application of the Louvain heuristic. We omit further details of this idea and for the purpose of this dissertation, we only consider

the single degree version of the vertex following heuristic for implementation and experimental evaluation.

3.5.4 Parallel algorithm

Our parallel algorithm has the following major steps:

1) VF preprocessing (Optional): Apply the vertex following heuristic by merging all single degree vertices into their respective neighboring vertices (as explained in Section 3.5.3). This step is performed in parallel. Label the resulting vertices from $1 \ldots n$ using an arbitrary ordering.
2) Coloring preprocessing (Optional): Color the input vertices using distance-k coloring. For this dissertation, we only explore distance-1 coloring. For coloring, we used the parallel implementation from (Catalyurek et al., 2012).
3) Phases: Execute phases one at a time as per Algorithm 1. Within each phase, the algorithm runs multiple iterations, with each iteration performing a parallel sweep of vertices without locks and using the community information available from the previous iteration. If coloring was applied, then the processing of each color set is parallelized internally and the community information from the previous coloring stages is available to make migration decisions in subsequent coloring stages. This is carried on until the modularity gain between successive iterations becomes negligible.
4) Graph rebuilding: Between two successive phases, the community assignment output of the completed phase is used to construct the input graph for the next phase. This is done by representing all communities of the completed phase as "vertices" and accordingly introducing edges, identical to the manner in which it is done in the serial algorithm. This step is also implemented in parallel as described in Section 3.5.5.

We note here that the above parallel algorithm, with the exception of coloring heuristic, is stable in that it always produces the same output regardless of the number of cores used. When coloring is applied, the use of multiple threads within a given iteration could potentially vary the order in which decisions are made, thereby leading to potential variations in the output. In our experiments, we found the magnitudes of such variations to be negligible.

3.5.5 Implementation

We implemented our parallel heuristics in C++ and OpenMP. It is to be noted that the heuristics themselves are agnostic to the underlying parallel architecture. There are a few implementation level variations to Algorithm 1. In Algorithm 1 the modularity calculation happens in lines $16-17$. In our actual implementation we do not explicitly calculate the intra- and inter-community edges required for modularity calculation. Instead we pre-aggregate these values in steps $7-14$ as the net modularity gains are being calculated for each vertex. This saves significant recomputation. Secondly, to update the source and target communities for each vertex i, we use intrinsic atomic operations `__sync_fetch_and_add()` and `__sync_fetch_and_sub()`; Additional version that used OpenMP's synchronization technique is also included.

We use a compressed storage format for graph data structures that store the adjacency lists for all the vertices in a contiguous memory location. Specific memory pointers for each vertex is maintained in a separate list. This format enables efficient access to neighborhood information for each vertex. We use the C++ STL `map` data structure to store the set of unique clusters that a vertex is connected to (i.e., neighboring communities). The number of possible choices is upperbounded by the degree of a vertex initially and depending on how fast the algorithm converges from iteration to iteration, the number of choices decreases. Since this step appears in the

computation for every vertex, we also experimented with several alternatives including the use of C++ STL `unordered_map` data structure, but did not find any significant improvements in performance.

The step to rebuild the graph between consecutive phases is implemented in parallel and serial in parts. This is achieved in a sequence of steps. Assume that the phase transition is between phase $i-1$ to i. We use G_{i-1} and G_i to refer to the graphs input to phases $i-1$ and i respectively. i) First, the set of vertices in G_i is constructed from the communities output from phase $i-1$. Since many communities which existed at the start of phase $i-1$ could have become empty by the end of that phase, we first renumber of communities numerically, using only non-empty communities. This step is currently implemented in serial, although our future plan is to explore a parallelization using prefix computation-based approach. ii) In the next step, a STL `map` structure is allocated for every new vertex in G_i to concisely store the set of neighboring communities attached to it. This step is parallel. iii) In the following step, all edges in G_{i-1} are traversed in parallel. If an edge is an intra-community edge, then the weight for the corresponding edge (connecting the community vertex to itself) in G_i is updated. Alternatively, an inter-community edge leads to an update to each of the two corresponding community vertices in G_i. A simple way to avoid dead lock is to only store one direction on information.

Our implementation is named *Grappolo*[2]. The software is available for download under the BSD 3-Clause license from here: http://hpc.pnl.gov/people/hala/grappolo.html.

3.5.6 Analysis

Within each iteration (refer to Algorithm 1), the vertices are scanned in parallel, and for every vertex their vertex neighborhood is scanned first to curate the set of distinct

[2]Italian word meaning a cluster (of grapes)

neighboring communities (steps $10-11$). Subsequently, the main step of modularity gain calculation is performed only for each distinct neighboring community (step 12), which is equal to vertex degree initially but is expected to rapidly reduce as the iterations progress. Consequently, the worst-case runtime complexity *per* iteration is $O(\max\{\frac{M+n\cdot\lambda}{p}, \lambda_{max}\})$, where p denote the number of processing cores, λ is the average (unweighted) degree of a vertex and λ_{max} is the maximum (unweighted) degree of a vertex. The space complexity is linear in the input for shared memory implementation (i.e., $O(m+n)$). The above analysis assumes that the entire collection of vertices is processed in one parallel step within each iteration. With the application of coloring, parallelism is limited to each color set, implying the number of color sets to correspond to the number of parallel steps within each iteration.

3.6 Experimental evaluation

3.6.1 Experimental setup

The test platform for our experiments is an Intel Xeon X7560 server with 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 cache per socket. Each socket has 64 GB of DDR3 memory with a peak bandwidth of 34.1 GB per second. The software was compiled with GCC version 4.8.2 using `-Ofast` option. We also enabled non-uniform memory distribution using `numactl` command and enabled thread binding by using `GOMP_CPU_AFFINITY` environment variable. The thread binding variable was configured to place the threads across the system as evenly as possible with the goal of maximizing the memory bandwidth. All experiments were run using one thread

per core.

We tested our heuristics on 11 different real world input graphs, which are summarized in Table 4.2. With the exception of inputs labeled "MG1" and "MG2", all other inputs were downloaded from the DIMACS10 challenge website *The 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering*; Bader et al., 2012, and the University of Florida sparse matrix collection Davis and Hu, 2011. "MG1" and "MG2" are graphs constructed for two different ocean metagenomics data, using the construction procedure described in Wu, Kalyanaraman, and Cannon, 2012.

Table 3.1 Input statistics for the real world networks used in our experimental study. "RSD" represents the relative standard deviation of vertex degrees for each graph. It is given by the ratio between the standard deviation of the degree and its mean.

Input graph	Num. vertices (n)	Num. edges (M)	Degree statistics (λ)		
			max.	avg.	RSD
CNR	325,557	2,738,970	18,236	16.826	13.024
coPapersDBLP	540,486	15,245,729	3,299	56.414	1.174
Channel	4,802,000	42,681,372	18	17.776	0.061
Europe-osm	50,912,018	54,054,660	13	2.123	0.225
Soc-LiveJournal1	4,847,571	68,475,391	22,887	28.251	2.553
MG1	1,280,000	102,268,735	148,155	159.794	2.311
Rgg_n_2_24_s0	16,777,216	132,557,200	40	15.802	0.251
uk-2002	18,520,486	261,787,258	194,955	28.270	5.124
NLPKKT240	27,993,600	373,239,376	27	26.666	0.083
MG2	11,005,829	674,142,381	5,466	122.506	2.370
friendster	51,952,104	1,801,014,245	8,603,554	69.333	17.354

The input graphs were tested using multiple variants of our implementation that use different combination of the proposed heuristics. These variants are as follows:

- **baseline:** represents our parallel implementation with only the Minimum Labeling (ML) heuristic;
- **baseline+VF:** represents the baseline implementation with the application of the Vertex Following (VF) heuristic in a preprocessing step. There were a few inputs (viz., Channel, MG1, MG2) for which their single degree vertices had already been pruned off when their respective graphs were generated, and consequently their baseline runs are equivalent to their baseline+VF runs[3]. For the remaining inputs, VF preprocessing was run only once, prior to the start of the first phase;
- **baseline+VF+Color:** represents the baseline implementation with the application of both the VF and coloring heuristics (in that order). Coloring was used as a preprocessing step for multiple phases until either the number of input vertices reduced below a preset cutoff (100K used for this paper) or the net modularity gain between phases is less than the user-defined threshold (10^{-2}). Once either of these conditions is met, the implementation does not perform coloring anymore and the remaining phases are executed using a default net modularity gain threshold of 10^{-6} for termination.

3.6.2 Performance evaluation

To assess the effectiveness of our parallel heuristics, we studied how quickly a given algorithm converges to its final modularity (as a function of the number of iterations)

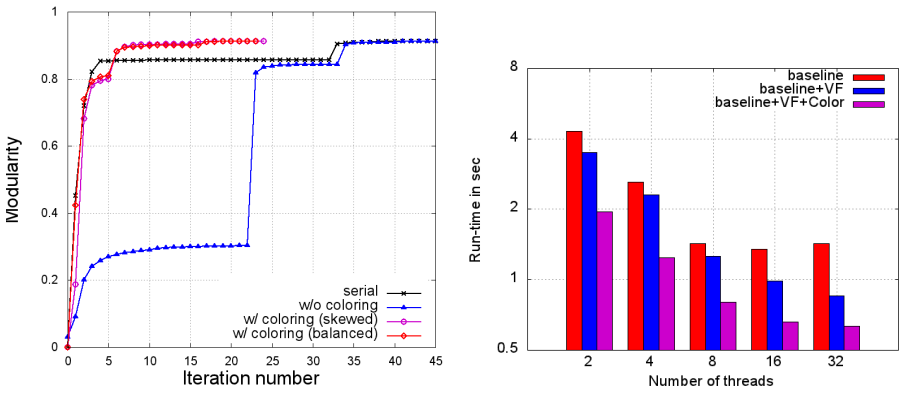
[3]For this reason, we show only their baseline+VF runs in their respective charts.

and compared it against the convergence rate of the corresponding serial Louvain[4] execution. We also compared the difference in runtimes and final modularities output by the individual approaches. Figures 3.3-3.6 show the evolution of modularity from the first iteration of the first phase to the last iteration of the last phase for all the 11 test inputs, and the parallel runtimes as a function of the number of cores.

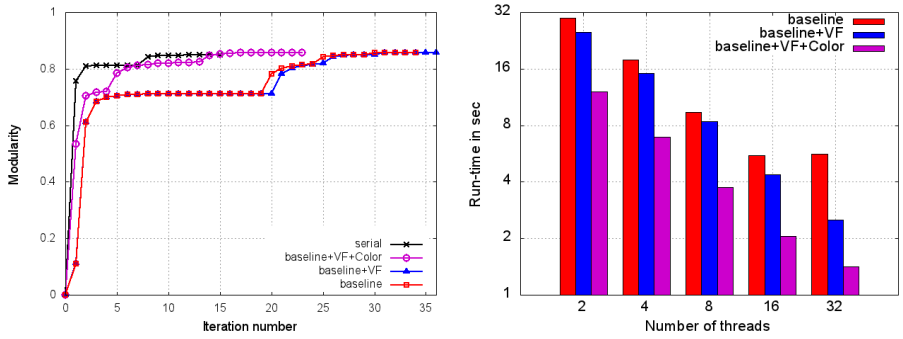
Effectiveness of the VF heuristic: The run-time charts in Figures 3.3-3.6 show the effectiveness of the VF heuristic in reducing run-time relative to our baseline implementation. The reduction in run-time can be attributed to the reduction in the number of vertices to be processed within each iteration. However, the effectiveness of the VF heuristic is also tied to the number of single degree vertices in the original input graph. While our results show that VF is able to produce run-time savings in most input cases, there were two exceptions: Europe-osm (Figure 3.4d) and Rgg_n_2_24_s0 (Figure 3.5g), for which the run-time was observed to increase. Upon further investigation, we found that the application of VF for these two inputs indeed caused a reduction in the time spent per iteration as expected; however, it also led to prolonging the convergence of the algorithm within the intial phases — i.e., it led to an increase in the number of iterations within a phase.

This delay in convergence within a phase shows a potential drawback of the VF heuristic on some input cases that can be intuitively explained as follows: consider a chain of "hub" nodes where the hubs are individually connected to a number of single degree vertices ("spokes"). In such cases, the compacted representation that results from the application of VF would have more incentive to continue in the current phase by gradually collapsing the chain into larger communities and achieving smaller gains in modularity that still surpass the minimum required cutoff. This results in

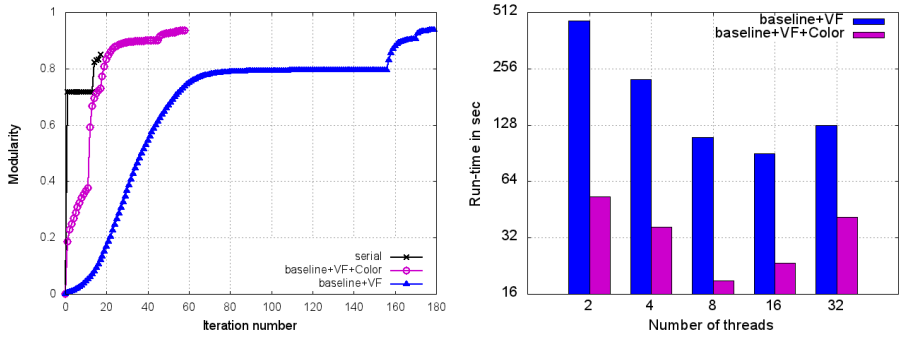
[4]All references to the "serial" implementation in the experimental results section corresponds to the original Louvain implementation available from *findcommunities*.



(a) Input: CNR

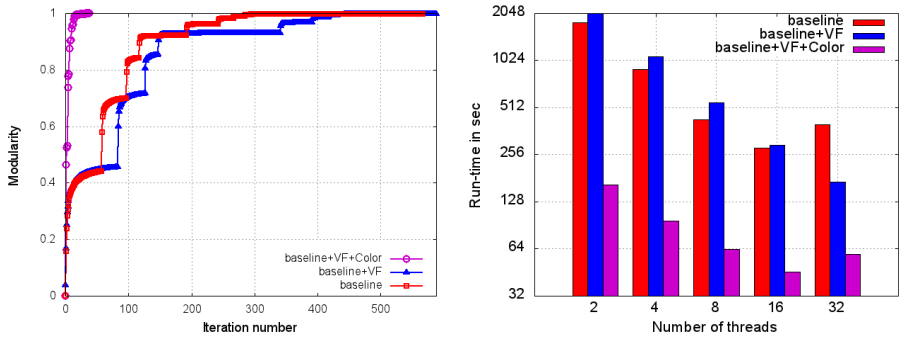


(b) Input: coPapersDBLP

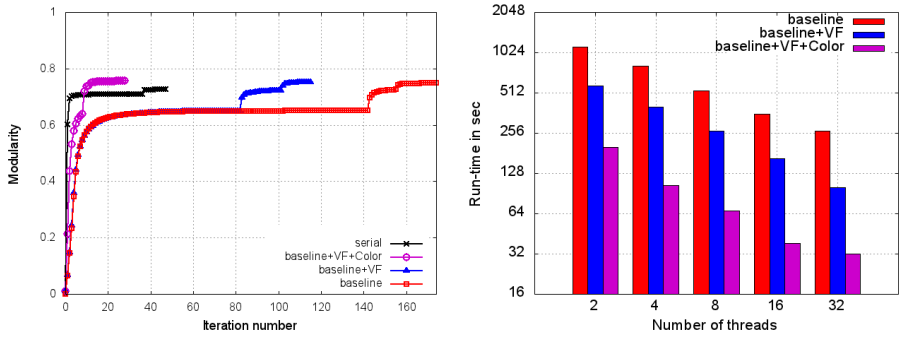


(c) Input: Channel

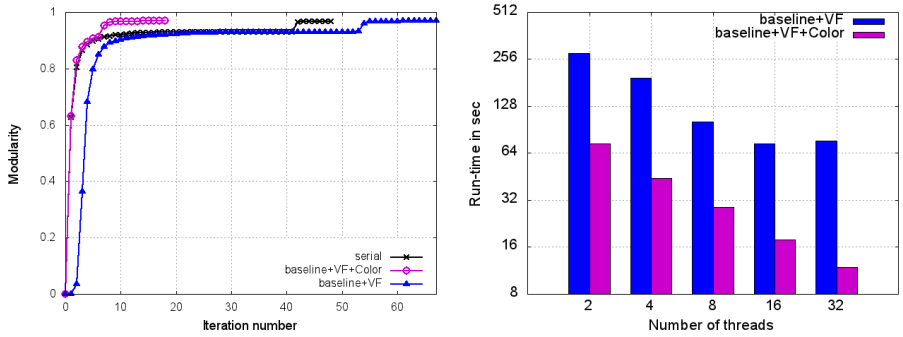
Figure 3.3 Charts showing the evolution of modularity (left column) and the parallel runtime performance (right column) for each test input. The steep climbs in modularity visible in the modularity curves correspond to phase transitions. Also shown for comparison are the corresponding performance of the serial algorithm.



(d) Input: Europe-osm



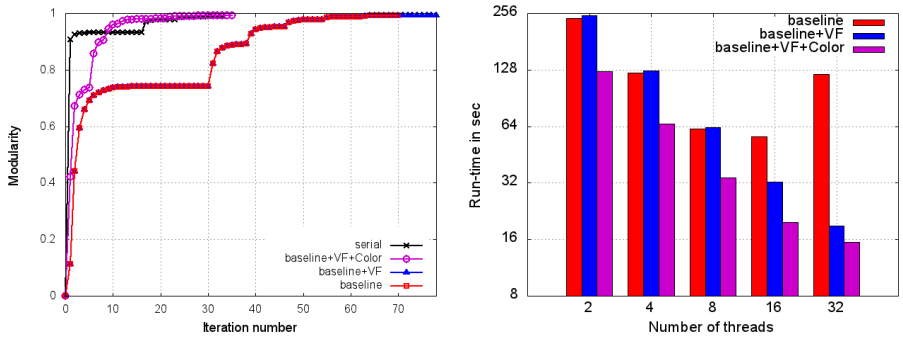
(e) Input: Soc-LiveJournal1



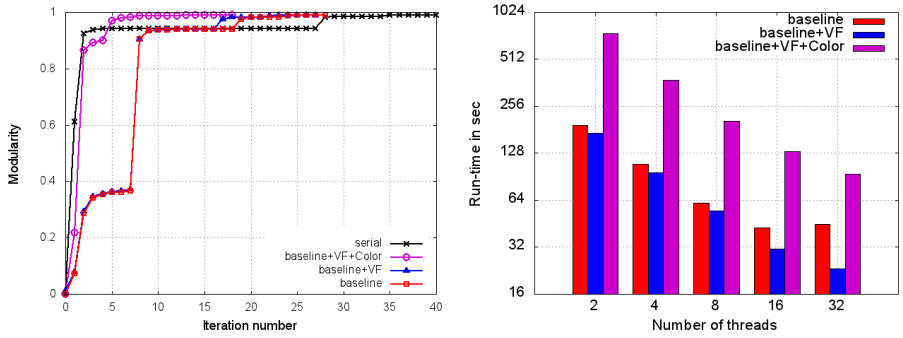
(f) Input: MG1

Figure 3.4 Charts showing the evolution of modularity (left column) and the parallel runtime performance (right column) for each test input.

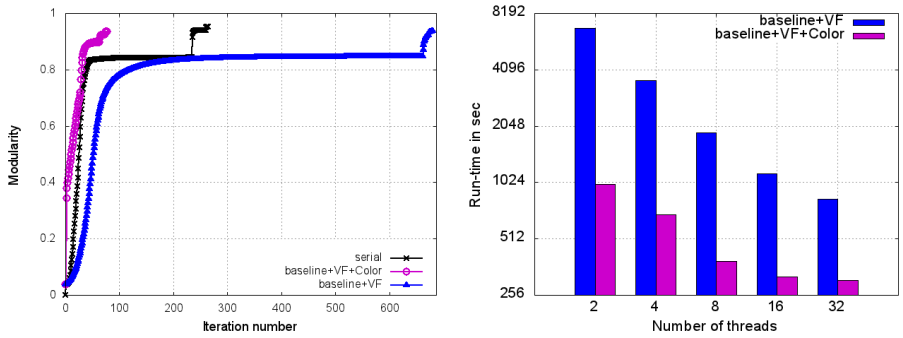
prolonging the termination of the current phase. In contrast, if we were to omit applying the VF heuristic on the input graph, then a hub node could potentially migrate into one of its spokes' communities and when that happens, there is an increased probability that the algorithm terminates the current phase sooner due



(g) Input: Rgg_n_2_24_s0



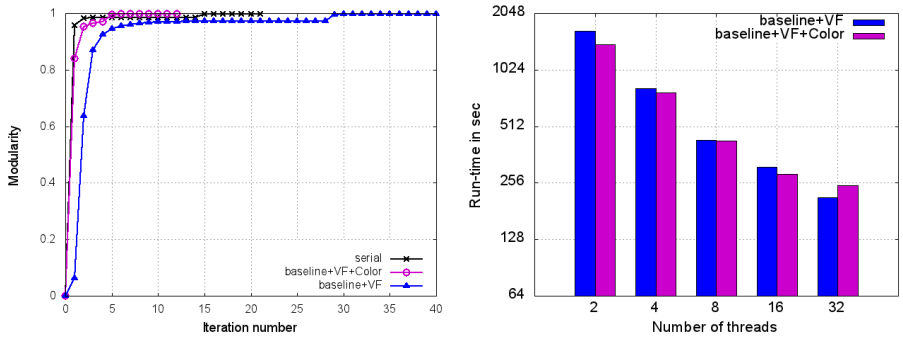
(h) Input: uk-2002



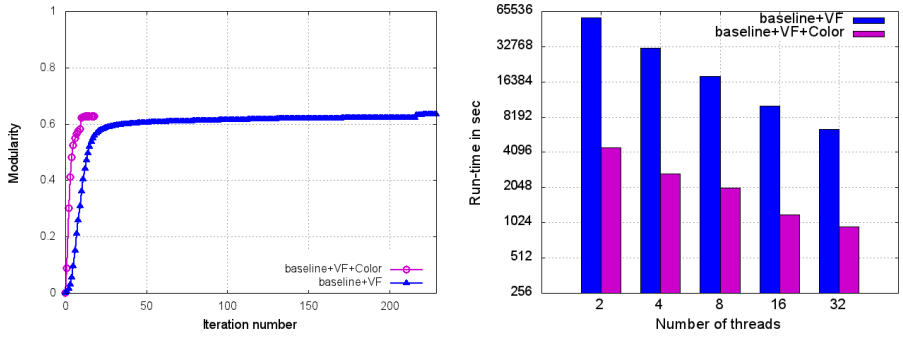
(i) Input: NLPKKT240

Figure 3.5 Charts showing the evolution of modularity (left column) and the parallel runtime performance (right column) for each test input.

to negligible modularity gain. While the resulting final modularity figures could be slightly lower than obtained with the application of VF, the gains in runtime may be more pronounced, which is what we observed for the two inputs Europe-osm and Rgg_n_2_24_s0. It is to this end, that the proposed extension of the VF heuristic that



(j) Input: MG2



(k) Input: friendster

Figure 3.6 Charts showing the evolution of modularity (left column) and the parallel runtime performance (right column) for each test input.

also seeks to compress paths (see discussion at the end of Section 3.5.3) could aid in obtaining a better balance between run-time benefit and modularity gain.

Effectiveness of coloring: The design intent of coloring is to reduce the number of iterations required to converge on a solution, and in the process reduce the time to solution. However, a potential drawback of coloring is reduced parallelism within each iteration; more specifically, the presence of numerous small color sets could result in an under-utilization of threads. In our experimental results, we found coloring to be highly effective in reducing both the number of iterations *and* the overall time to solution. The run-time improvements of *baseline+VF+coloring* over *baseline+VF* were anywhere from $\sim 3.48\times$ to $16.52\times$. However, the run-time improvements were either negligible in the case of MG2 (Figure 3.6j) or negative in the case of uk-

2002 (Figure 3.5h). These observations correlate with the highly skewed color size distributions for these two graphs. For instance, 943 colors were used for uk-2002 in the first phase and the color sets had a high Relative Standard Deviation (RSD) of 18.876 in their sizes. We are exploring an alternative approaches to create balanced coloring sets that are targeted at addressing this performance issue. For all other inputs, however, the benefit of coloring is evident in the drastically reduced number of iterations for convergence and subsequent savings in the time to solution. These results also show the combined effect of applying both VF and coloring heuristics, as they yield an additive net gain in performance.

Scaling and run-time results

Figure 3.7 shows the speedup curves for our parallel implementation (*baseline+VF+Color*). Two speedup curves are shown: a) *relative speedup*, which calculates the speedup of the parallel execution over the corresponding 2-thread run (discussed in this section); and b) *absolute speedup*, which is the speedup calculated over the corresponding serial Louvain implementation's execution *findcommunities* (to be discussed in Section 3.6.2).

The relative speedup curves show that on most inputs, the parallel implementation continues to deliver increasing speedups up to 32 threads, although the speedups become sub-linear beyond 8 threads. While the input sizes play a role, it can be observed from the results that the size alone is not the sole determinant of performance. For instance, the implementation achieves higher peak relative speedups ($\sim$8$\times$) on some of the smaller inputs such as coPapersDBLP (540K vertices, 15M edges) and Rgg_n_2_24_s0 (16M vertices, 132M edges) than on a larger input such as NLPKKT240 (51M vertices, 1.8B edges). Parallel performance is affected by a combination of input characteristics and the serial bottlenecks within the parallel implementation.

Inputs Channel and NLPKKT240 have a low RSD in vertex degree distribution (Table 4.2) and also have a poor community structure (reflected in their low modularity scores). This combination leads to an increased number of iterations in the initial phases, as the algorithm continues within a phase albeit incremental modularity gains. The increased number of iterations in the first phase in particular (where the graph size is the largest) adversely affects on performance. This is because within each iteration the step to recalculate the new modularity score involves updating community structures (internal edge and incident edge counts); and as the number of communities begins to reduce in the later iterations of a phase, more parallel overhead due to locking is incurred.

In contrast, consider the input *Rgg_n_2_24_s0* which also has a low RSD in its vertex degree distribution but for which a superior parallel performance is observed. This input is a random geometric graph, which despite its uniform degree distribution, is also known to have a high community structure (reflected by its high modularity score). This attribute allows the algorithm to rapidly converge within the first phase, thereby aiding better overall parallel performance.

Another significant contributing factor affecting parallel performance is the time taken to rebuild the graph between consecutive phases. To analyze this effect, we recorded the breakdown of total run-time by the different phases of the parallel algorithm (described in Section 3.5.4). Figure 3.8 shows the breakdown - viz. time to rebuild the graph between phases (VF cost is included here), time to perform coloring, and the remaining time attributed to performing the iterations ("clustering"). The charts (shown for four representative inputs) explain the discrepancies in scaling among the inputs. For Rgg_n_2_24_s0 and MG2, we can see that the time spent in the main clustering iterations dominates, which is desirable from a scaling point of view. However, for inputs Europe-osm and NLPKKT240, an increasing portion of time is

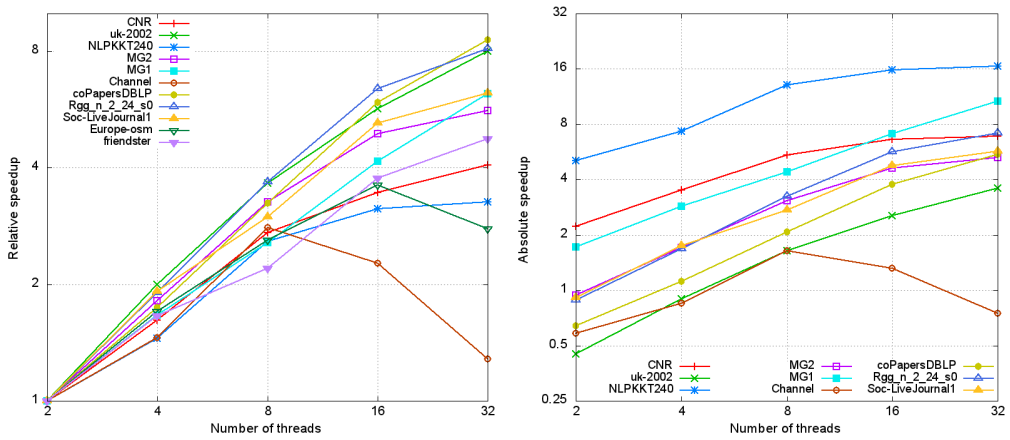
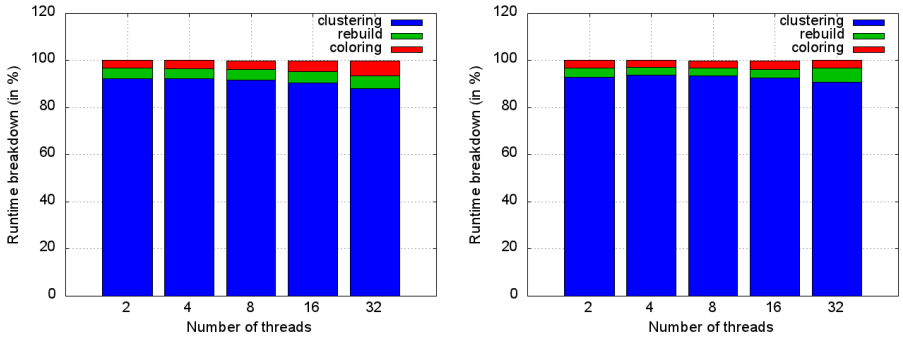


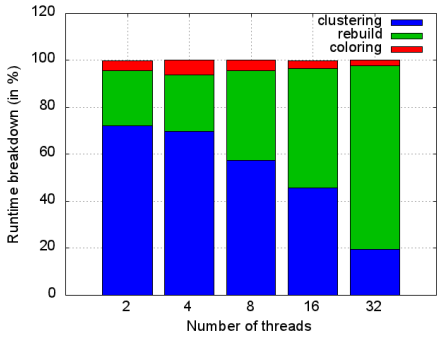
Figure 3.7 Speedup charts for our parallel implementation, *Grappolo*. The chart on left shows the relative speedup of the parallel implementation using the 2-thread run as the reference. The chart on the right shows the absolute speedup — i.e., relative to the serial Louvain implementation *find-communities*. All speedups are calculated using the *baseline+VF+Color* implementation of *Grappolo*. Note that in the absolute speedup chart, curves for Europe-osm and friendster are not shown because the serial Louvain implementation failed to complete on these two inputs.

being spent in the rebuild phase with an increase in the number of cores. Given that our current implementation of the rebuild phase has serial bottlenecks (as explained in Section 3.5.5), the speedups achieved for these inputs become sub-linear for higher number of cores. Figure 3.9 confirms these observations about the rebuild phase. More specifically, for inputs Europe-osm and NLPKKT240, the first phase ends in a low modularity (0.533470 and 0.038107 respectively), which implies that a dominant fraction of the edges remain as inter-community edges. In the graph rebuild phase, each such edge corresponds to two locks (one for each community) affecting parallel performance. In contrast, input MG2 ends with a high modularity score of 0.969587 resulting in an improved performance during the rebuild phase as well.

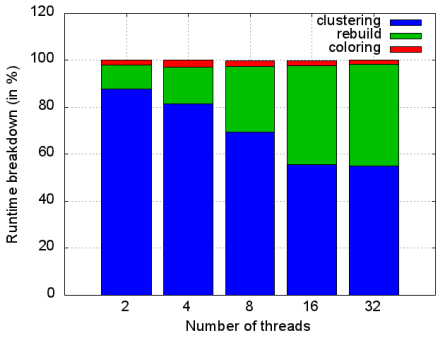


(a) Input: Rgg_n_2_24_s0

(b) Input: MG2



(c) Input: Europe-osm



(b) Input: NLPKKT240

Figure 3.8 Breakdown of the parallel run-times by the different steps of the algorithm - viz. coloring, time to perform the graph transformations between phases, and the time spent in the iterations. The runs correspond to the *baseline+VF+Color* implementation.

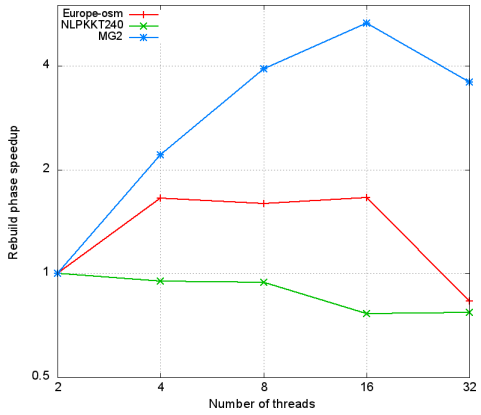


Figure 3.9 Chart showing the speedup curves for the graph rebuilding phase of our parallel algorithm.

Table 3.2 Comparison of the modularities and run-times achieved by our parallel implementation *baseline+VF+Color* (using 8 threads) against the corresponding values achieved by the serial Louvain implementation *find-communities*. All runs were performed on the same test platform described under Experimental Setup. The "N/A' entries denote cases where the serial Louvain implementation did *not* complete (i.e., crashed). It is to be noted that the serial Louvain implementation is a 32-bit implementation.

Input	Output modularity		Run-time (in sec)		
	Parallel	Serial	Parallel (8 threads)	Serial	Speedup (8 threads)
CNR	0.912608	**0.912784**	0.8	4.3	5.37×
coPapersDBLP	**0.858088**	0.848702	3.7	7.7	2.08×
Channel	**0.933388**	0.849672	21.2	30.9	1.45×
Europe-osm	**0.994996**	N/A	63.4	N/A	N/A
MG1	**0.968723**	0.968671	28.8	126.6	4.39×
uk-2002	0.989569	**0.9897**	210.3	335.9	1.59×
MG2	0.998397	**0.998426**	457.8	1313.7	2.86×
NLPKKT240	0.934717	**0.952104**	388.4	5077.2	13.07×
Rgg_n_2_24_s0	**0.992698**	0.989637	34.2	111.1	3.24×
Soc-LiveJournal	**0.751404**	0.726785	67.05	182.7	2.72×
friendster	**0.626139**	N/A	2036.8	N/A	N/A

Comparison to serial Louvain

We also comparatively evaluated the performance of our parallel implementations proposed in this paper against the publicly available serial Louvain distribution *find-communities*. Figure 3.7 shows the absolute speedup achieved over the serial implementation for 9 out of the 11 inputs. (For the remaining two inputs, Europe-osm and friendster, the serial implementation failed to run.) Table 3.2 compares the final

modularities achieved by both implementations and also the corresponding run-times. For 7 out of the 11 inputs, our parallel implementation delivers higher modularity compared to the serial implementation in shorter time to solution. For example, this difference is as much as >0.1 for coPapersDBLP and > 0.08 for Channel. Even in 3 out of the 4 cases where the serial implementation delivers higher modularity, the modularities reported by both methods agree up to the first three decimal places. Note that the heuristic nature of the algorithm combined with the parallel ordering of vertices which could differ from the serial ordering imply that serial and parallel results cannot be guaranteed to be identical. Our results demonstrate that parallelization is at least capable of preserving (if not surpassing) output quality for most of the inputs tested.

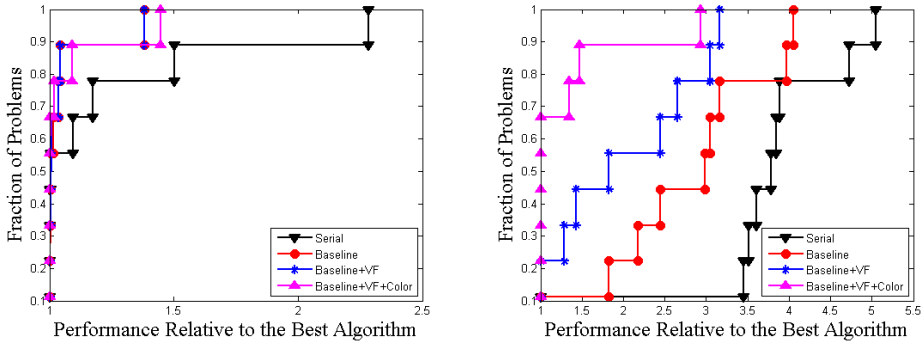
As for the run-times, our parallel implementation delivers absolute speedups in the range of 1.45× to 13.07× using 8 threads. Larger speedups were observed using more number of threads, as can be observed from the absolute speedup chart in Figure 3.7. A top speedup of 16.51× was observed for the NLPKKT240 input using 32 cores. The two cases where we observe low speedups — Channel (1.45×) and uk-2002 (1.59×) — represent two different cases. For the Channel input, observe from Table 4.2 that the degree distribution is highly uniform. This could cause vertices to migrate to any one of the neighboring communities and therefore the vertex ordering is expected to have a more pronounced effect on the convergence rate. It is for this reason that the serial implementation, which uses an arbitrary ordering, converges faster albeit with a lower modularity, while our parallel implementation with coloring takes more iterations to converge and does so with a higher modularity. For uk-2002, the skew in the color set size distribution is the reason behind low speedup (as was explained earlier in the section).

Performance charts and qualitative evaluation

Figure 3.10 shows the relative performance profiles among the three parallel implementations – *baseline*, *baseline+VF*, and *baseline+VF+Color* – along with the serial Louvain implementation for the collection of inputs tested. For plotting these performance charts, we used results from all 9 inputs for which we had results from both serial and parallel implementations. The X-axis represents the factor by which a given scheme fares relative to the best performing scheme for that particular input. The Y-axis represents the fraction of problems (i.e., inputs). The closer a heuristic curve is to the Y-axis the more superior its performance is relative to the other schemes over a wider range of inputs. Also, in these performance charts, the order in which inputs appear along each curve is strictly a function of that corresponding heuristic's relative performance to the other schemes — i.e., the points along a curve are sorted from the corresponding heuristic's best to worst performing inputs. Thus, the charts illustrate the relative performance of each scheme over other schemes for the *collection* of 9 inputs tested (as opposed to the individual inputs).

The following observations can be made from the two performance charts. The *baseline+VF+Color* shows an overall run-time performance advantage over all other schemes. For instance, consider the run-time curve for *baseline+VF+Color* in Figure 3.10b. This implementation outperforms all other heuristics for about 70% of the problems, about 1.5× worse compared to a best performing implementation for 20% of the problems, and 3× worse than the best for 10 percent of the problems. Similarly, the serial implementation is the slowest ranging from $2 \times -5\times$ relative to other best performance schemes. From a modularity standpoint, all parallel heuristics perform comparably to serial method across the input set.

Qualitative comparison: In addition to comparing modularities, we also com-



(a) Modularity profile

(b) Run-time profile

Figure 3.10 Relative profile of performance for three combinations of heuristics: The relative performance of different heuristics and serial implementation for the test problems with respect to the best algorithm for a given problem. Europe-osm and friendster are not included in the comparison because the serial Louvain implementation crashes on those inputs. Final modularity scores are shown in the figure on left (part a), and run-times are shown on the right (part b). Run-time results from 32 thread runs were used to plot curves for the parallel heuristics. It is to be noted that the longer a heuristic's curve stays near the Y-axis the more superior its performance relative to the other schemes over a wider range of inputs.

pared the sets of communities by their composition generated by the parallel and serial implementations. The methodology for comparison is as follows. Let S denote the set of communities generated by the serial implementation; and P denote the set of communities generated by one of our parallel implementations — we used results from the *baseline+VF+Color* for this purpose. Treating the serial output as the "benchmark" we compared the parallel output against it as follows. Any vertex pair (u, v) can be categorized into one of the four following bins:

- **True Positive (TP):** if u and v belong to the same community in both partitions;
- **False Positive (FP):** if u and v belong to the same community only in partition P;

- **False Negative (FN):** if u and v belong to the same community only in partitions S;

- **True Negative (TN):** if u and v belong to two different communities in both partitions;

Based on the above measures, more qualitative measures, viz. specificity (SP), sensitivity (SE), overlap quality (OQ) and Rand Index, can be calculatated as follows: $SP = \frac{TP}{TP+FP}$, $SE = \frac{TP}{TP+FN}$, $OQ = \frac{TP}{TP+FP+FN}$, and Rand index $= \frac{TP+TN}{TP+FP+FN+TN}$.

Note that if both results match identically, all these measures will evaluate to 100%. Also note that this comparison takes $\Theta(n^2)$ time because there are $\binom{n}{2}$ pairs. For this reason, we performed this qualitative comparison only for two of the inputs — CNR and MG1.

Table 3.3 shows the results of our comparative study. There are two observations that one can make from these results. First, as can be expected, the partitioning produced by the two methods are different. However, the fact that there is *no* explicit biasing toward false positives or false negatives implies that the cores of communities captured by both methods agree to a large extent — the OQ values reflect the degree of this agreement. Secondly, given that these two partitioning yield nearly identical modularities imply that the vertex pairs consistently grouped by both schemes (i.e., True Positives) contribute to the bulk of the modularity score.

Table 3.3 Qualitative comparison between the parallel and serial community outputs by their composition.

Input	SP	SE	OQ	Rand index
CNR	83.41%	89.71%	76.13%	99.42%
MG1	99.60%	99.83%	99.43%	100.00%

3.6.3 Effect of multiphase coloring

Table 3.4 Comparative results showing the effect of using coloring for only the first phase input vs. for multiple phases of the parallel algorithm. The multi-phase coloring scheme is same as the *baseline+VF+Color* scheme. All run-times are reported in seconds for runs corresponding to two threads.

Input	First phase coloring		Multi-phase coloring	
	[Min.,Max.] Modularity	Run-time (#iter)	[Min.,Max.] Modularity	Run-time (#iter)
Channel	[0.9344,0.9352]	103.22 (96)	[0.9304,0.9333]	52.96 (58)
uk-2002	[0.9895,0.9895]	670.12 (18)	[0.9894,0.9895]	748.15 (18)
Europe-osm	[0.9988,0.9988]	759.94 (306)	[0.9988,0.9989]	118.97 (38)
MG2	[0.9984,0.9984]	1422.75 (14)	[0.9984,0.9984]	1397.90 (12)

Coloring can be potentially applied to preprocess the input for any phase of the algorithm. However, the time spent coloring is an overhead and a colored graph exposes less parallelism. Therefore, it can be expected that the benefits of coloring, which is to hasten convergence, is expected to diminish as phases progress and the transformed graph becomes smaller. It is for this reason we used a scheme in which coloring is applied until either the number of input vertices reduces below a cutoff (100K for our experiments) or the net modularity gain between phases diminishes below a relatively higher threshold (10^{-2}) as described in Section 3.6.1. However, to clearly demonstrate the effect of coloring multiple phases, we devised an alternative implementation in which coloring is applied only to the first phase input. The goal was to observe differences in reported modularity and run-times between the two schemes.

Table 3.4 shows the effect of coloring single phase to multiphase. Inputs picked

are those for which at least two phases of coloring was applicable. For the other inputs, the results are identical between single phase and multiphase coloring schemes. The results demonstrate the benefit of multi-phase coloring as it produces highly comparable modularities over multiple experiments while reducing time-to-solution, for all inputs except uk-2002.

3.6.4 Effect of varying the modularity gain threshold

Table 3.5 Table showing the effect of varying the modularity gain threshold. Two sets of experiments were performed, each running the *baseline+VF+Color* implementation, while one using 10^{-2} and another 10^{-4} as the value for the modularity gain threshold used within the colored phases.

Input	Threshold = 10^{-4}		Threshold = 10^{-2}	
	[Min.,Max.] Modularity	Run-time (#iter)	[Min.,Max.] Modularity	Run-time (#iter)
CNR	[0.9125,0.9125]	5.00 (48)	[0.9125,0.9126]	1.77 (24)
CoPaperDBLP	[0.8555,0.8577]	16.17 (27)	[0.8570,0.8580]	10.64 (23)
Channel	[0.9423,0.9485]	816.79 (282)	[0.9304,0.9333]	52.96 (58)
Europe-osm	[0.9989,0.9989]	250.62 (56)	[0.9947,0.9949]	125.35 (17)
MG1	[0.9687,0.9687]	271.23 (41)	[0.9687,0.9687]	73.80 (18)
Rgg_n_2_24_s0	[0.9926,0.9927]	227.03 (52)	[0.9926,0.9926]	118.21 (35)
uk-2002	[0.9895,0.9896]	1768.73 (22)	[0.9894,0.9895]	748.15 (18)
Nlpktt240	[0.9426,0.9476]	3563.41 (147)	[0.9319,0.9347]	880.94 (78)
MG2	[0.9984,0.9984]	2652.37 (16)	[0.9983,0.9983]	1312.44 (7)

We also studied the effect of varying the modularity gain threshold used within the coloring phases. Using a larger value of threshold may prompt phase transitions to happen earlier (and possibly faster convergence) but at the possible expense of the

final output modularity. On the other hand, a smaller value could help improve gains within phases but also could prolong phase transitions and eventual completion. Two sets of experiments were performed, using values of 10^{-2} and 10^{-4} for the threshold and the results are summarized in Table 3.5. As can be observed, the modularities achieved by both schemes are highly comparable, while there is a marked run-time advantage if the threshold is higher. This study shows that the run-time benefit of using a higher threshold outweighs the qualitative gains of using a lower threshold, at least for the threshold values compared.

From a modularity standpoint, coloring has a more pronounced effect than the threshold used. The charts in Figure 3.3a,d,e illustrate this effect — observe that coloring provides substantial increases in the modularity at the initial phases of the algorithm *before* a finer modularity threshold could take effect in the later phases.

3.7 Extensions to Community Detection

In this section we introduces two types of extensions to our parallel community detection method. In Section 3.7.1, we present variants of our implementation which are aimed at exploiting a diminishing returns property of our algorithm. In Section 3.7.2, we present our preliminary ideas for extending our community detection algorithmic framework to study dynamic graphs. We refer to the original implementation of our *Grappolo* community detection framework as the *baseline* version.

3.7.1 Synchronization-based Extensions

Full Synchronization

In Section 3.5, we introduced heuristics to construct a non-synchronized version of parallel community detection, with the goal of improving concurrency during com-

putation. In contrast, we also implemented a fully-synchronized version of parallel community detection, with the goal of improving the convergence rate of the iterative community detection process. The main difference between the two versions is that the fully-synchronized version uses the most recent community information available of any neighboring vertex while making a migration decision (line 10-14 in Algorithm 1). This is achieved by locking all neighboring vertices so that they do *not* migrate or get updated during this processing. Since the fully-synchronized version uses the most current neighborhood information, and involves locking, it has the potential to decrease number of iterations required to converge, while also running the risk of increasing the runtime cost of each iteration.

Early Termination

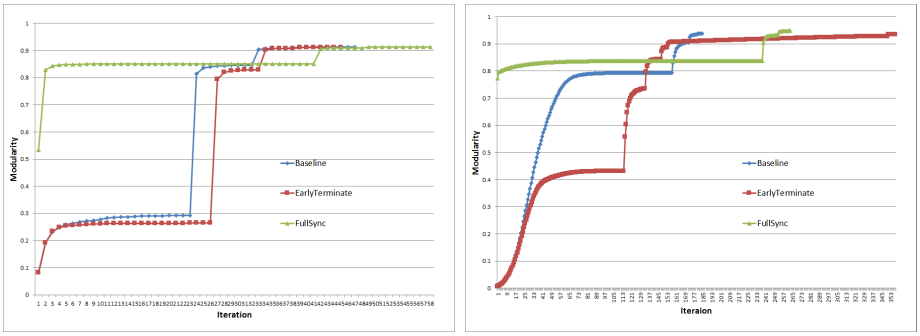
Our iterative community detection algorithm has a key property that can be described through the following observation: As shown in Figures 3.3-3.6, for all inputs we tested, the gain in modularity plateaus after a few initial iterations. This is because the number of community updates to vertices drastically reduces during the later iterations. This observation suggests that most of the computation that is performed in the later iterations, are likely to devoted toward processing vertices that have already finalized their community affiliation—thereby, leading to wasteful work.

Therefore, to reduce wasteful work and in the process, improve run-time performance in practice, we take advantage of this diminishing returns property as follows: We keep a counter at every vertex to denote the number of iterations that have elapsed since its last successful migration (i.e., change in community). If the counter exceeds a certain threshold, then we "terminate" that vertex—i.e., we (optimistically) stop considering that vertex during any subsequent iteration of that phase; implying that the vertex is locked into that community. This is obviously a heuristic intended to

improve performance by reducing the number of vertices that need to be processed at the later stages of execution.

Experimental Results

We compared the baseline against the two extensions described above—viz., Full Synchronization and Early Termination. As Figure 3.11 shows, the fully-synchronized version has a faster convergence rate than the baseline, while the early termination version exhibits a slower convergence. However, slower convergence does *not* necessarily imply a higher runtime. As shown in Table 3.6, the runtime for the Fully-Synchronized version is consistently higher than either of the two other versions, despite faster convergence. This is because of the locking overhead associated with the full synchronization scheme within each iteration. Conversely, the Early Termination version shows a reduced runtime compared to the baseline version for those graphs that need a large number of iterations. This is a result of the lower number of vertices being processed within each iteration of the Early Termination version.



(a) Input: CNR (b) Input: Channel

Figure 3.11 Charts showing the evolution of modularity for the different versions (viz. baseline, Early Termination and Fully-Synchronized) of our community detection method on 16 threads.

Table 3.6 Runtime statistics with 16 threads.

Input graph	Runtime			Modularity		
	Baseline	Early T.	FullSync	Baseline	Early T.	FullSync
CNR	3.04162	3.355803	9.956494	0.91242	0.91157	0.91284
Channel	27.640296	17.470792	148.105455	0.93831	0.9336	0.9479

3.7.2 Extension to Dynamic Graphs

So far we considered the community detection problem on standard graphs. In this extension, we present our preliminary ideas for extending our algorithmic framework to dynamic graphs. A *dynamic graph* is a graph that evolves over multiple time steps. Typically, we consider changes in the form of vertex/edge additions and vertex/edge removals.

Once presented with a dynamic graph, mining for communities from *across* the time steps becomes important, since the finer level changes in the dynamic graph can be effectively summarized through a higher level description of how communities evolve. For example, consider the example of a graph consisting of the people working in a research field as vertices, co-author relationships captured in edges (in a temporal manner), and the various collaborating sub-groups representing the communities. Given such a time-varying (dynamic) graph, and without any other prior knowledge, identifying and tracking communities in such a graph could not only capture the information on how individual collaborating groups evolve over a period of time (i.e., by either splitting or merging) but also can shed light on the start of a new sub-field or the dissolution of an older (and perhaps obsolete) sub-field.

We present a simple idea to extend our algorithmic framework to detect and track communities in a dynamic graph. The main idea of our approach is as follows: Let

$P_{t1} = \{C_1, C_2, \ldots C_k\}$, $P_{t2} = \{D_1, D_2, \ldots D_{k'}\}$ denote the set of all communities in a given partitioning of the vertex set V_{t1} in $G_{t1}(V, E, \omega)$ and V_{t2} in $G_{t2}(V, E, \omega)$, where $t1$ and $t2$ representing different time steps. We construct a bipartite graph $G_b(V_{left} = P_{t1}, V_{right} = P_{t2}, E = P_{t1} \times P_{t2},\ \omega = \{\frac{|C_i \cap D_j|}{|C_i \cup D_j|}, \forall i, j\})$, where communities in each graph become vertices and edges weights are defined as the fraction of vertices share between two communities across time steps.

The bipartite graph G_b is the representation of dynamic communities. We could then use maximum weighted bipartite graph matching (Halappanavar, 2009) or one of its variants (such as b-matching (Khan et al., 2016)) on G_b to detect and subsequently track changes in community structures across the different timesteps. The weighted matching formulations provide a way to track community merges and splits. In cases where communities shuffle their composition across timesteps, matching may not be adequate. Under such scenario, we could adopt bipartite community detection techniques such as the *biLouvain* method (Pesantez-Cabrera and Kalyanaraman, 2016).

Algorithm 1 The parallel Louvain algorithm for a single phase. The inputs are a graph ($G(V, E, \omega)$) and an array of size $|V|$ that represents an initial assignment of community for every vertex C_{init}.

1: **procedure** PARALLEL LOUVAIN($G(V, E, \omega), C_{init}$)
2: $ColorSets \leftarrow Coloring(V)$, where $ColorSets$ represents a color-based partitioning of V.
▷ If the coloring step is omitted, then it automatically implies that all vertices belong to the same color set.
3: $Q_{curr} \leftarrow 0$
4: $Q_{prev} \leftarrow -\infty$ ▷ Current & previous modularity
5: $C_{curr} \leftarrow C_{init}$
6: **while** `true` **do** ▷ Iterate until modularity gain becomes negligible.
7: **for each** $V_k \in ColorSets$ **do**
8: $C_{prev} \leftarrow C_{curr}$
9: **for each** $i \in V_k$ in `parallel` **do**
10: $N_i \leftarrow C_{prev}[i]$
11: **for each** $j \in \Gamma(i)$ **do** $N_i \leftarrow N_i \cup \{C_{prev}[j]\}$
12: $target \leftarrow \arg\max_{t \in N_i} \Delta Q_{i \to t}$
13: **if** $\Delta Q_{i \to target} > 0$ **then**
14: $C_{curr}[i] \leftarrow target$
15:
16: $C_{set} \leftarrow$ the set of non-empty communities corresponding to C_{curr}
17: $Q_{curr} \leftarrow$ Compute modularity as defined by C_{set}
18: **if** $|\frac{Q_{curr} - Q_{prev}}{Q_{prev}}| < \theta$ **then** ▷ θ is a user specified threshold.
19: `break` ▷ Phase termination
20: **else**
21: $Q_{prev} \leftarrow Q_{curr}$

CHAPTER 4

BALANCED COLORING

4.1 Problem Overview

Decomposing a computational task into constituent parts that can be executed simultaneously or identifying elements of composite data that can be safely updated simultaneously is a pervasive primitive in parallel computing. An 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.

In this chapter, we consider two types of graph coloring problems: *distance-1 coloring*, which is defined on a general (unipartite) graph, and, *partial distance-2 coloring*, which is defined on a bipartite graph. A distance-1 coloring of a general graph $G(V, E)$ is an assignment of colors to vertices such that any two adjacent vertices receive different colors. A partial distance-2 coloring of a bipartite graph $G_b = (V_1, V_2, E)$ is an assignment of colors to one of the vertex sets, say V_2, such

that any two vertices in V_2 that are two edges away from each other receive different colors. More formal definitions of these problems will be given in Sec. 4.2 and 4.5, but we remark at this point that distance-1 coloring is the "usual" graph coloring problem.

Standard formulations of the distance-1 and partial distance-2 coloring problems aim 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. They therefore permit cases where the color classes can be highly unbalanced. In fact, by their nature, most practical algorithms for the standard formulations of these graph coloring problems produce highly skewed color classes. This will be undesirable as the smaller color classes may *not* provide sufficient workload for parallel efficiency. In this chapter, we deal with the design, implementation and performance evaluation of algorithms for the distance-1 and partial distance-2 coloring problems that also require that color classes be *balanced* in their sizes. A preliminary version of this work that focused only on distance-1 coloring appeared in (Lu et al., 2015)

There is a body of work in the graph theory literature on *equitable* distance-1 colorings—a formulation in which color classes are required to be *perfectly* balanced—but little work exists on fast, practical, balanced coloring algorithms and their parallelization on contemporary and emerging platforms. Further, to the best of our knowledge, there exists no prior work on balanced partial distance-2 coloring. We seek to address these deficiencies.

The remainder of this chapter is organized as follows. We provide background and motivate our work in Sec. 4.2. We describe the sequential versions of the various algorithms we explore for balanced distance-1 coloring in Sec. 4.3 and discuss how they are parallelized in Sec. 4.4. We discuss how the algorithms developed for distance-1

coloring are modified to handle the partial distance-2 coloring case in Sec. 4.5. We review essential features of the platforms for which the implementations are targeted in Sec. 4.6. We present and discuss experimental results in Sec. 4.7.

4.2 Problem Statement and Background

Given a general graph $G = (V, E)$, a distance-1 coloring of G is an assignment of colors to vertices such that any two adjacent vertices (which are at distance 1 from each other) are assigned different colors. This is the "usual" coloring problem, and henceforth, for brevity, we will drop the qualifier "distance-1" unless we need to distinguish it from distance-2 coloring.

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 (Meyer, 1973). Its history, however, goes even further back to a conjecture by Erdös, a conjecture settled in 1970 by Hajnal and Szemerédi (Hajnal and Szemerédi, 1970) forming their celebrated theorem: a graph with maximal degree Δ is equitably k-colorable if $k \geq \Delta + 1$. This bound is sharp. One of the directions of early theoretical research in this field had been to obtain better upper bounds than $\Delta + 1$ for special graph classes (Erdös, Rényi, and Sós, 1970).

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 (Bodleander and Fomin, 2005) 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.

In this dissertation, we take a less formal route and think of balanced coloring without fixing a parameter l. More specifically, given a graph $G(V, E)$, the problem is to compute a distance-1 coloring such that each color class receives approximately $\frac{|V|}{C}$ vertices, where C is the number of colors used.

4.2.1 Related Work

The equitable coloring problem asks for an equitable k-coloring with the smallest possible k. This problem is NP-hard, as the classical coloring problem can be trivially reduced to it. Polynomial time equitable coloring algorithms are known for various special classes of graphs, including trees (Chen and Lih, 1994), r-partite graphs (Wang and Zhang, 2000), line graphs (Wang and Zhang, 2000), and planar graphs (Yap and Zhang, 1998). Furmanczyk (Furmańczyk, 2004) provides a survey of work on equitable colorings until the early 2000's.

Equitable coloring and balanced coloring (in the sense just mentioned) find important applications in various areas. Examples include load-balanced partitioning for domain decomposition methods (Smith, Bjørstad, and Gropp, 1996), parallel sparse matrix computations on irregular grids (Melhem and Ramarao, 1988), and various types of scheduling and timetabling problems (Blazewick et al., 2001). Tucker in a 1973 paper (Tucker, 1973) 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 (Robert K. Gjertsen, Jones, and Plassmann, 1996),

where they developed a balanced, distributed memory parallel coloring heuristic building on their own earlier work on parallel graph coloring that was unconcerned with balancing color classes (Jones and Plassmann, 1993). Their balancing heuristic draws ideas from approximation algorithms for the *bin packing* problem (E.G. Coffman, Garey, and Johnson, 1997) and a coloring work in (Pommerell, Annaratone, and Fichtner, 1992).

The work in (Robert K. Gjertsen, Jones, and Plassmann, 1996) has connections to that in this paper, but it differs both in terms *context* and *approach*. In particular, the context in (Robert K. Gjertsen, Jones, and Plassmann, 1996) is a distributed memory setting in which the vertex set of a graph is already partitioned among processors. Further, the authors assume that the partitioning is a good one in the sense that each processor is assigned nearly the same number of vertices. Based on an initial coloring of the partitioned graph, the authors then run a balancing heuristic that respects the vertex partitioning (avoids relocation of vertices to processors). In contrast, in this work, we do not assume any *a priori* partitioning of the vertex set. In fact, the assignment of vertices to processors (or threads) is expected to be done after the balanced coloring is achieved, which is an advantage. In terms of approach, the work in (Robert K. Gjertsen, Jones, and Plassmann, 1996) focuses on one class of algorithms: given an initial coloring, how can balancing be achieved without increasing the number of colors used? In contrast, here we consider a wider variety of algorithms, provide implementations on modern day multi-core and manycore platforms, and experimentally evaluate their performance as well as the trade-offs they offer.

4.2.2 A Foundational Scheme

For the standard distance-1 graph coloring problem, despite its NP-hardness, the greedy scheme outlined in Algorithm 2 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 2 Greedy

Greedy $(G = (V, E))$

for each $v \in V$ in some order **do**

  for each w adjacent to v **do**

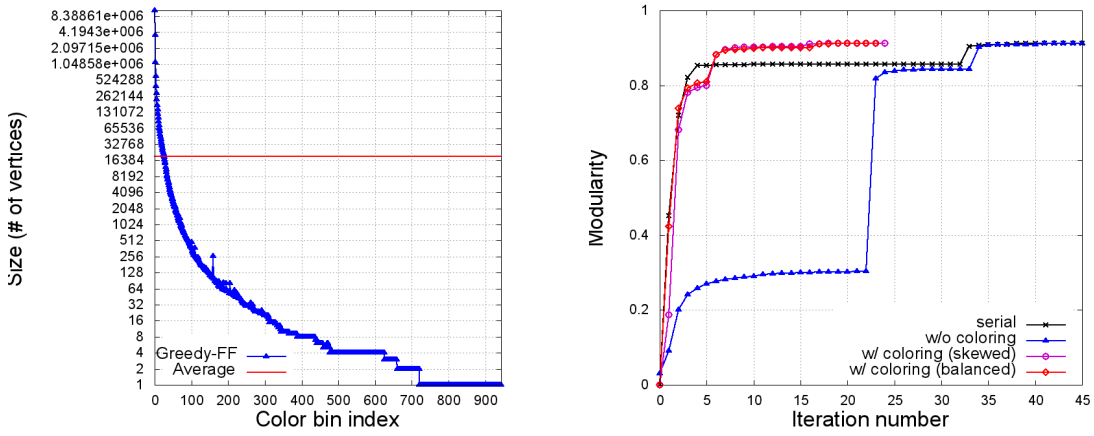
    Mark the color of w as forbidden to v

  Assign v a color not marked as forbidden to v

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 with regards to (a) is to rank the vertices in a non-ascending order of "degree", where degree is suitably defined (e.g. as the number of neighbors, or the number of already colored neighbors, or the number of differently colored neighbors). The intuition is to treat vertices that are likely to be harder to color, earlier in the process.

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 analogy to the bin packing problem 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



(a) Greedy Coloring, Input: uk-2002

(b) Community Detection, Input: cnr

Figure 4.1 a) The size distribution of the color classes obtained by the Greedy First Fit heuristic for distance-1 coloring on 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 Louvain method (Lu, Halappanavar, and Kalyanaraman, 2015). 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.

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, could be 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. 4.1a confirms this expectation on a real world graph. 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.

4.2.3 Community Detection: A Motivating Application

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 consider balanced coloring in the context of parallel community detection, based on an implementation called "*Grappolo*" that we developed for multi-core and manycore architectures (Lu, Halappanavar, and Kalyanaraman, 2015; Chavarría-Miranda, Halappanavar, and Kalyanaraman, 2014). The parallel implementation is based on the sequential Louvain heuristic (Blondel et al., 2008). The Louvain method, which is one of the most widely used community detection algorithms, uses the modularity function (Newman, 2004b) 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 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 (Lu, Halappanavar, and Kalyanaraman, 2015). 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 (as shown in Fig. 4.1a) can reduce overall scalability, particularly while processing the smaller color classes. The purpose of balancing the color classes is thus to improve thread utilization for those smaller color classes, while ensuring that the overall output quality of the solution (modularity) is maintained.

The chart in Fig. 4.1b demonstrates this purpose. The chart shows that balanced coloring matches skewed coloring in its impact on community detection both in terms of convergence rate (i.e., number of iterations taken to complete) and in terms of output quality (final modularity), while offering the added advantage of improved thread utilization within every iteration, since the color classes are balanced.

4.3 Algorithms for Balanced Distance-1 Coloring

In this section, we present multiple heuristics to compute a balanced distance-1 coloring of an input graph, as summarized in Table 4.1.

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. We refer to these approaches as "guided" (to signify that they are guided by an initial coloring).

4.3.1 *Ab initio* balancing strategies

Within the *ab initio* category, we consider two well-known variants of the Greedy scheme outlined in Algorithm 2 that differ in how the choice of color to be assigned

to a vertex in each step is done. Both variants seek to achieve balanced coloring by virtue of the color choice strategy:

- *Greedy-LU*: A vertex is assigned the *least used* color among all currently available permissible colors. If no permissible color exists, then a new color is created and assigned to the vertex.
- *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 $B = \Delta + 1$, where Δ is the maximum clique size. 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 (Manne and Boman, 2005).

4.3.2 Guided balancing 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 refer to a color class of size greater than

γ as an *over-full bin*, and a color class of size less than γ as an *under-full bin*. (We use the terms *bin* and *color class* interchangeably throughout the chapter.)

Broadly, we classify our guided strategies into two 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, instead of enforcing that the number of color classes remains unchanged, *all* vertices are colored afresh, this time with a balance constraint imposed. We call this strategy *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 4.4.

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 (Culberson and Luo, 1996) 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. 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 is less than γ.

4.4 Parallel Algorithms

We parallelized all of the guided balanced coloring algorithms presented in Section 4.3 for the shared memory model. For each heuristic we developed two OpenMP-based implementations—one for conventional multicores and another for the Tilera many-core platform. To obtain the initial coloring we used a parallel implementation available for Greedy-FF from a previous effort (Catalyurek et al., 2012). 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 that 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 leave bins unbalanced.

4.4.1 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., size reaches γ) at any 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 (Catalyurek et al., 2012). 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 2. Such a parallelization will not preclude the possibility of a pair of adjacent vertices from 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 Algo-

rithm 3. 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 3 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 ≠ 0 do
    for v ∈ U do in parallel
        Let k be the smallest index of an under-full bin that is permissible   ▷ FF
        if k exists then
            Let j ← color[v]
            color[v] ← k
            Update size of bins k and j                                        ▷ synch. step
    R ← ∅
    for v ∈ U do in parallel                                                   ▷ check for conflicts
        for w ∈ adj(v) do
            if (color[w] = color[v] and v > w) then
                R ← R ∪ {v}
                Update size of bin color[v]                                    ▷ synch. step
    U ← 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 prac-

tice, we observed that the closely related quantity – the actual number of iterations needed to clear all conflicts – is typically bounded by 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 bin 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. Depending on the strategy used to pick an under-full bin (FF or LU), we refer to this Color-centric parallelization scheme as either CFF or CLU. A template for the color-centric scheme is shown in Algorithm 4.

Algorithm 4 Color-centric parallel balanced coloring

ColorParallelGuidedBalancing($G = (V, E)$)

Obtain an initial coloring of G

Let Q be the set of over-full 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, j ▷ synch. 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 computing an initial coloring in VFF and CFF, while for VLU and CLU the use of any initial coloring scheme is reasonable.

4.4.2 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 to assign more vertices to smaller-indexed color classes. In other words, color classes are expected 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. 4.1.

Our parallel algorithm with scheduled moves is outlined in Algorithm 5. Intu-

itively, 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.

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 5) 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 decreasing order increases the likelihood of 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.

4.4.3 Parallel Recoloring

The parallelization we use for the balancing based on Recoloring is outlined in Algorithm 6. This is similar to the vertex-centric parallel scheme given in Algorithm 3 with the main difference being that we recolor *all* the vertices from scratch and that balance is *imposed* as the recoloring proceeds. In particular, the recoloring approach colors vertices roughly in the order of vertices as they appear from the largest color to the smallest color of the initial coloring. The rationale for this ordering, as mentioned in Section 4.3.2, is that the re-coloring procedure would have an opportunity

[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).

Algorithm 5 Parallel shuffling using scheduled moves

ScheduledBalancing($G = (V, E)$)

Obtain an initial coloring of G using Greedy-FF

Let C be the number of colors used, and let $\gamma = |V|/C$

Let Q_O be an ordered set of over-full bins in increasing order of color index

Let Q_U be an ordered set of under-full bins 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**

$color[v] \leftarrow k$

to use fewer colors, since now the vertices that were "difficult" to color initially are processed earlier. In the process of recoloring, we also strive for the size of each color class to be as close as possible to the average size of a color class γ obtained from the initial coloring.

Algorithm 6 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 an 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 $bin[k] \leq \gamma$

    Increment $bin[k]$ by 1 ▷ synch. 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**

        $R \leftarrow R \cup \{v\}$

  $U \leftarrow R$

4.4.4 Complexity

With careful choice of data structures, the sequential Greedy scheme (Algorithm 2) 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 3 through 6, the total "additional" work incurred due to parallelization is no more than the work involved in Algorithm 2. Furthermore, the number of rounds required by the iterative variants (Algorithm 3 and Algorithm 6), as argued earlier, is typically a small constant in practice. Therefore, the net total work in any of our schemes can be upper-bounded by $O(|V| \cdot \Delta)$.

The above complexity represents a worst-case, where for instance, multiple vertices with relatively high degrees tend to occupy the overfull bins in the initial coloring. But such cases also require a relatively high number of such vertices in the input, which is less likely to be observed in real world networks with power-law like degree distributions.

4.5 Partial Distance-2 Coloring

4.5.1 Preliminaries

As mentioned in Section 4.1, besides distance-1 coloring, we considered in the current work a balanced version of another coloring problem, *partial distance-2 coloring*, that is defined on a bipartite graph $G_b = (V_1, V_2, E)$. The coloring "rule" here is to assign colors to vertices in one of the vertex sets, say V_2, such that any pair of vertices v_i, v_j from V_2 that share a common neighbor v_c in V_1 (that is, $(v_c, v_i) \in E$ and $(v_c, v_j) \in E$)

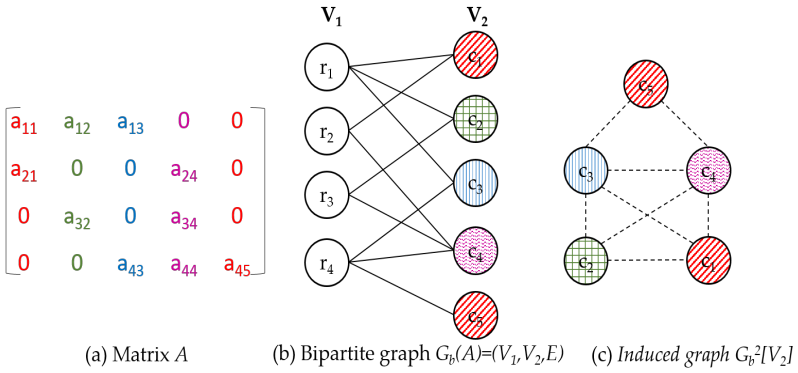


Figure 4.2 Illustration of equivalence among structurally orthogonal partition of a matrix A (a), partial distance-2 coloring of the vertices in V_2 in bipartite graph $G_b = (V_1, V_2, E)$ of A (b), and distance-1 coloring of the subgraph of the square graph induced by V_2, that is ${G_b}^2[V_2]$ (c).

get different colors[2]. The objective of the standard version of the problem, as in distance-1 coloring, is to use as few colors as possible. The balanced variant seeks to, in addition, balance the color classes.

The partial distance-2 coloring problem is an important model in computations involving nonsymmetric matrices. For example, a partitioning of the columns of a nonsymmetric matrix A into groups of *structurally orthogonal* columns—a group in which no two columns share nonzero entries in the same row position—can be modeled by a partial distance-2 coloring of the column vertices V_2 of the bipartite graph $G_b(A) = (V_1, V_2, E)$ representing the sparsity structure of the matrix A (Gebremedhin, Manne, and Pothen, 2005). Such a model is useful, for example, in an efficient computation of a Jacobian matrix A using automatic differentiation. Figure 4.2 shows a small example that illustrates how a structurally orthogonal column partition of a matrix is modeled as as a partial distance-2 coloring of the bipartite graph.

[2]This variant is called "partial" since only one of the two vertex sets in the bipartite graph is colored. It is called "distance-2" since, in the vertex set to be colored, any two vertices that are two edges away from each other are required to get different colors. The naming was first used in (Gebremedhin, Manne, and Pothen, 2005).

Figure 4.2c illustrates yet another equivalence. In general, a partial distance-2 coloring on the vertex set V_2 of a bipartite graph $G_b = (V_1, V_2, E)$ is equivalent to a distance-1 coloring of a certain derived graph—the subgraph of the *square graph* of G_b induced by the vertex set V_2, which we denote by ${G_b}^2[V_2]$. For a general graph $G = (V, E)$, the square graph $G^2 = (V, F)$ is a graph defined on the same vertex set V and where the edge set F consists of pairs of vertices that are distance less than or equal to two edges from each other. In other words, $F = E \cup E'$, where E' corresponds to pairs of vertices that are at distance exactly two edges from each other in G. If the graph under consideration is a bipartite graph $G_b = (V_1, V_2, E)$, then the square graph ${G_b}^2 = (V_1, V_2, F)$, which no longer is bipartite, is such that the edge set can be viewed as having three parts: $F = E \cup E_1 \cup E_2$, where E_1 corresponds to the "new" edges that run among vertices in V_1 (those that are at distance 2 in G_b) and E_2 corresponds to the edges that run among vertices in V_2 (those that are at distance 2 in G_b). The subgraph of ${G_b}^2$ induced by the vertex set V_2 could then be written as $G_c = (V_2, E_2)$. In network science literature, the graph G_c is also sometimes referred to as a "projection" on to the vertex set V_2, and in the numerical optimization community G_c is also known as the "column intersection graph" of the associated non symmetric matrix.

The discussion in the paragraph above focused on the case where the vertex set in the bipartite graph to be colored is V_2. Entirely analogous discussion and definitions apply if the vertex set to be colored were V_1 instead.

Now if partial distance-2 coloring on the vertex set V_2 of the bipartite graph $G_b = (V_1, V_2, E)$ is equivalent to distance-1 coloring of the intersection graph $G_c \equiv {G_b}^2[V_2]$, why don't we then simply construct G_c and solve the distance-1 coloring problem on it using algorithms developed for distance-1 coloring, instead of developing algorithms tailored for partial distance-2 coloring? As has been argued in (Gebremedhin, Manne,

and Pothen, 2005), there are several reasons for this. First, the partial distance-2 coloring formulation offers greater flexibility. In particular, the intersection graph necessarily looses structural information contained in the original bipartite graph. For example, considering the context of an underlying nonsymmetric matrix, given two column vertices in V_2 joined by an edge in G_c, one cannot determine the row at which the two columns share nonzero entries. Second, for some structures, the intersection graph ${G_b}^2[V_2]$ could turn out to be substantially denser (have more edges) than the original graph G_b and hence require more memory. Note that by construction, each vertex $u \in V_1$ of $G(V_1, V_2, E)$ would correspond to a k-clique in $G_b^2[V_2]$, where k is the degree of u. Third, the partial distance-2 coloring formulation avoids the need for building a different data structure than the one used to represent the input graph.

4.5.2 Algorithms

Algorithm 7 Greedy Partial Distance-2 Coloring

GreedyPD2C($G_b = (V_1, V_2, E)$)

```
for each v ∈ V2 in some order do
    for each w adjacent to v do
        for each x adjacent w do
            Mark the color of x as forbidden to v
    Assign v a color not marked as forbidden
```

The greedy scheme for distance-1 coloring outlined in Algorithm 2 can be easily modified to obtain a solution for the partial distance-2 coloring problem. The needed modifications are: (i) the input is a bipartite graph $G_b = (V_1, V_2, E)$, (ii) the for-loop iterates over $v \in V_2$, and (iii) the *neighbors* of a vertex v are those vertices x that are connected to v by a path v, w, x of length two edges. The algorithm modified in this

manner is outlined in Algorithm 7. Let $\delta(V_2)$ denote the average degree of a vertex in V_2, and $\Delta(V_1)$ denote the maximum degree of a vertex in V_1. Consequently, the runtime complexity of the algorithm is as follows (Gebremedhin, Manne, and Pothen, 2005): $\mathcal{O}(|V_2|\delta(V_2)\Delta(V_1))$, which is same as $\mathcal{O}(|E|\Delta(V_1))$.

We note here that it is possible to develop a faster greedy algorithm by exploiting the bipartite structure of the graph and by relaxing the constraint on the maximum number of colors used. One such approach is to linearly scan the vertices in V_1 and assign different colors to all its neighbors in V_2, while using colors in a greedy first-fit fashion. Such an approach, while improving the runtime ($\mathcal{O}(|E|)$), could potentially use more colors relative to Algorithm 7. Furthermore, parallelization of such an approach may necessitate multiple iterations to resolve potential conflicts introduced during the parallel coloring procedure. For these reasons, we use Algorithm 7 as the basis for the balanced variants developed and results presented in this paper.

The parallelization strategies discussed in Section 4.4 focused on distance-1 coloring. The corresponding algorithms for partial distance-2 coloring follow the same methodology, with the modifications outlined earlier in this paragraph. We therefore omit detailed presentation of algorithms for partial distance-2 coloring.

4.5.3 Another example of an application

Parallel implementation of the *coordinate descent algorithm* is another example of an application in which partial distance-2 coloring is useful. Let us review the coordinate descent algorithm briefly to show how partial distance-2 coloring is relevant there. In the coordinate descent algorithm, the rows of matrix A correspond to *samples* and the columns correspond to *features*. The corresponding dimensions are often denoted by n (samples) and k (features). Formulated in a generic fashion, the coordinate descent (CD) algorithm consists of four steps that are performed iteratively until convergence

is reached:

Step (1) *Select* a set of coordinates J

Step (2) *Propose* increment δ_j, $j \in J$

Step (3) *Accept* some subset $J' \subseteq J$ of the proposals

Step (4) *Update* weight w_j for all $j \in J'$.

The approach taken in the Select step determines the type of the CD Algorithm. For example in *cyclic* or *stochastic* CD the selection targets a singleton, whereas in *greedy* CD one selects a set—and in the extreme case (fully greedy), the entire set of features is selected. Clearly, greedy is better suited for parallel CD. In a parallel formulation of CD, the Propose and Update steps need to be performed concurrently. In order to perform the Update step with maximal exploitation of parallelism, one needs to identify groups of structurally orthogonal columns (features), since then matrix entries in all the columns in a group can be updated safely concurrently. This identification in turn is what is modeled using partial distance-2 coloring (on the column-vertices) of the bipartite graph representing the sparsity structure of the matrix A.

4.6 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. (These NOCs include one for coherence traffic, a user-programmable message passing NOC, and a dedicated I/O NOC.)

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 (Chavarría-Miranda, Halappanavar, and Kalyanaraman, 2014) 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.

4.7 Experimental Results

4.7.1 Experimental Setup

Test Platforms: We used two platforms for testing: the manycore Tilera TileGX36 presented in Section 4.6, and an x86 AMD Interlagos 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. 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. The community detection code has been parallelized using OpenMP and Tilera-specific extensions for memory management, synchronization and atomic operations.

The AMD multi-core platform consists of a dual-socket Interlagos processor with 64GB of memory. Each socket has 16 cores running at 2.1 GHz. Each pair of cores is grouped into a module sharing a single floating-point functional units and separate integer functional units. Each core contains 16 KB of L1 cache, while each module shares 2 MB of L2 cache. Four modules share 8 MB of L3 cache. Each socket contains eight modules on our system.

Test inputs: The test inputs used in the different experiments on distance-1 coloring are summarized in Table 4.2. These inputs were all downloaded from the University of Florida sparse matrix collection (Davis and Hu, 2011), with the following exceptions: $MG2$ is a custom-built biological network obtained from protein sequences of a metagenomics data set (Daily et al., 2015); $rgg11-22$ represents a random geometric graph (Penrose, 2003) that was generated using the generator described in (Halappanavar, 2009); and $random2$ was generated using a simple random function that applies a probability of edge between any pair of vertices. 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 4.3).

All results pertaining to distance-1 coloring are presented in Sections 4.7.2 through 4.7.4

The test inputs and results on partial distance-2 coloring are presented in Section 4.7.5. The extensions to balanced coloring are presented in Section 4.8.

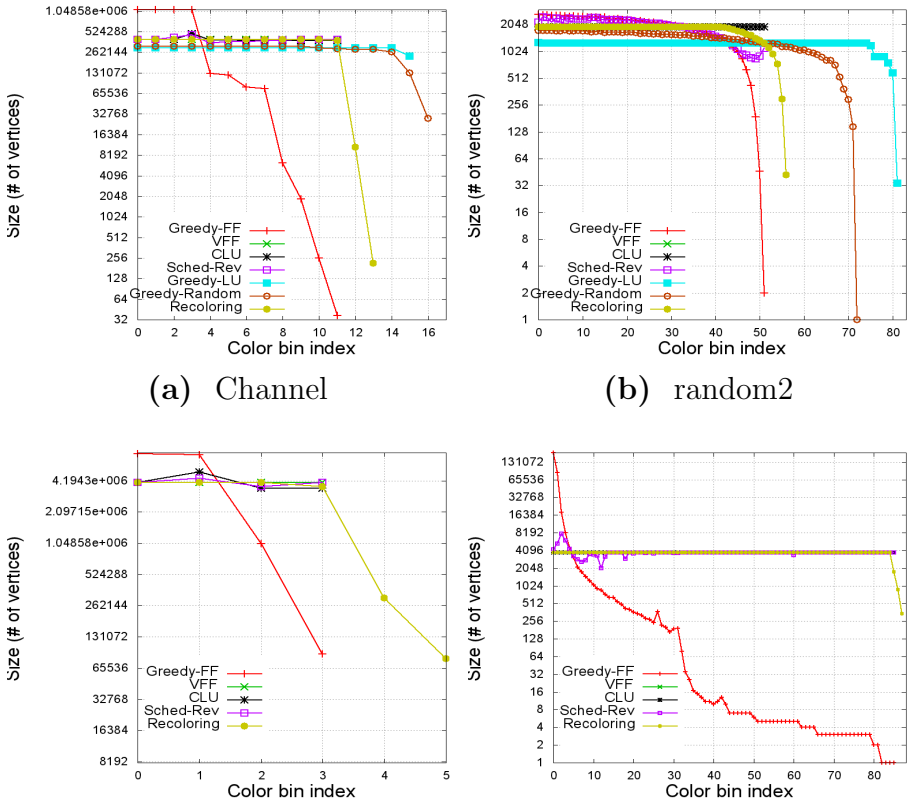
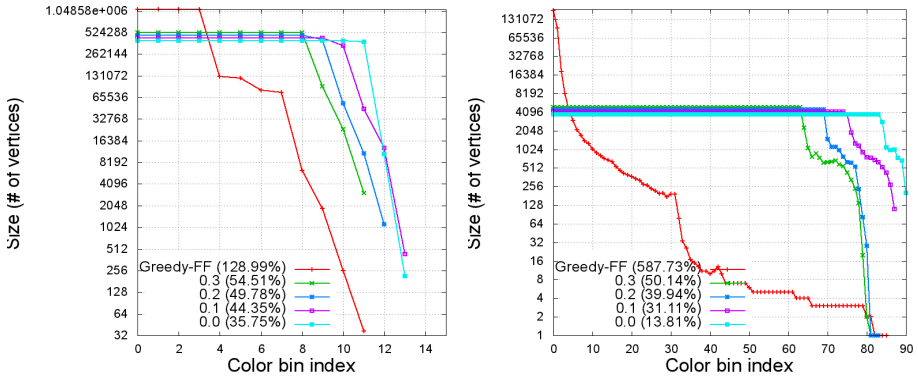


Figure 4.3 Distance-1 coloring: Distribution of color class sizes produced by the different balanced coloring schemes (horizontal axis corresponds to colors (bins) and vertical axis to sizes of color classes). Recall that smaller color class sizes correspond to reduced parallelism in the end-application, while higher number of colors corresponds to increased number of parallel steps within the application. For Channel and random2, color class sizes from all balancing schemes are shown. For NLPKKT200 and CNR, color class sizes only from the balancing schemes that produce same or comparable number of colors to the Greedy-FF scheme are shown.

4.7.2 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 4.1 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 mean color size. The closer this value is to zero 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).



(a) Channel **(b)** CNR

Figure 4.4 Recoloring: The figure illustrates the impact of different bounds on bin sizes for the recoloring scheme (Algorithm 6). The default recoloring scheme which uses $\tau = 0.0$ (identifed in the chart by the "0.0" label) fixes the average size for each bin based on the numbers from the initial (unbalanced) coloring scheme. The average size is then varied from 10%, 20% and 30% (identified by labels "0.1", "0.2" and "0.3" respectively), which results in a smaller number of colors used and possibly a higher imbalance in bin sizes than the default recoloring scheme. The percentages inside paranthesis against each τ setting indicates the imbalance, measured by the Relative Standard Deviation of the resulting color class sizes.

Table 4.3 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. In fact, we evaluate this tradeoff in the context of partial distance-2 coloring (see Section 4.7.5).

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 4.4.3, 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 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.

Fig 4.3 illustrates the effect of the different balancing schemes—it shows the sizes of all the color classes produced by the different balancing schemes.

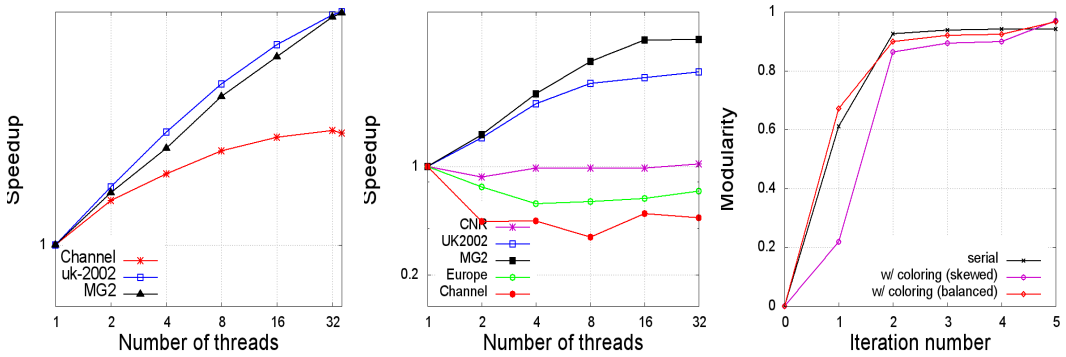
Looking further into the Recoloring Scheme: For the Recoloring scheme, by allowing some of the color classes to slightly exceed the average color class size (γ), it is possible to achieve a reduction in the number of colors used compared to the baseline version of Recoloring (Algorithm 6). However, the balancing quality may

not necessarily be maintained. We empirically investigated this tradeoff between the number of colors and balancing quality. In particular, we modified Algorithm 6 such that the line $bin[k] \leq \gamma$ is replaced by $bin[k] \leq \gamma \cdot (1 + \tau)$, where τ is a small fraction indicating how much offset from γ is "tolerated". Fig. 4.4 shows results for four values of τ (0, 0.1, 0.2 and 0.3) on two test inputs. The results suggest that the nonzero values for τ help reduce the number of colors used. However, as expected, the balancing quality degrades with increase in τ.

4.7.3 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 4.4 and 4.5 show the run-times taken by the VFF balancing scheme. (We select VFF because it was one of the schemes that produced the best balancing results (as was discussed in Section 4.7.2).) The corresponding speedup charts are shown in Fig. 4.5.

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 (data not shown). The corresponding performance gain on the Tilera platform was a more modest 2× (Table 4.6).



(a) Speedup on Tilera

(b) Speedup on x86

(c) Community detection

Figure 4.5 (a, b) Speedup obtained by our Tilera manycore and x86 multicore implementations of the VFF balancing scheme. Speedups are relative to one thread executions on both systems. (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 (Blondel et al., 2008).

The speedup trends observed on both architectures also show the impact of the number of initial colors on parallel performance. As shown in Figures 4.5a and b, the speedups obtained on MG2 (2K colors) and uk-2002 (943 colors) are superior than on other inputs. Intuitively, fewer colors imply a higher probability for concurrent bin size updates.

From the parallel runtimes shown in Tables 4.4 and 4.5, it can be observed that, on a *per-core* basis, the Tilera platform is generally 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 wider superscalar instruction scheduling. However, at full system scale, the runtimes on the Tilera platform begin to become comparable to the runtimes on the x86 platform.

This observation, coupled with the observation that the Tilera platform exhibited a better scalability than x86, (compare Figures 4.5a vs. 4.5b), indicates that the Tilera platform, on larger system sizes, has the potential to make up for the difference with respect to x86 and even surpass its absolute performance.

In Table 4.6, 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 $\sim 2\times$ faster than VFF[3]. Considering the fact that Sched-Rev also performed appreciably well in terms of balance quality (Section 4.7.2), we conclude that it provides a reasonable trade-off between quality and performance among the different balancing schemes presented in this paper.

Table 4.7 compares the runtime performance between a guided scheme (VFF) and an *ab initio* scheme (Greedy-LU). As was shown earlier in Table 4.3, the VFF and Greedy-LU schemes represent one of the top balancing schemes in their respective categories (guided and *ab initio*). Since the guided schemes require an initial coloring, we include that runtime as well in the Table 4.7. The results confirm the runtime advantage expected for *ab initio* schemes as they compute a balanced coloring directly, without requiring an initial coloring. However, in most cases, the total runtime for guided schemes (initial coloring + VFF balancing) is comparable to the *ab initio* runtimes, while in general delivering a better balancing quality (Table 4.3).

4.7.4 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 4.2.3. Since VFF was one of the leading schemes for balance quality, we

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

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 4.8 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 4.8 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 (with or without 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 4.8.

The results in Table 4.8 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 modularlity in the first phase, as illustrated in Fig. 4.5(c) —which is a consequence of the revised

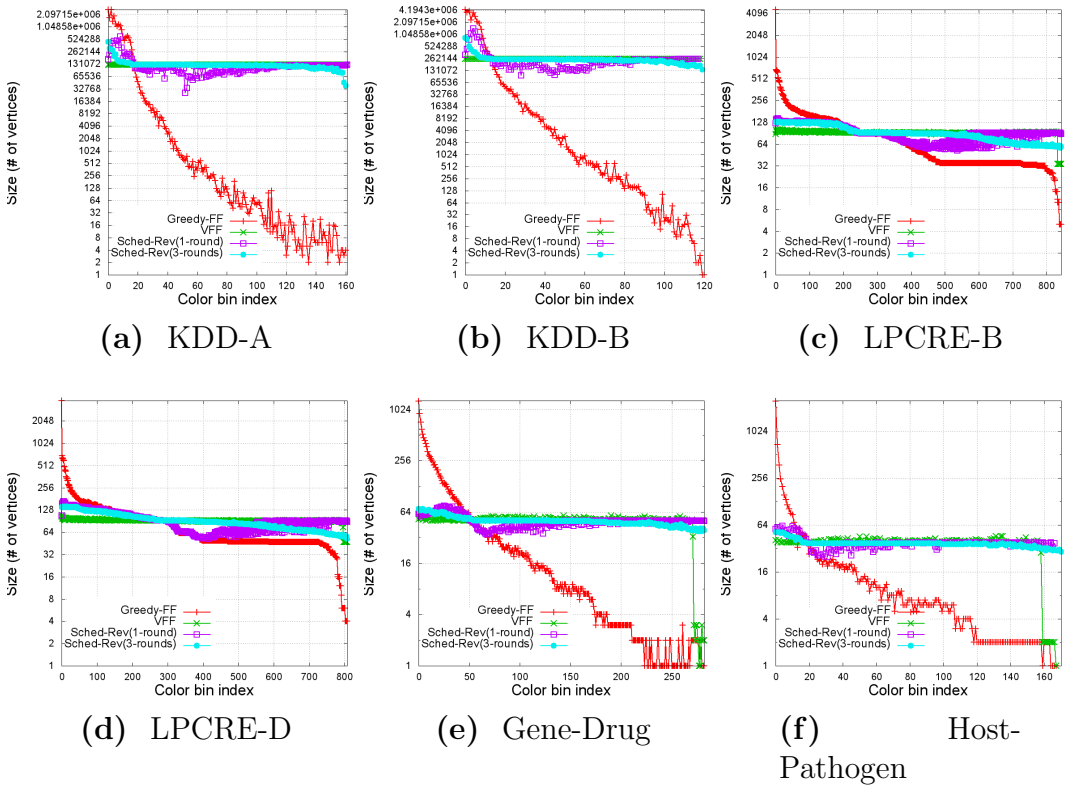
ordering of vertices due to balanced coloring.

4.7.5 Results on Partial Distance-2 Coloring

Our work on partial distance-2 coloring is partly motivated by its application in the parallelization of the stochastic coordinate descent (SCD) algorithms Scherrer et al., 2012. We therefore selected for our experiments two test instances (KDD-A and KDD-B) related to SCD and representing data from the KDD Cup 2010 Challenge on educational data mining (Yu et al., 2011). Further, we selected two additional instances (LPCRE-B and LPCRE-D) from the Florida Sparse Matrix Collection (Davis and Hu, 2011) that arise from linear programming problems. We consider the bipartite graphs representation of the sparsity structures of these matrices (instances), $G = (V_1, V_2, E)$, where the set V_1 corresponds to the rows and the set V_2 corresponds to the columns, and the edge set E corresponds to the nonzero entries. In partial distance-2 coloring, we color only the vertex set V_2 (please refer Section 4.5 for definition and other details of partial distance-2 coloring).

As additional test cases from another application domain, we experimented on two bipartite graph inputs obtained from the biology domain—viz. gene-drug interactions (Griffith et al., 2013) and host-pathogen network (Wardeh et al., 2015). Balanced coloring can be used as a way to determine efficient partial orderings of vertices for parallel processing of such networks in co-clustering applications (Hanisch et al., 2002). The testsets used in our experiments are summarized in Table 4.9.

Table 4.10 shows the parallel runtimes for two balancing schemes—VFF and Sched-Rev—on the AMD platform. We note that these are bipartite graph implementations of those schemes. For Sched-Rev we provide results from running the algorithm for one round and for three rounds (simply iterated three times). We observed relatively better performance from three rounds than two rounds. The distributions



(a) KDD-A **(b)** KDD-B **(c)** LPCRE-B

(d) LPCRE-D **(e)** Gene-Drug **(f)** Host-Pathogen

Figure 4.6 Partial distance-2 coloring: The distributions of color class sizes produced by the different balanced coloring schemes (horizontal axis corresponds to colors (bins) and vertical axis (in $\log_2$ scale) to sizes of color classes).

of color class sizes while using the different schemes are provided in Figure 4.6. We observe that the quality of output from Sched-Rev with three rounds is comparable to VFF, but the execution time is relatively high. Sched-Rev (with one round) provides faster execution times with comparable quality. The corresponding distributions of color class sizes from the unbalanced coloring scheme (Greedy-FF) are also provided in Figure 4.6 (in red). Overall, the results demonstrate the general effectiveness of our balancing schemes for partial distance-2 balanced coloring as well.

4.8 Extensions to Balanced Coloring

In this section, we present two extensions to balanced coloring, motivated by two different practical considerations.

4.8.1 Weighted Balancing

Color class sizes (the number of vertices having the same color) form the basis for the balancing schemes presented thus far. This assumes that the work associated with each vertex is uniform in the end application. However, in some applications, the workload may vary from vertex to vertex. For such scenarios, a weighted treatment of vertices may be more appropriate. Following a similar approach as in Robert K. Gjertsen, Jones, and Plassmann, 1996, we implemented and evaluated a weighted extension for balancing, by setting the weight of a vertex to its degree. We implemented by modifying the VFF balancing scheme as follows. Let $\omega(u)$ denote the weight of vertex u. Then, the target size for each color C is given by:

$$\gamma_\omega = \frac{\sum_{u \in C} \omega(u)}{|C|}$$

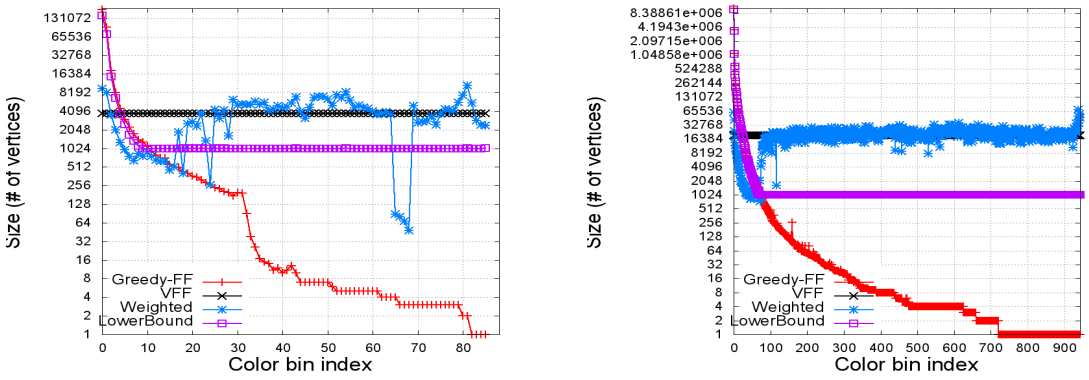
4.8.2 Lower Bound-based Coloring

One purpose of balancing is to ensure a uniform distribution of parallel workload across color classes in the end application. An alternative, more pragmatic strategy would be to *not* aim for a balancing across all color classes, but rather focus on ensuring that the smaller color classes are filled to a threshold size, sufficient to utilize the threads efficiently. Such a scheme could have an advantage in reducing the balancing cost—because fewer vertex migrations are needed. This motivated our second extension. We modified the VFF scheme to take as input a *lower bound*

(threshold) size (denoted by γ_ℓ) for each color class. During the process of balancing, we consider only bins (colors) that have less than the threshold as potential candidates for receiving vertices (from larger bins). Therefore, the resulting color size distribution is expected to be less balanced compared to the other balancing schemes, whereas the overhead for balancing is reduced. We denote this modified scheme as "LB-based" coloring scheme.

4.8.3 Experimental Results

We compared the two extensions outlined above (weighted and LB-based) against the "baseline" schemes of VFF balancing and the Greedy-FF (initial coloring). Fig. 4.7 and Table 4.11 show the results of our comparative evaluation.



(a) CNR **(b)** uk2002

Figure 4.7 Balanced coloring extensions: Distribution of color class sizes produced by the two distance-1 balanced coloring extensions (Weighted, LowerBound) compared to VFF balancing (without weights or lower-bounds) and and initial coloring (Greedy-FF). Horizontal axis corresponds to colors (bins) and vertical axis to sizes of color classes (measured in the *number* of vertices).

Fig. 4.7 shows the balancing quality produced by the extension schemes and the baseline schemes. As can be expected, the weighted scheme does *not* exhibit a good balance based on the number of vertices per color bin. However, if one were to take

into account the sum of the weights of the vertices per bin, then the weighted scheme does indeed achieve a balance. The LB-based scheme also shows an expected behavior, where the color classes that contain less than γ_ℓ vertices in the initial coloring are the only classes to receive new vertices into their bins (until the lower bound is met). This implies that the balancing quality achieved by the LB-scheme is sub-optimal but the benefits are expected to lie in its reduced balancing cost (which we evaluate next).

Table 4.11 shows the run-times for the balancing step and the community detection step (application impact) corresponding to the extension schemes and the baseline schemes. In the case of weighted scheme, the balancing cost is comparable to that of VFF balancing. As for the community detection step, the benefits due to using vertex weights during balancing are *not* readily realized. This is because our parallel implementation of the community detection step (Grappolo) currently uses a vertex-based parallelization instead of an edge-based parallelization. The latter is more conducive to the weighted scheme. As part of our future study, we plan to implement and evaluate such an edge-based scheme in Grappolo, to better exploit the benefits of the weighted scheme.

In the case of LB-based scheme, we observe that the balancing cost is significantly smaller than the VFF balancing cost. This confirms our expectation of reduced work during the balancing procedure through the use of a lower bound size. Note that this may cause sub-optimal balancing (as shown in Figure 4.7), which in turn may affect the end application parallel performance. Table 4.11 shows a marginal increase in the run-times for community detection as per this expectation. In fact, as the lower bound value (γ_ℓ) is further reduced, the size distribution of the color classes would tend closer to the skewed distribution of the initial coloring.

Table 4.1 A comprehensive list of balancing strategies for distance-1 coloring presented and studied in this paper. The input graph is denoted by $G = (V, E)$. The same set of strategies are also extended to obtain a balanced partial distance-2 coloring on bipartite graph inputs.

Strategy	**Category**	**Description**
Greedy-LU	*ab initio*	Run Algorithm 2 with LU color choice among permissible colors.
Greedy-Random	*ab initio*	Run Algorithm 2 with Random color choice among permissible colors.
Shuffling-unscheduled	guided	Run Algorithm 2 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. Specializations include Vertex-centric (*VFF*) and Color-centric(*CLU*).
Shuffling-scheduled	guided	Run Algorithm 2 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 2 with FF color choice strategy. Let the number of colors used be C. Let $\gamma = \lvert V\rvert/C$. 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 2 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.

Table 4.2 Input statistics for the graphs used in our distance-1 coloring study.

Input graph	Num. vertices (n)	Num. edges (m)	Degree stats max.	Degree stats avg.
random2	100,000	9,994,356	263	199.89
CNR	325,557	2,738,970	18,236	16.28
coPapersDBLP	540,486	15,245,729	3,299	56.41
rgg11-22	4,194,304	27,306,228	23	13.02
Channel	4,802,000	42,681,372	18	17.77
MG2	11,005,829	674,142,381	5,466	122.50
NLPKKT200	16,240,000	215,992,816	27	26.60
uk-2002	18,520,486	261,787,258	194,955	28.27
Europe-osm	50,912,018	54,054,660	13	2.12

Table 4.3 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 Greedy-FF, Recoloring and the two *ab initio* schemes is provided in parenthesis (next to their respective RSD values).

Input	**Init. coloring**		**Guided schemes**					***Ab initio* schemes**		
graph	Greedy-FF		VFF	CLU	Sched-Rev	Recoloring		Greedy-LU		Greedy-Random	
random2	37.17%	(52)	**0.08%**	0.09%	26.21%	22.74%	(57)	9.10%	(82)	21.04%	(73)
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)
rgg11-22	129.03%	(15)	**0.00%**	**0.00%**	14.69%	26.66%	(15)	4.29%	(17)	38.93%	(16)
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)
NLPKKT200	99.88%	(4)	**0.00%**	17.65%	7.95%	72.13%	(6)	0.00%	(23)	1.64%	(23)
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)

Table 4.4 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 4.5 Parallel run-times (in seconds) of the VFF scheme on different number of cores of the AMD x86 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
CNR	0.15	0.17	0.15	0.15	0.15	**0.14**
Channel	**0.79**	1.79	1.77	2.27	1.59	1.70
uk-2002	35.60	23.30	14.03	10.31	9.49	**8.74**
Europe	**11.78**	15.98	20.55	19.90	18.97	16.96
MG2	70.67	44.14	24.00	14.84	10.76	**10.69**

Table 4.6 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

Table 4.7 Runtime (in seconds) comparison for the Guided VFF scheme vs. *Ab initio* (Greedy-LU) balancing scheme. All runs were performed on 32 threads of the AMD x86 platform.

Input graph	Guided		*Ab initio*
	Init. coloring	VFF	(Greedy-LU)
CNR	0.15	0.14	0.19
Channel	1.29	1.70	3.21
uk-2002	5.74	8.74	10.1
Europe	8.69	16.96	21.85
MG2	27.75	10.69	36.88

Table 4.8 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		Mod.	Run-time			Mod.
	Init. coloring	Comm. Dec.		Init. coloring	*VFF balancing*	Comm. Dec.	
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 4.9 Statistics on structure of the bipartite real-world graphs $G = (V_1, V_2, E)$ used in our study. Recall that Δ corresponds to maximum degree.

Input	$\lvert V_1 \rvert$	$\lvert V_2 \rvert$	$\lvert E \rvert$	$\Delta(V_1)$	$\Delta(V_2)$
KDD-A	8.4×10^6	2.0×10^7	3.0×10^8	85	6.7×10^4
KDD-B	1.9×10^7	2.9×10^7	5.6×10^8	75	3.0×10^6
LPCRE-B	9.6×10^3	7.7×10^4	2.6×10^5	844	14
LPCRE-D	8.0×10^3	7.3×10^4	2.4×10^5	808	13
Gene-Drug	3.0×10^3	1.4×10^4	2.9×10^4	283	144
Host-Pathogen	8.9×10^3	6.3×10^3	2.2×10^4	166	1631

Table 4.10 Parallel run-times (in seconds) of unbalanced and balanced {VFF, Sched-Rev} partial distance-2 schemes on the AMD platform. 32 cores were used for the KDD inputs, and 16 cores were used for the remaining inputs (due to smaller size). All runs were performed to color vertices in V_2.

Input graph	Init. coloring	VFF balancing	Sched-Rev (1 round)	Sched-Rev (3 rounds)
KDD-A	213	**130**	117	305
KDD-B	606	**354**	154	424
LPCRE-B	0.08	**0.1**	0.12	0.29
LPCRE-D	0.08	**0.09**	0.11	0.24
Gene-Drug	0.008	**0.02**	0.04	0.07
Host-Pathogen	0.005	**0.01**	0.03	0.04

Table 4.11 Run-time evaluation of our extensions (weighted and LB-based) in comparison to the VFF balancing scheme and the Greedy-FF (initial coloring) scheme. All executions were performed on 16 threads of our x86 platform.

Input	Steps	Greedy-FF	VFF	**Weighted**	**LB-based** (γ_ℓ =1024)
CNR	Balancing	0	0.15	0.15	**0.07**
	Comm. Det.	1.30	1.18	1.42	1.25
MG2	Balancing	0	10.33	10.94	**7.86**
	Comm. Det.	204.72	122.67	123.31	149.45
uk-2002	Balancing	0	9.42	9.75	**3.59**
	Comm. Det.	50.91	41.25	44.05	51.08

CHAPTER 5

CONCLUSION

In this dissertation, we addressed two graphs operations: community detection and balanced graph coloring. For community detection, we introduced effective heuristics for parallelizing an important and widely used community detection method called the Louvain method. As for balanced graph coloring, we provided a thorough treatment of the problem, with our contributions spanning algorithm development, parallelization, and application.

5.1 Community Detection

We attempted to address the dual objectives of maximizing concurrency, and retaining the quality with respect to the serial implementation. To this end, we made two main contributions in this dissertation. First, we presented a detailed discussion of the challenges pertaining to parallelization of the Louvain algorithm for community detection, and described effective heuristics to extract parallelism from the algorithm. Second, we empirically supported the observations with a set of carefully conducted experiments using 11 real-world networks representing a diverse set of application domains. Compared to the serial Louvain implementation (Blondel et al., 2008),

our parallel implementation is able to produce community outputs with a higher modularity for many of the inputs tested, in comparable number of iterations, while providing real speedups of up to 16x using 32 threads. In addition, our parallel implementation was able to scale linearly up to 16 threads for larger inputs.

5.2 Balanced Coloring

We presented multiple balancing schemes and developed parallel implementations on conventional multicores and an emerging manycore platform (Tilera). We 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 dissertation 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. We believe that the mathematical discussion, heuristics, and experimental evidence of these two graph operations provided in this dissertation will benefit a wide range of researchers dealing with increasingly larger data sets and continually weaker serial hardware performance.

5.3 Future Works

In this section we propose multiple future work elements.

i) **Modularity:** As mentioned in Section 3.1 and Section 3.2, modularity is the we use to measure the quality of communities detected. However, it is not a perfect metric because issues such as resolution limit have been identified (Fortunato

and Barthelemy, 2007). Therefore, a future work is to investigate the definitions of a more suitable metric that could be used to better assess the quality of communities in the context of specific application domains.

ii) **Vertex ordering:** As mentioned in Section 3.3, the *Louvain* method is an multi-phase iterative process where operations are performed on each of the vertex. In Blondel's paper (Blondel et al., 2008), they mention that the order of vertex processing has negligible influence on the final output modularity. However, we found that not to be the case for some of the inputs tested—i.e., the order indeed changes the community configurations (with or without changing the modularity). Therefore the impact of vertex ordering on the overall quality and convergence for an iterative algorithm like Louvain needs to be studied more carefully.

iii) **Extensions to vertex following:** As mentioned in Section 3.5.3, the vertex following heuristic is only a partial solution toward reducing the runtime. More specifically, in its current flavor, the vertex following heuristic folds single-degree vertices into their neighboring communities, thereby eliminating them from the iterative steps. Instead of limiting the heuristic's reach to only the single-degree vertices, there is a way to extend it to longer paths until such a point where further collapsing of subpaths into communities could potentially decrease the net modularity. This idea can be thought of in a similar vein as the k-core decomposition routine (Alvarez-Hamelin et al., 2005), except without prior information on the value of a suitable k or fixing the value of k. However more research is needed to expand this idea and make it effective in practice.

iv) **Dynamic communities:** As mentioned in Section 3.7, we proposed to incrementally detect dynamic communities based on the notion of communities derived from individual timesteps. However, a more complete and mathematically

rigorous definition would directly account for a dynamic community to span across variable number of timesteps. More investigation is necessary to establish the relationship between incremental detection and a holistic detection of communities directly from a dynamic graph.

v) **Distributed implementation:** To enhance the reach of our current implementation to extreme-scale graphs, it is important to scale up the amount of memory and processing power available for computation. To achieve this, therefore, we can think of implementing our community detection and coloring operations on distributed memory platforms. We have a preliminary version of a distributed implementation for community detection. The current version does not however, provide the benefits associated with using the coloring heuristic. We are currently exploring alternative ways to derive maximum concurrency on a distributed platform.

BIBLIOGRAPHY

Alvarez-Hamelin, José Ignacio et al. (2005). "K-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases". In: *arXiv preprint cs/0511007*.

Bader, David and Joe McCloskey (2009). "Modularity and graph algorithms". In: *SIAM AN10 Minisymposium on Analyzing Massive Real-World Graphs*, pp. 12–16.

Bader, David A et al. (2012). "Graph partitioning and graph clustering". In: *10th DIMACS Implementation Challenge Workshop*.

Barabási, Albert-László (2009). "Scale-free networks: a decade and beyond". In: *science* 325.5939, pp. 412–413.

Batagelj, Vladimir and Matjaž Zaveršnik (2002). "Generalized cores". In: *arXiv preprint cs/0202039 (2002)*.

Bell, Shane et al. (2008). "Tile64-processor: A 64-core soc with mesh interconnect". In: *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*. IEEE, pp. 88–598.

Berry, Jonathan W et al. (2011). "Tolerating the community detection resolution limit with edge weighting". In: *Physical Review E* 83.5, p. 056119.

Blazewick, J. et al. (2001). *Scheduling computer and manufacturing processes*. 2nd edition. Berlin: Springer.

Blondel, Vincent et al. (2008). "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment*, P10008.

Bodleander, H.L. and F.V. Fomin (2005). "Equitable colorings of bounded treewidth graphs". In: *Theoret. Comput. Sci.* 349.1, pp. 22–30.

Brandes, Ulrik et al. (2008). "On modularity clustering". In: *Knowledge and Data Engineering, IEEE Transactions on* 20.2, pp. 172–188. (Visited on 10/10/2013).

Catalyurek, Umit et al. (2012). "Graph coloring algorithms for multi-core and massively multithreaded architectures". In: *Parallel Computing* 38.11, pp. 576–594.

Chavarría-Miranda, Daniel, Mahantesh Halappanavar, and Ananth Kalyanaraman (2014). "Scaling graph community detection on the Tilera Many-core architecture". In: *IEEE International Conference on High Performance Computing (HiPC)*. Goa, India, p. 11.

Chen, B.L. and K.W. Lih (1994). "Equitable coloring of trees". In: *J. Combin. Theory Ser. B* 61, pp. 83–87.

Connor, Richard C, Michael R Heithaus, and Lynne M Barre (2001). "Complex social structure, alliance stability and mating access in a bottlenose dolphin superalliance". In: *Proceedings of the Royal Society of London B: Biological Sciences* 268.1464, pp. 263–267.

Culberson, Joseph and Feng Luo (1996). "Exploring the k-colorable landscape with iterated greedy". In: *Cliques, coloring and satisfiability: Second DIMACS implementation challenge*. Ed. by DS Johnson and MA Trick. American Math. Society, pp. 245–284.

Daily, Jeff et al. (2015). "A work stealing based approach for enabling scalable optimal sequence homology detection". In: *Journal of Parallel and Distributed Computing* 79–80. Special Issue on Scalable Systems for Big Data Management and Analytics,

pp. 132 –142. ISSN: 0743-7315. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001518.

Dániel, MARX (2004). "Graph colouring problems and their applications in scheduling". In: *Periodica Polytech., Electr. Eng* 48.1-2, pp. 11–16.

Davis, Timothy A. and Yifan Hu (2011). "The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1, 1:1–1:25.

DIMACS10. *The 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering*. URL: http://www.cc.gatech.edu/dimacs10/ (visited on 10/11/2013).

Duraisamy, Karthi et al. (2015). "High performance and energy efficient wireless NoC-enabled multicore architectures for graph analytics". In: *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE Press, pp. 147–156.

— (2016). "High-Performance and Energy-Efficient Network-on-Chip Architectures for Graph Analytics". In: *ACM Transactions on Embedded Computing Systems (TECS)* 15.4, p. 66.

E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson (1997). "Approximation Algorithms for Bin Packing: A Survey". In: *Approximation Algorithms for NP-hard Problems*. Ed. by Dorit Hochbaum. PWS Publishing Company, pp. 46–86.

Erdös, P., A. Rényi, and V.T. Sós, eds. (1970). *Combinatorial Theory and its Application*. London: North-Holland.

Erdös, Paul and Alfréd Rényi (1960). "On the evolution of random graphs". In: *Publ. Math. Inst. Hung. Acad. Sci* 5.17-61, p. 43.

Fortunato, Santo (2010). "Community detection in graphs". In: *Physics Reports* 486.3-5, pp. 75–174. ISSN: 03701573. (Visited on 07/09/2012).

Fortunato, Santo and Marc Barthelemy (2007). "Resolution limit in community detection". In: *Proceedings of the National Academy of Sciences* 104.1, pp. 36–41.

Furmańczyk, Hanna (2004). "Equitable coloring of graphs". In: *Graph Colorings*. Ed. by Marek Kubale. Providence, Rhode Island: American Mathematical Society, pp. 35–53.

Gebremedhin, Assefaw, Fredrik Manne, and Alex Pothen (2005). "What color is your Jacobian? Graph coloring for computing derivatives". In: *SIAM Review* 47.4, pp. 629–705.

Gebremedhin, Assefaw Hadish and Fredrik Manne (2000). "Scalable parallel graph coloring algorithms". In: *Concurrency - Practice and Experience* 12.12, pp. 1131–1146.

Girvan, Michelle and Mark EJ Newman (2002). "Community structure in social and biological networks". In: *Proceedings of the National Academy of Sciences* 99.12, pp. 7821–7826. ISSN: 0027-8424.

Granovetter, Mark S (1973). "The strength of weak ties". In: *American journal of sociology*, pp. 1360–1380.

Griffith, Malachi et al. (2013). "DGIdb: mining the druggable genome". In: *Nature methods* 10.12, pp. 1209–1210.

Hajnal, A. and E. Szemerédi (1970). "Proof of a conjecture of P. Erdös". In: *Combinatorial Theory and its Application*. Ed. by P. Erdös, A. Rényi, and V.T. Sós. London: North-Holland, pp. 601–623.

Halappanavar, Mahantesh (2009). "Algorithms for vertex-weighted matching in graphs (Ph.D. thesis)". PhD thesis. Norfolk, VA: Old Dominion University.

Hanisch, Daniel et al. (2002). "Co-clustering of biological networks and gene expression data". In: *Bioinformatics* 18.suppl 1, S145–S154.

Hendrickson, Bruce and Tamara G Kolda (2000). "Graph partitioning models for parallel computing". In: *Parallel Computing* 26.12, pp. 1519–1534. ISSN: 0167-8191.

Jensen, Tommy R and Bjarne Toft (2011). *Graph coloring problems*. Vol. 39. John Wiley & Sons.

Jones, Mark T. and Paul Plassmann (1993). "A parallel graph coloring heuristic". In: *SIAM Journal of Scientific Computing* 14, pp. 654–669.

Kalyanaraman, Ananth et al. (2016). "Fast Uncovering of Graph Communities on a Chip: Toward Scalable Community Detection on Multicore and Manycore Platforms". In: *Foundations and Trends® in Electronic Design Automation* 10.3, pp. 145–247.

Kernighan, Brian W and Shen Lin (1970). "An efficient heuristic procedure for partitioning graphs". In: *The Bell system technical journal* 49.2, pp. 291–307.

Khan, Arif et al. (2016). "Efficient approximation algorithms for weighted b-matching". In: *SIAM Journal on Scientific Computing* 38.5, S593–S619.

Kubale, Marek, ed. (2004). *Graph Colorings*. Providence, Rhode Island: American Mathematical Society.

Louvain. *findcommunities*. URL: https://sites.google.com/site/findcommunities/ (visited on 10/11/2013).

Lu, Hao, Mahantesh Halappanavar, and Ananth Kalyanaraman (2015). "Parallel heuristics for scalable community detection". In: *Parallel Computing* 47, pp. 19–37.

Lu, Hao et al. (2014). "Parallel heuristics for scalable community detection". In: *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, pp. 1374–1385.

Lu, Hao et al. (2015). "Balanced coloring for parallel computing applications". In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, pp. 7–16.

Lu, Hao et al. (2016). "Algorithms for Balanced Graph Colorings with Applications in Parallel Computing". In: *IEEE Transactions on Parallel and Distributed Systems*.

Manne, Fredrik and Erik Boman (2005). "Balanced greedy colorings of sparse random graphs". In: *The Norwegian Informatics Conference, NIK'2005*, pp. 113–124.

Melhem, R.G. and V.S. Ramarao (1988). "Multicolor reorderings of sparse matrices resulting from irregular grids". In: *ACM Transaction of Mathematical Software* 14, pp. 117–138.

Meusel, Robert et al. *Web Data Commons - Hyperlink Graphs*. URL: http://webdatacommons.org/hyperlinkgraph (visited on 2014).

Meyer, Walter (1973). "Equitable coloring". In: *The American Mathematical Monthly* 80.8, pp. 920–922.

Milgram, Stanley (1967). "The small world problem". In: *Psychology today* 2.1, pp. 60–67.

Newman, M. E. J (2003). "The structure and function of complex networks". In: *SIAM Review* 45, pp. 167–256.

— (2004a). "Analysis of weighted networks". In: *Phys. Rev. E* 70.5, p. 056131.

Newman, M. E. J. and M. Girvan (2004). "Finding and evaluating community structure in networks". In: *Physical Review E* 69.2, p. 026113.

Newman, Mark EJ (2004b). "Fast algorithm for detecting community structure in networks". In: *Physical review E* 69.6, pp. 66–133.

Papadopoulos, Nicholas et al. (2012). "Binding and neutralization of vascular endothelial growth factor (VEGF) and related ligands by VEGF Trap, ranibizumab and bevacizumab". In: *Angiogenesis* 15.2, pp. 171–185.

Penrose, Mathew (2003). *Random Geometric Graphs*. Oxford University Press. ISBN: 0198506260.

Pesantez-Cabrera, Paola and Ananth Kalyanaraman (2016). "Detecting Communities in Biological Bipartite Networks". In: *Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*. ACM, pp. 98–107.

Pommerell, C., M. Annaratone, and W. Fichtner (1992). "A set of new mapping and coloring heuristics for distributed-memory parallel processors". In: *SIAM J. Sci. Statist. Comput.* 13, pp. 194–226.

Reddy, P Krishna et al. (2002). "A graph based approach to extract a neighborhood customer community for collaborative filtering". In: *International Workshop on Databases in Networked Information Systems*. Springer, pp. 188–200.

Robert K. Gjertsen, Jr., Mark T. Jones, and Paul Plassmann (1996). "Parallel heuristics for improved, balanced graph colorings". In: *Journal of Parallel and Distributed Computing* 37, pp. 171–186.

Scherrer, Chad et al. (2012). "Feature clustering for accelerating parallel coordinate descent". In: *Advances in Neural Information Processing Systems*, pp. 28–36.

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

Solomonoff, Ray and Anatol Rapoport (1951). "Connectivity of random nets". In: *The bulletin of mathematical biophysics* 13.2, pp. 107–117.

Steinhaeuser, Karsten, Nitesh V Chawla, and Auroop R Ganguly (2011). "Complex networks as a unified framework for descriptive analysis and predictive modeling in climate science". In: *Statistical Analysis and Data Mining* 4.5, pp. 497–511.

Sylvester, John Joseph (1878). "Chemistry and algebra". In: *Nature* 17, p. 284.

Traag, Vincent A, Paul Van Dooren, and Y Nesterov (2011). "Narrow scope for resolution-limit-free community detection". In: *Physical Review E* 84.1, p. 016114.

Tucker, A. (1973). "Perfect graphs and an application to optimizing municipal services". In: *SIAM Review* 15, pp. 585–590.

Wang, W. and K. Zhang (2000). "Equitable colorings of line graphs and complete r-partite graphs". In: *Systems Sci. Math. Sci.* 13.2, pp. 190–194.

Wardeh, Maya et al. (2015). "Database of host-pathogen and related species interactions, and their global distribution". In: *Scientific data* 2.

Watts, Duncan J and Steven H Strogatz (1998). "Collective dynamics of small-worldnetworks". In: *nature* 393.6684, pp. 440–442.

Wu, Changjun, Ananth Kalyanaraman, and William R Cannon (2012). "pGraph: efficient parallel construction of large-scale protein sequence homology graphs". In: *Parallel and Distributed Systems, IEEE Transactions on* 23.10, pp. 1923–1933. ISSN: 1045-9219.

Yap, H.P. and Y. Zhang (1998). "Equitable colorings of planar graphs". In: *J. Combin. Math. Combin. Comput.* 27, pp. 97–105.

Yu, Hsiang-Fu et al. (2011). "Feature engineering and classifier ensemble for KDD Cup 2010". In: *In JMLR Workshop and Conference Proceedings*.

Yule, G Udny (1925). "A mathematical theory of evolution, based on the conclusions of Dr. JC Willis, FRS". In: *Philosophical transactions of the Royal Society of London. Series B, containing papers of a biological character* 213, pp. 21–87.