The Pennsylvania State University
The Graduate School
College of Engineering

IRREGULAR GRAPH ALGORITHMS ON MODERN MULTICORE, MANYCORE, AND DISTRIBUTED PROCESSING SYSTEMS

A Dissertation in
Computer Science & Engineering
by
George M. Slota

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2016

The dissertation of George M. Slota was reviewed and approved* by the following:

Kamesh Madduri
Assistant Professor of Computer Science & Engineering
Dissertation Advisor, Chair of Committee

Mahmut Kandemir
Professor of Computer Science & Engineering

Wang-Chien Lee
Associate Professor of Computer Science & Engineering

Soundar Kumara
Professor of Industrial & Manufacturing Engineering

Siva Rajamanickam
Senior Member of Technical Staff, Sandia National Laboratories
Special Member

Raj Acharya
Professor & Director, School of Electrical Engineering and Computer Science
Department Chair

*Signatures are on file in the Graduate School.

Abstract

Graph analysis is the study of real-world interaction data, be it through biological or chemical interaction networks, human social or communication networks, or other graph-representable datasets pervasive throughout the social and physical sciences. Due to increasing data sizes and complexities, it is important to develop efficient and scalable approaches for the algorithms, tools, and techniques used to study such data. Efficient utilization of the increasing heterogeneity and complexity of modern high performance computing systems is another major consideration for these efforts.

The primary contributions of this thesis are as follows: First, parallel and scalable solutions to several basic graph analytics are presented. An implementation of the *color-coding* algorithm for subgraph isomorphism is introduced as Fascia (Fast Approximate Subgraph Counting and Enumeration). Using several optimizations for work avoidance, memory usage reduction, and cache/data movement efficiency, Fascia demonstrates up to a five orders-of-magnitude per-core speedup relative to prior art. Fascia is able to calculate the counts of subgraphs up to 10 vertices on multi-billion edge graphs in minutes on a modest 16 node cluster and use these counts for a variety of analytics. Using Fascia's baseline approach, FastPath is also introduced to find minimum weight paths in weighted networks. The Multistep method is next introduced as an approach for graph connectivity, weak connectivity, and strong connectivity, with a generalization of Multistep also presented for graph biconnectivity. The Multistep approaches are shown to demonstrate a 2-7× mean speedup relative to the prior state-of-the-art. A graph partitioner called PuLP (Partitioning using Label Propagation) is also introduced along with a general distributed graph layout strategy, DGL. PuLP was specifically designed to partition small-world graphs having skewed degree distributions, such as social interaction networks and web graphs. PuLP is able to partition such graphs an order of magnitude faster and with a fraction of the memory of other comparable partitioners (ParMETIS, PT-Scotch) while giving comparable partitions in terms of cut quality and balance. Additionally, this thesis

presents how using techniques derived from these efforts, a suite of distributed graph analytics could be implemented and applied to the largest publicly-available web crawl of 3.5 billion pages and 130 billion links. End-to-end execution of analysis using these implementations completing in 20 minutes on only 256 nodes of the Blue Waters supercomputing system.

Throughout this thesis, analyses of the algorithms and subroutines that comprise the MULTISTEP, FASCIA/FASTPATH, and PULP/DGL implementations is undertaken. Common optimizations are then identified (e.g., multiple levels of queues to match the memory hierarchy, techniques for non-blocking and asynchronous updates to shared data, efficient distributed communication patterns, among others) and their effects on performance are quantified. It is demonstrated how the optimization techniques can be utilized when processing under the higher degree of parallelism available in modern manycores (GPUs, Intel Xeon Phis) as well as how the techniques can be extended for more general-purpose graph processing in both the shared- and distributed-memory spaces. Also under consideration is the state of current hardware trends, with the goal of identifying how to modify and extend these general optimizations for forthcoming high performance computing architectures. Additionally, new optimizations and potential further research areas are introduced which might also be applicable for accelerating graph processing on these future systems.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

I would first and foremost like to thank my advisor Kamesh Madduri for giving me the opportunity to return to Penn State and complete my PhD. His assistance and guidance has made this research possible. I also thank him for guiding me towards challenging problems to work on and for inspiring my interest in the research contained in this thesis. He has also gone above and beyond in assisting with my professional development and opening up many new opportunities.

I would also like to thank Siva Rajamanickam of Sandia National Labs for acting as my mentor during the past three years. He has also done exceptionally well at guiding my development as a researcher. He has also made available the personal and technical expertise and equipment of Sandia, which has undoubtedly greatly helped with the bulk of work that went into this document.

I'd further like to thank the National Center for Supercomputing Applications and the Blue Waters project for offering me the Blue Waters Graduate Fellowship during my previous academic year as well as continued use of the Blue Waters supercomputing system. This assistance has enabled a considerable amount of research that would have otherwise not been possible.

Thanks are also extended to my mom, dad, and other family and friends. There is no doubt that I would not have made it through almost nine years of undergraduate and graduate schooling without your continued emotional support.

In addition, the following acknowledgment towards the grants and funding has made my degree and this research possible:

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070, ACI-1238993, and ACI-1444747) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This work is also supported by NSF grants ACI-1253881, CCF-1439057, and the DOE Office of Science through the FASTMath SciDAC Institute. Sandia National Laboratories is a multi-program laboratory

managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Dedication

To Cheebs.

Chapter 1
Introduction

This thesis explores the topic of *graph analytics* through novel algorithm development, implementation methodologies for modern systems, and applications from a broad reach of disciplines. *Graph* in this context refers to the mathematical construct which models pairwise relationships between discrete objects. Graph analytics forms an essential aspect of the study of biology, chemistry, physics, social networks, and numerous other sources of graph-structured data within the scientific realm [1–5]. The mathematical and computational difficulties associated with analyzing highly dimensional and complex graphs is widely recognized, and it is correspondingly listed as one of DARPA's 23 toughest mathematical challenges [6]. The general problem of computation at a massive scale is also considered on that list. Modern security concerns additionally create a definite need for real-time analysis of large and variable data [7]. Developing fast and scalable approaches for graph analysis is necessary in order to tackle these challenges.

This thesis will explore the development of algorithms and implementations for basic graph problems such as subgraph isomorphism and its variants, graph connectivity and traversal, and graph partitioning, as they form a foundation upon which a lot of graph analysis techniques are built. A major theme of this thesis is graph implementation techniques for modern high performance computing architectures, such as multicore and manycore processors and highly parallel distributed systems. The overarching goal of this thesis was to determine how to best improve the parallel efficiency of irregular graph computations through exploiting the architectural features of modern hardware.

1.1 Overview of Graph Problems Considered

One of the longstanding computational problems in graph theory is subgraph isomorphism [8], which is determining if a smaller graph is contained within a larger one. Subgraph isomorphism and its variants are extremely useful for analyzing large graphs [2]. For instance, counting and enumerating the occurrences of a repeating substructures in a graph is how one might determine the number and location of certain interaction types within a social network or financial transaction network [3]. Finding frequently occurring subgraphs, or **motifs**, enables important information about the data to be retrieved without any prior knowledge as to the structure of the data [4]. **Anomaly detection** allows the discovery of unwanted or irregular patterns in the data, such as erroneous operation of an electrical network or the attempted penetration of a secure computer system [5]. One of the primary difficulties with analyzing large graphs is that several of the aforementioned problems are considered NP-Hard. Using heuristic-based algorithms, it is not uncommon for graphs with only millions of edges and nodes to take hours to search, even using fast and massively parallel algorithms [9]. However, state-of-the art heuristics and approximation algorithms still demonstrate considerable promise for these problems; consider that on even moderate scale problems of hundreds of thousands of edges, an approximation algorithm can improve execution times for subgraph counting from over an hour and a half in the naïve case to under half of a second with minimal error [10]. As such, one of early goals of this thesis effort was to develop a subgraph counting tool and suite of analytics to allow fast and memory efficient subgraph-based analysis of modern real-world datasets.

Another important graph-theoretic set of problems is those that deal with graph connectivity. Graph connectivity and its related problems, such as weak connectivity, strong connectivity, and biconnectivity, are useful preprocessing or analytic steps for web and social graph analysis [11–14], model verification [15], and scientific computing [16]. Although these connectivity problems have solutions computable through optimal linear time serial algorithms [17, 18], these algorithms involve depth-first search as a central subroutine, which isn't parallelizable and doesn't scale to modern real-world graph sizes. In previous decades, a number of efficient parallel algorithms were introduced for these problems [15, 19, 20]. However, for problems on large real-world irregular data, such as computing the strongly

connected components of a crawl of the Italian .it domain, parallel execution times using a prior approach are nearly half an hour; using a modern approach, this time can be reduced to under two seconds [21]. Therefore, a second area of this thesis research was to develop new approaches for these problems with modern multicore architectures in mind to enable scalable analysis and preprocessing of such graphs.

One major problem involving graph analytics is that, due to the structure common to most emerging real-world graphs (poor locality of memory access, high complexity, etc.), the efficient parallelization of most graph algorithms on emerging platforms is very challenging. Advanced hardware architectures utilizing many simple cores in a shared memory environment, such as GPUs or the Intel Xeon Phi, pose a large set of challenges for algorithm designers to overcome [22–26]. Since many-core GPUs and coprocessors now power the world's fastest supercomputers alongside multicore CPUs, the importance of tackling such challenges is considerable. As such, this thesis considered how best to design graph algorithms that are effective on new many-core architectures and heterogeneous environments. This thesis includes a thorough investigation of the performance of graph algorithms, such as graph traversal, components decomposition, and color propagation, running on multicore and manycore systems and an analysis in how to best exploit the highly threaded and computationally-focused nature of modern manycore hardware to ensure high performance of these and other memory-intensive algorithms.

When analyzing massive graphs in distributed memory architectures, an initial and highly important question is, *how must one organize the data structures representing a real small-world graph (web crawl, social network, etc.) on a cluster of multicore nodes, with each node having 32-64 GB memory*? Fully replicating the data structures on each process is infeasible for massive graphs. A graph topology-agnostic partitioning will lead to severe load imbalances when processing graphs with skewed degree distributions. Additionally, current partitioning tools [27–29] often don't optimize for the objectives and constraints important for the high performance of small-world graph computations, nor do they scale to the sizes of modern real-world datasets. Further, intra-node vertex ordering after partitioning can also have a considerable effect on execution times of complex analytics [30]. This thesis investigated the effects of small-world graph layout on graph analytic performance and subsequently developed partitioning and ordering strategies under such considerations.

Recently, a large number of openly-available parallel graph-analytic frameworks have been introduced with the stated goal of high scalability (GraphX [31], GraphLab and its variants [32–34], FlashGraph [35], Giraph [36], etc. [37–39]). However, in practice, most of these frameworks fail to demonstrate performance better than optimized serial code in a large cluster while failing to even being able process graphs larger than would fit in memory of a typical workstation (GraphX, GraphLab, and most others) [40], demonstrate strong performance but require specialized hardware (FlashGraph), or only give reasonable performance at a very large scale (Giraph) [41]. Combing lessons learned throughout earlier research efforts on optimizing shared- and distributed-memory graph computations, a final aspect of this thesis deals with creating a general-purpose distributed graph computation approach that shows performance and scalability at all computational scales from a single node to thousands of nodes.

The fact that graph-structured data is so universal means that these research directions are useful to almost all fields within the social and physical sciences. Due to the large graphical datasets commonly collected for many fields, it is necessary to develop new parallel graph algorithms and techniques in order to allow analysis of these massive data sets on modern multicore, manycore, and distributed processing systems. This thesis work aims towards a better understanding of the world through enabling the analysis of large amounts of scientific graphical data.

1.2 Thesis Organization

The following content of this thesis document is organized into 10 chapters in about 4 related sections. Chapters 2-5 discuss subgraph counting and related problems on multicore and distributed systems as well as some of its applications; Chapters 6-8 discuss graph connectivity algorithms and their implementation on multicore and manycore processors; Chapters 9 and 10 discuss small-world graph partitioning and vertex ordering approaches for graph layout in distributed memory; Chapter 11 discusses implementation and optimization approaches for massive scale graph analysis and its application towards analyzing the current (as of early 2016) largest publicly available web crawl.

More specifically, Chapters 2-5 examines subgraph counting, subgraph counting applications for graph analysis, minimum and maximum weighted path finding,

and graph alignment. The content of these chapters is composed from the following papers and presentations:

- **Fast Approximate Subgraph Counting and Enumeration**, published at the *2013 International Conference on Parallel Processing (ICPP13)*. [10]
- **Complex Network Analysis using Parallel Approximate Motif Counting**, published at the *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS14)*. [42]
- **Parallel Color-Coding**, published in *Parallel Computing, Systems & Applications*. [43]
- **Fascia: Fast Approximate Subgraph Counting and Enumeration**, poster presentation at the *2013 SIAM Workshop on Network Science (NS13)*.
- **Characterizing Biological Networks Using Subgraph Counting and Enumeration**, presented at the *16th SIAM Conference on Parallel Processing for Scientific Computing (PP14)*.

Chapters 6-8 look at the problems of connectivity, weak and strong connectivity, biconnectivity, and the implementation of a subset of these and related algorithms in manycore GPU and Xeon Phi environments. The content of these chapters is composed from following papers and presentations:

- **BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems**, published at the *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS14)*. [21]
- **Simple Parallel Biconnectivity Approaches for Multicore Platforms**, published at the *IEEE Conference on High Performance Computing (HiPC 2014)*. [44]
- **High-performance Graph Analytics on Manycore Processors**, published at the *29th IEEE International Parallel and Distributed Processing Symposium (IPDPS15)*. [45]
- **Computing Strongly Connected Components in Modern Architectures**, presented at the *2013 SIAM Annual Meeting (AN13)*.

- **Parallel Strongly Connected Components in Shared Memory Architectures**, presented at the *16th SIAM Conference on Parallel Processing for Scientific Computing (PP14)*.

Chapters 9-10 introduce the PuLP partitioner for complex partitioning of small-world graphs and the DGL layout strategy for distributed graph processing. These chapters were composed from the following papers and presentations:

- **Complex Network Partitioning Using Label Propagation**, currently under review. [46]
- **PuLP: Scalable Multi-Objective Multi-Constraint Partitioning for Small-World Networks**, published at the *2nd IEEE Conference on Big Data (BigData 2014)*. [47]
- **Distributed Graph Layout for Scalable Small-World Network Analysis**, currently under review. [30]
- **PuLP: Fast and Simple Complex Network Partitioning**, presented at the *Dagstuhl Seminar 14461, High-performance Graph Algorithms and Applications in Computational Science*.
- **Parallel Complex Network Partitioning**, poster presentation at the *2014 IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC14)*.

Chapter 11 details an end-to-end methodology for distributed processing of large-scale real world graphs, an implementation methodology for several distributed and scalable graph analytics, and applying both for analyzing real-world data. The content of this chapter is composed from the following paper:

- **A Case Study of Complex Graph Analysis in Distributed Memory: Implementation and Optimization**, to be published at the *30th IEEE International Parallel and Distributed Processing Symposium (IPDPS16)*. [48]

Chapter 2
Fascia Subgraph Counting

2.1 Introduction

This chapter describes the implementation of a known fast approximate algorithm for the problem of subgraph counting and enumeration. We perform a parallel implementation designed for shared-memory multicore processors and describe several optimization techniques applicable both to our specific code as well as is generalizable to other algorithms for which the approximation technique is applicable. Our approaches run both orders-of-magnitude faster and use orders-of-magnitude less memory than prior work, which allows the analysis of networks much larger than previously possible.

2.2 Color-coding and Subgraph Counting

The color-coding method is a simple and elegant graph-theoretic strategy that gives fixed parameter tractable algorithms for several NP-hard and NP-complete problems. Color-coding was first proposed by Alon, Yuster, and Zwick [49]. In this chapter, we present efficient shared- and distributed-memory parallelizations of this strategy for subgraph counting. We use new data structures and optimizations to reduce peak memory utilization and inter-processor communication. We also create the software tool Fascia, which uses parallel color-coding and subgraph counting to solve several subgraph-based graph analytic problems.

Subgraph counting is the problem of counting the number of occurrences of a *template* or subgraph within a larger graph. This problem is a variant to the

classical NP-complete [50] subgraph isomorphism problem. Other related problems, such as subgraph enumeration, tree isomorphism, motif finding, frequent subgraph identification, etc. are all fundamental graph analysis methods to identify latent structure in complex data sets. They have applications in bioinformatics [4, 51, 52], chemoinformatics [53], online social network analysis [54], network traffic analysis, and many other areas.

Subgraph counting and enumeration are compute-intensive problems. A naïve algorithm, which exhaustively enumerates all vertices reachable in k hops from a vertex, runs in $O(n^k)$ time, where n is the number of vertices in the network and k is the number of vertices in the subgraph. For large networks, this running time complexity puts a constraint of the size of the subgraph (value of k). If k is larger than 2 or 3, exact counting becomes prohibitively expensive. Thus, there has been a lot of recent work on approximation algorithms. Approaches are generally based on sampling or on exploiting network topology. Sampling-based methods analyze a subset of the network and extrapolate counts based on the observed occurrences and network properties. Some tools based on sampling are MFINDER [55], FANMOD [56], and GRAFT [57]. The other class of methods impose some constraint on the network or transform the network so that the possible search space is restricted. Examples of tools imposing constraints on the network are NEMO [58] and SAHAD [9]. Tools based on the color-coding method belong to the second category, and this forms the basis of our current work.

The color-coding method for this problem uses a dynamic programming scheme to generate an approximate count of a given *non-induced tree-structured* subgraph/template (also referred to as a *treelet*) in $O(m \cdot 2^k)$ time, where m is the number of edges in the network. The algorithm can be informally stated as follows: every node in a network is randomly colored with one of at least k possible colors. The number of *colorful* embeddings of a given input template is then counted, where colorful in this context means that each node in the template embedding has a distinct color. The total embedding count is then scaled by the probability that any given embedding of the template is colorful, in order to generate an approximation for the total number of possible embeddings. This colorful embedding counting scheme avoids the prohibitive $O(n^k)$ bound seen in exhaustive search.

2.3 Related Work

Subgraph counting has recently emerged as a widely-used graph analytic in various domains, especially the biological and social sciences. Pržulj has demonstrated that *graphlets*—all 2-5 vertex induced undirected subgraphs—are a useful analytic for biological network comparisons [51]. Pržulj and Milenkovič et al. have extended this work to several other subgraph-based comparative metrics [1, 59, 60]. Bordino et al. used counts of both small undirected and directed subgraphs, similar to graphlets, to cluster networks of various types (e.g. citation networks, road networks, etc.) [61].

Alon et al. implemented color-coding subgraph counting to demonstrate its applicability for finding large tree-structured motifs in biological networks [52]. Zhao et al. implemented distributed color-coding subgraph counting for large graphs via both MPI and MapReduce, with applications in social network analysis [9, 62]. Color-coding can also be used to count and enumerate cycles, cliques, and bounded treewidth subgraphs [49].

2.4 Color-coding Implementations

2.4.1 Subgraph Counting with Fascia

We first present the algorithmic details of applying the color-coding method for tree-structured subgraph counting [10]. As shown in Algorithm 2.1, there are three main phases in the algorithm: template partitioning, random coloring, and the dynamic programming count phase. The pseudocode for the dynamic programming phase is described in Algorithm 2.2. The coloring and dynamic programming steps are repeated for multiple iterations to estimate the subgraph count. Alon et al. [49] prove that to guarantee a count bound of $\mathcal{C}(1 \pm \epsilon)$ with probability $1 - 2\delta$ ($\mathcal{C}$ being the exact count and ϵ being the error), we would need to run at most $Niter$ iterations, as defined in Algorithm 2.1. Using a topology-aware coloring scheme [63–65], prior work has shown that a tighter upper bound can be obtained. We observe that the number of iterations necessary to produce accurate global counts on large networks is far lower in practice [9,10], and we will also demonstrate this in Section 2.5.

In the input template partitioning phase, a single vertex is first specified to be

Algorithm 2.1 Subgraph counting using color coding.

1: Partition input template T (k vertices) into subtemplates using *single edge cuts*.
2: Determine $Niter \approx \frac{e^k \log 1/\delta}{\epsilon^2}$, the number of iterations to execute. δ and ϵ are input parameters that control approximation quality.
3: **for all** $it = 1$ to $Niter$ **do** ▷ Outer loop parallelism
4: Randomly assign to each $v \in G$ a color between 0 and $k-1$.
5: Use a dynamic programming scheme to count *colorful* occurrences of T.
6: Take average count of all $Niter$ counts to be final count.

the root of the template. A single edge adjacent to the root is cut, creating two children subtemplates. The child subtemplate containing the original root vertex is called the *active child*, with its root specified again as the original root vertex. The other child will be termed as the *passive child*, with its root as the vertex that was connected to the original root vertex through the edge that was cut. We now have two rooted subtemplates. We recursively continue to cut these subtemplates down to single vertices, keeping track of the *partitioning tree*, where each subtemplate greater than one vertex in size has both an active and passive child subtemplate. Every subtemplate has a parent. This tree can be traced from the bottom up to the original template, which is how we will perform the dynamic programming phase of the color coding algorithm. We also sort them in the order in which the subtemplates are accessed, in order to reduce memory usage.

The graph G is next randomly colored. For every vertex v, we assign a color between 0 and $k-1$, where k is the maximum number of colors. k needs to be greater than or equal to the number of vertices in T. We will consider k equal to the size of T now for simplicity. It has been demonstrated that higher values of k can decrease the required iterations for a given error bound [66]. However, note that this considerably increases memory requirements as well.

Consider first a naïve table structure. We need to be able to store non-zero counts for every vertex and for all possible *color sets*. For a given subtemplate S_i of size h, a color set can be considered the mapping of h unique color values to each vertex in S_i. We create a three dimensional tabular structure and initialize all values to zero. We can then proceed to the inner loops, which contain the dynamic programming-based counting step of the algorithm.

Algorithm 2.2 details the inner nested loops that we have for the algorithm. The

outermost loop will perform, in order, the bottom-up count for each subtemplate, tracing along the partition tree that we previously created. For every subtemplate, we will then consider every single vertex $v \in G$. If our subtemplate is of size 1, we know that its count at v is 0 for all possible k color sets of a single vertex, except for the color set that consists of the color equal to the color randomly assigned to v, where it is 1.

Algorithm 2.2 The dynamic programming step in FASCIA.

1: **for all** sub-templates S_i created from partitioning T, in reverse order of their partitioning **do**
2: **for all** vertices $v \in G$ **do** ▷ Inner loop parallelism
3: **if** S_i consists of a single vertex **then**
4: table$[S_i][v][color\ of\ v] \leftarrow 1$
5: **else**
6: S_i consists of active child a_i and passive child p_i
7: **for all** color sets C of unique values mapped to S_i **do**
8: $count \leftarrow 0$
9: **for all** $u \in N(v)$, where $N(v)$ is the neighborhood of v **do**
10: **for all** C_a and C_p created by uniquely splitting C **do**
11: $count \leftarrow count$ + table$[a_i][v][C_a]$·table$[p_i][u][C_p]$
12: table$[S_i][v][C] \leftarrow count$
13: $templateCount \leftarrow \sum_v \sum_C$ table$[T][v][C]$
14: $P \leftarrow$ probability that the template is colorful
15: $\alpha \leftarrow$ number of automorphisms of T
16: $finalCount \leftarrow \frac{1}{P \cdot \alpha} \cdot templateCount$

If the size of the subtemplate is greater than 1, we know that it must have an active (a_i) and passive (p_i) child. We then look at all possible color sets C of size h with unique values. The count for this color set at v, which we will later store in our table at table$[S_i][v][C]$, is initialized to zero. Next, we will consider for every neighbor, u, of v, the counts of a_i rooted at v and p_i rooted at u. We will then split C into C_a and C_p, which are the mappings onto the active and passive child of the colors in C. The count for S_i rooted at v with color set C is then the sum over all u and over all possible C_a and C_p of table$[a_i][v][C_a]$·table$[p_i][u][C_p]$.

Once we have run through as many iterations as initially specified, we can then take the average over all counts to be our estimate for the total number of embeddings in the graph. We return this value and the algorithm is complete.

2.4.2 Color-coding Implementation Optimizations

We now discuss some improvements to the baseline algorithm presented in the previous subsection. These include the representation of colorings through a combinatorial index system, careful memory management, and partitioning the input template to reduce work performed.

2.4.2.1 Combinatorial Indexing System

We represent a color set as a 32-bit integer. This representation considerably simplifies table accesses and stores for any arbitrary color set of arbitrary size. It also avoids having to explicitly define, manipulate, and pass arrays or lists of color set values. In order to ensure that each combination of colors is represented by a unique index value, these values are calculated based on a combinatorial number system [67]. For a subtemplate S_i of size h with k possible colors, the color set C would be composed of colors $c_1, c_2, \ldots, c_h$, each of possible (unique and increasing) values $0, 1, \ldots, k-1$, the corresponding index I would be $I = \binom{c_1}{1} + \binom{c_2}{2} + \cdots + \binom{c_h}{h}$.

In the innermost loops of the algorithm, we also look at all color sets C_a and C_b created by uniquely distributing the colors of C to the two children subtemplates of the partitioned S_i. By precomputing all possible index values for any given C, and any given sub-color set of size $1, \ldots, h-1$, we are able to replace explicit computation of these indexes with memory lookups. This considerably reduces the number of indexing operations on these innermost loops. It also allows these loops to exist as simple *for* loops incrementing through the index values, rather than the slower and more complex loops required with the handling of an explicit color set. The total storage requirements for the complete set of indexes is proportional to 2^k, and the representation only takes a few megabytes even for templates of size 12.

2.4.2.2 Memory Utilization Optimizations

A major consideration in the color-coding algorithm is the memory required for tabular storage of counts. This table grows proportional to $n\binom{k}{\frac{k}{2}}$ (n is the number of vertices in the graph, and k is the number of vertices in the template). For $k = 12$ and $n = 2,000,000$, this would mean that we require 32 GB of memory to determine a subgraph count using this algorithm. We have thus implemented a number of different memory-saving techniques to reduce the table size.

As previously mentioned, we initialize our table as a three-dimensional array. The first dimension is for each subtemplate generated through our initial template partitioning. We organize the order of the partitioning tree so that at any instance, the tables and counts for at most four subtemplates need to be active at once. Using the bottom-up dynamic programming approach for counting means that once the counts for a parent subtemplate are completed, the stored counts for the active and passive children can be deleted. We can also exploit symmetry in the template by analyzing possible rooted automorphisms that exist in the partitioned subtemplates. An obvious example can be seen in template U7-2 shown in Figure 2.1. We can reorganize the parent/child relationships in the partitioning tree so that only one of the isomorphic subtemplates needs to be analyzed, as the counts will be equivalent for both.

The second dimension in the table is for every vertex in the full graph. For our dynamic table, we only initialize storage for a given vertex v if that vertex has a value stored in it for any color set. This also allows a boolean check to be done when calculating new subtemplate counts for a given vertex. Since the counts for that vertex are based on the active child's count at v and the passive child's counts at $u \in N(v)$, we can avoid considerable computation and additional memory accesses if we see that v is uninitialized for the active child and/or u is uninitialized for the passive child. As we will discuss later, partitioning the graph in a certain way allows considerable algorithmic speedup by exploiting this further.

The third and innermost dimension of our table is for the counts for each color set value. As discussed, these values are set and read based on the combinatorial number system index for the specific coloring. By organizing the table in this way, accesses can be quickly done as table[*subtemplate*][*vertex*][*color index*]. This storage follows the loops of the algorithm, which can help reduce cache misses on the innermost loops.

We have also developed a hashing scheme that can be used in the place of a three-dimensional array for high-selectivity templates. The key values used are calculated for vertex v and color set C as follows, where v_{id} is the numeric identifier of the vertex, I is the color set's combinatorial index, and N_c is the total number of color set combinations for the current subtemplate: $key = v_{id} \cdot N_c + I$. Calculating the key in this way ensures unique values for all combinations of vertices and color sets. Additionally, if we initialize and resize the hash table to simply be a factor

of $n \cdot N_c$, where n is the number of vertices in G, we can use a very simple hash function of ($key \mod n$). This gives a relatively uniform distribution across all possible hash indexes based on the initial random coloring of G. This hashing scheme will generally save memory over a simpler array-based scheme when a template occurs with high regularity rooted at certain vertices within G, but with low regularity relative to the number of possible color sets.

2.4.2.3 Template Partitioning

We also explore various template partitioning strategies. When possible, we employ a one-at-a-time approach, where we partition a given subtemplate so that either the active or passive child is a single vertex. There are two reasons why we do this. The running time of the two innermost loops of the algorithm are dependent on $\binom{k}{h_i} \cdot \binom{h_i}{a_i}$, where k is the number of colors, h_i is the number of vertices in the subtemplate we are getting the count for, and a_i is the number of vertices in the active child of that subtemplate (note that $\binom{h_i}{a_i} = \binom{h_i}{p_i}$, where p_i is the number of vertices in the passive child). The running time of the algorithm grows as the sum over all $\binom{k}{h_i} \cdot \binom{h_i}{a_i}$, for every pair of h_i and a_i, at each step in the partitioning tree. A one-at-a-time approach can minimize this sum for larger templates (except when exploiting rooted symmetry), as the larger multiplicative factors tend to dominate with a more even partitioning.

However, we observe faster performance with a one-at-a-time partitioning approach over the symmetry-based template partitioning. This is due to the fact that by setting the active child as the single partitioned vertex at each step when possible, we can reduce the total number of color sets at each vertex v by a factor of $\frac{k-1}{k}$. The count at each v is dependent on the count for the active child with a given color set, and only one color set for a single vertex subtemplate exists that has a non-zero count: the coloring of v.

Also, note that the root selection can impact how the template can be partitioned using the one-at-a-time approach. Our strategy is to randomly select a leaf vertex as the initial root. After the first cut, we continue to greedily prune leaf vertices whenever possible. We have not yet explored other ways of determining the root. This might make for interesting future work.

2.4.3 Shared-Memory Parallelism

We support shared-memory parallelism in FASCIA using the OpenMP programming model and have two modes of multithreaded parallelism. The choice is left to the user and is dependent on graph and template size. For large graphs, we parallelize the loop that calculates counts for all vertices $v \in G$. Each thread is assigned a unique set of vertices, for which it calculates and stores the next level of counts. Because vertices are partitioned among threads, and given the tabular layout of the counts table, there is no concurrent writes to shared locations.

However, for small graphs and small templates, the ratio of available parallel work to the necessary serial computational portion is low, and multithreaded performance suffers. Therefore, for this instance, we perform multiple outer loop iterations concurrently, where each thread independently computes the full counts for a subset of the total number of iterations. Each thread necessarily has its own dynamic table. The counts are then collected and averaged after the specified number of iterations is completed. Due to the fact that each thread initializes its own table, the memory requirements increase linearly as a function of the number of threads. However, for smaller graphs where this *outer loop* parallelization works better, the vertex counts are small enough that this is unlikely an issue, even while running on a system with limited memory.

While both inner and outer loop parallelism offer speedups over serial code, the choice is dependent on graph and subgraph topology as well as the runtime system. A hybrid strategy that combines both levels of parallelism is additionally possible. A dynamic scheduler that determines the optimal parallel strategy for a given input would make for interesting future work, but our current version of FASCIA leaves the choice as an input parameter to be given by the user.

2.4.4 Distributed Memory Parallelism

2.4.4.1 Distributed Counting

There are several avenues for distributed-memory parallelization of color-coding subgraph counting. Just as we implemented an outer loop method in shared-memory, we can extend this to distributed memory. A chunk of iterations of the outer loop can be assigned to a task. The inner-loop shared-memory parallelization

can be complementarily performed. We refer to this hybrid parallelization strategy as *distributed counting*, and pseudocode is given in Algorithm 2.3.

Algorithm 2.3 Dynamic programming routine with distributed counting.

```
for it = 1 to Niter do in parallel                    ▷ MPI task-level parallelism
    Color G(V, E) with k colors
    Initialize 3D count table
    for all S_i in reverse order of partitioning do
        for all v ∈ V do in parallel                  ▷ thread-level parallelism
            Update count table for template S_i
            using child subtemplate counts
```

2.4.4.2 Partitioned Counting Algorithm

For modest-sized graphs (more than 2 million vertices) and large templates ($k > 10$), memory utilization quickly becomes problematic with distributed counting. We have therefore also implemented a distributed graph partitioning-based approach, where each task performs counts of a subset of all vertices, to reduce per-task memory requirements further. A description of the algorithm is given in Algorithm 2.4.

Algorithm 2.4 FASCIA Fully Partitioned Counting Approach.

```
Partition subgraph S using single edge cuts
for it = 1 to Niter do
    Color G(V, E) with k colors
    for all S_i in reverse order of partitioning do
        Init table[S_i][V_r][·] for V_r (vertex partition on task r)
        for all v ∈ V_r do                            ▷ Thread-level parallelism
            for all c ∈ C_i do
                Compute all table[S_i][v][c]
        ⟨N, I, B⟩ ← Compress(table[S_i][V_r][·])
        All-to-all exchange of ⟨N, I, B⟩
        Update table[S_i][V_{r_N}][·] based on information received
    Count_r ← Count_r + Σ_v^{V_r} Σ_c^{C_T} Table[T][v][c]
Count ← Reduce(Count_r)
Scale Count based on Niter and colorful embed probability
```

The graph is partitioned in a one-dimensional manner among the MPI tasks, with each task storing V_r vertices and their adjacencies. For every S_i, we only

initialize the current table for the task's specific subset of vertices V_r. We compute all the counts for the subset of vertices for the current subtemplate. We then compress the table in the Compressed Sparse Row (CSR) format (details in the next subsection), with N denoting the array of count values, I the color mapping indexes, and B containing the start offsets for each vertex. The compressed table is ordered according to the ordering of tasks that have $v \in V_r$ in their one-hop neighborhood (V_{r_N}), as only these vertices are required in calculating the counts for each S_i, and we want to reduce communication costs. We distribute the counts in an all-to-all fashion among all r nodes, so that each node now has the child counts required to compute counts for the new parent template.

For the final S_i, i.e., the original template T, each task computes the final count for the template for their subset of vertices. We simply keep a running sum of the counts for each task, for every iteration. After all iterations are completed, we perform a global reduction of the sum from all nodes, scale the value by the number of iterations and probability that the template is colorful, to get the final count estimate. Note that no additional approximations are introduced during this procedure, and so a count produced with say, 15 MPI tasks, will be the same as the count generated by the serial algorithm (assuming the random graph colorings are seeded with the same value).

2.4.4.3 Table Compression

Due to the large memory footprint of the dynamic programming-based arrays, the partitioned approach also incurs a substantial inter-node communication cost. We reduce the total volume of data transferred by using a compressed sparse row (CSR) representation for storing the non-zero counts. The CSR format is commonly used in numerical analysis for the storage of sparse matrices. Storage using this format consists of three arrays. One array stores all values held in the matrix in row-major ordering. This array would be organized as $[(\text{row}_1)(\text{row}_2)\cdots(\text{row}_n)]$, where (row_i) is a list of all nonzero values in that row. A second array of the same length as this first array is used to hold the column indexes at each of the nonzero values stored in the first array. The final array is of length n, or the number of rows, and it holds indexes to the start of the sequence of values for each row.

By considering a table for each discrete subtemplate S_i as a matrix of size $n \times c_i$, where n is the number of vertices in G and c_i is the number of possible color sets

for S_i, we can apply the CSR format to our table, in order to compress it for faster transfer across tasks running on our cluster. The first array N stores all non-zero counts for all vertices and color mappings. The second array I is the color mapping indexes for each count value, as computed using the combinatorial number system approach. The final array B denotes the indexes for the start of count values for each vertex.

Due to the large graph and template sizes considered in our study, the overall per-vertex and per-color set count magnitudes can be quite massive in scale. This requires the N array to be of type 64-bit `double` to avoid overflow. Similarly, because the N array length can exceed the limits provided by 32-bit `unsigned int` for array indexing, the B array is of type 64-bit `unsigned long`. We use a 16-bit integer to store the color set index array B, which will allow unique indexes up to templates of 18 vertices in size.

Because the lookup for any specific (x, y) index can be slow using this format and the color-coding approach requires a significant number of such lookups, we ideally want to re-expand the compressed values. However, in order to further reduce memory footprint, we only re-expand for each vertex when they are needed to compute the count of the new parent subtemplate. The overhead for this uncompression step is minimal in practice.

2.5 Results and Analysis

2.5.1 Experimental Setup

Experiments were performed on various parallel platforms and interactive systems, including Stampede at the Texas Advanced Computing Center, and the Hammer and Cyberstar systems at Penn State University. For experiments where execution times are reported, we used the Compton system at Sandia National Laboratories. Each Compton node has 2 Intel Xeon E5-2670 (Sandy Bridge) processors with 64GB DDR3 memory running RHEL 6.1. We use up to 16 nodes for our experiments. The MPI libraries used were from Intel (version 4.1) and we used OpenMP for shared-memory parallelism. Code was compiled with `icc` using the `-O3` optimization flag, and `KMP_AFFINITY` was used to control thread to core pinning.

We evaluate performance of our implementations on a collection of several

Network	n	m	d_{avg}	d_{max}	$\widetilde{D}$	Source
Enron Email	34 K	180 K	11	1.4 K	9	[68, 69]
PA Roads	1.1 M	1.5 M	2.8	9	430	[69, 70]
Portland	1.6 M	31 M	39	275	16	[71]
Orkut	3.1 M	117 M	76	33 K	9	[69, 72]
Twitter	44 M	2.0 B	37	750 K	36	[73]
sk-2005	44 M	1.6 B	73	15 M	308	[74, 75]
H. pylori	710	1.4 K	4.0	54	10	[76]
S. cerevisae	5.1 K	22 K	8.7	290	11	[76]
H. sapiens	9.1 K	41 K	9.0	250	10	[77]

Table 2.1. Network sizes and average and maximum degrees and approximate diameter for all networks used in our analysis.

large-scale low diameter graphs, listed in Table 2.1. Orkut and Twitter (follower network) are crawls of online social networks obtained from the SNAP Database and the Max Planck Institute for Software Systems [69,72,73]. Also from the SNAP database is the Pennsylvania Road network [70] and the Enron Email Corpus [68]. sk-2005 is a crawl of the Slovakian (.sk) domain performed in 2005 using UbiCrawler and downloaded from the University of Florida Sparse Matrix Collection [74, 75]. Portland is a large synthetic social contact network modeled after the city of Portland, from the Virginia Tech Network Dynamics and Simulations Science Laboratory (NDSSL) [71]. The H. pylori (intestinal bacteria) and S. cerevisae (yeast) networks where obtained from the Database of Interacting Proteins [76], and the H. sapiens (human) network is from Radivojac et al. [77].

All the networks considered are undirected. The originally-directed Twitter and sk-2005 graphs were preprocessed to ignore edge directivity, remove multiple edges and self loops, and extract only the largest connected component. Table 2.1 lists the properties of the graphs after this preprocessing.

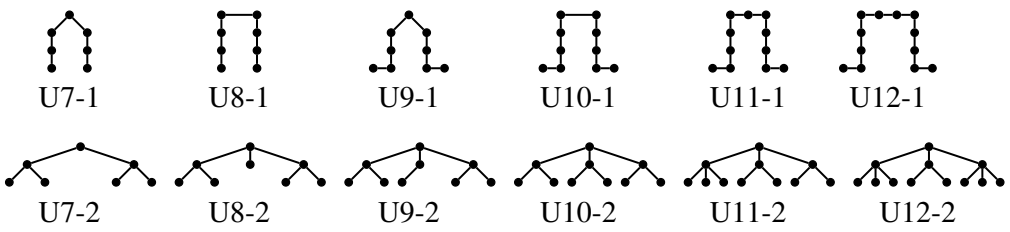

Figure 2.1. Select templates used in performance analysis.

While analyzing execution times and scaling on the larger networks, we considered two different templates with 5, 7, 10, and 12 vertices. For each size, one template is a simple path and the other one is a more complex structure. The path-based templates are labeled as U5-1, U7-1, U10-1, and U12-1. These templates and their labels are shown in Figure 2.1. Other templates are used in the analysis that are not listed follow the same naming convention, with UX-1 implying a simple path and UX-2 being a more complex tree. For motif finding, we looked at *all possible treelets* of size 7, 10, and 12. $k = 7$, 10, and 12 would imply 11, 106, and 551 possible tree topologies, respectively. The treelets for $k = 7$ are given in Figure 2.2.

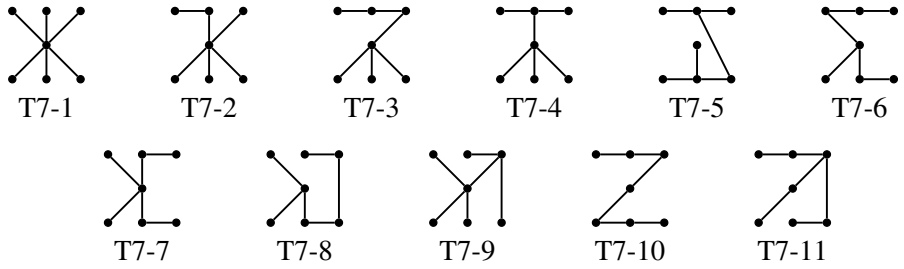

Figure 2.2. All possible 7 vertex undirected tree-structured templates.

2.5.2 Single-node performance

To assess single node FASCIA performance, we will examine running times of a single iteration on moderate-sized networks across varying template sizes, running times for several iterations across all motifs on smaller PPI networks, parallel scaling, memory utilization with the various strategies, as well as an analysis of approximation error.

2.5.2.1 Running times vs. template size

Figure 2.3 gives the absolute single-node running times for all templates listed in Figure 2.1 on the Portland and Orkut networks. These results are from running inner loop-parallel version on 16 cores. We observe minimal to no performance improvement when using hyperthreading, so most tests were performed with only a single thread per core despite two hardware thread contexts.

As can be observed on Figure 2.3, the single-iteration time for smaller templates is extremely low, making it feasible to obtain realtime count estimates for 7 vertex

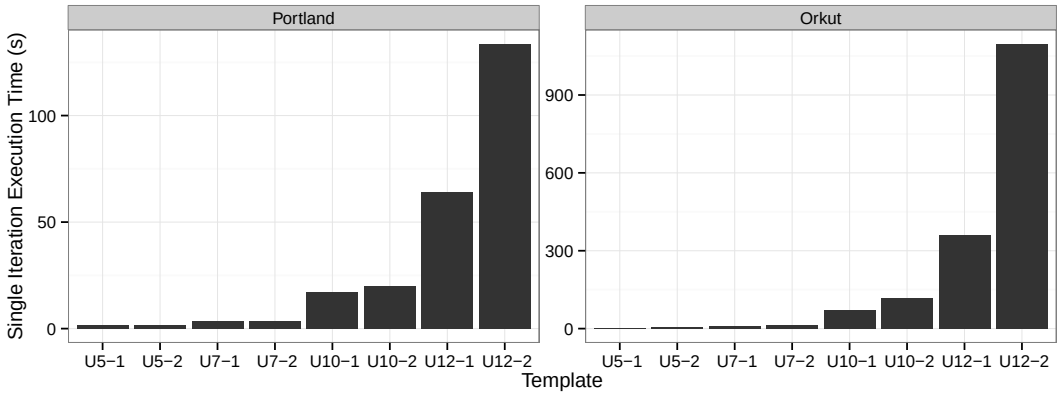

Figure 2.3. FASCIA running times on templates of size 5, 7, 10, and 12 vertices, on the Portland and Orkut networks, for a single iteration, with inner loop parallelism.

templates on both networks. Even for the largest template, the total running time was still less than 20 minutes on both networks. The U12-2 template took the longest time as expected. This template was explicitly designed to stress subtemplate partitioning and therefore gives a practical upper bound for our running times across all template of size 12 and smaller. Another observation was that the running time was fairly independent of the template structure, particularly for the smaller templates. Even for the larger 12-vertex templates, there is just a 3× variation in running time.

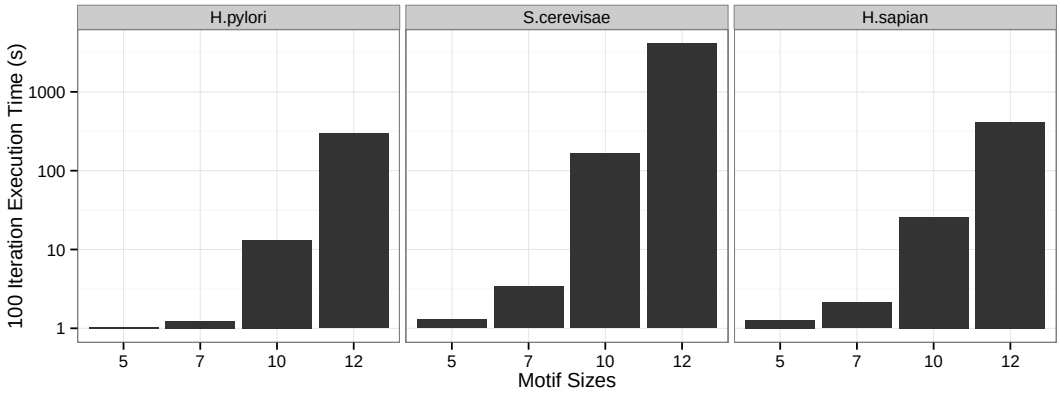

Figure 2.4. FASCIA running times on templates of size 5, 7, 10, and 12 on the H. pylori, S. cerevisae, and H. sapiens PPI networks for 100 iterations with outer loop parallelism.

Figure 2.4 gives the running times for 100 iterations across all possible 5, 7, 10, and 12-vertex templates on the H. pylori, S. cerevisae, and H. sapiens protein

interaction networks. For these tests, we use outer loop parallelism. It demonstrates superior running times on smaller networks with larger iteration counts, as there is less parallel overhead on a per-iteration basis. Note the log scale in Figure 2.4. Both the running time of FASCIA and the total number of possible templates increase exponentially with increasing network size. This demonstrates the importance of implementing fast serial algorithms for subgraph counting when analyzing motifs of larger sizes.

2.5.2.2 Parallel Scaling

We now will observe how our approaches scale when increasing the number of processing cores. Figure 2.5 gives the parallel speedups for 1 to 16 cores on the Portland network with inner-loop parallelism, as well as the H. pylori network with both outer loop and inner loop parallelism. As was mentioned previously, we observe better speedups with outer loop parallelism on the smaller networks, as the per-iteration parallelization overhead is reduced. However, on larger networks, we still observe very good speedups and near-linear scaling with the inner-loop parallelism, since the computational requirements overshadow the parallel overheads for these instance. Overall, our implementation scales quite well, demonstrating about 15× and 14× speedups on the larger Portland and smaller H. pylori networks, respectively.

2.5.2.3 Reduction in Memory Use

Figure 2.6 demonstrates impact of the memory-reducing optimizations over the baseline naïve approach. The peak memory footprint is given in both the networks for various template sizes. For the Portland network, we see a 20% savings over baseline with the improved dynamic programming table representation. Further, if we consider the case of per-vertex labels in the graph (all vertices randomly initialized with one of 8 labels), the memory requirements drop considerably, due to the much higher selectivity that the label restriction imposes.

In Figure 2.6, we also see how a hash table representation can improve memory usage dramatically over using the three-dimensional table, on certain networks. The PA road network is quite regular and nearly planar, and so it is expected that for any given template, a vertex will have an embedding rooted at it. This results

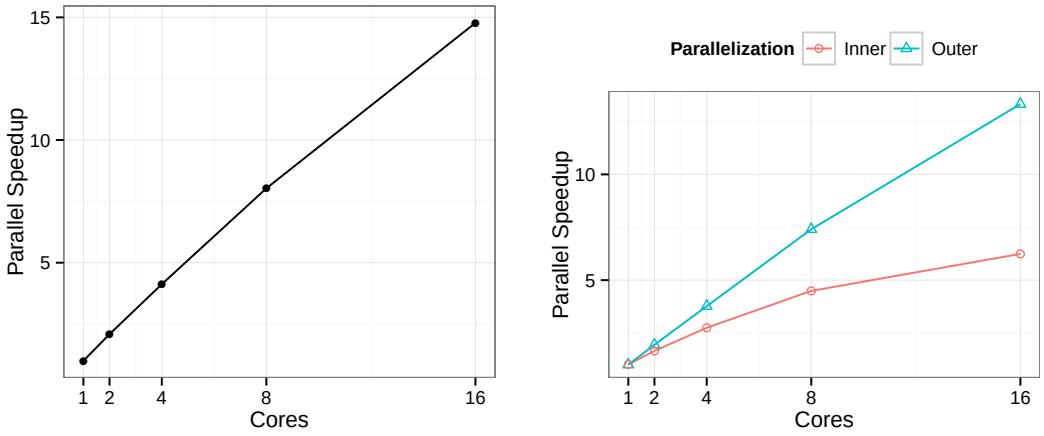

Figure 2.5. Parallel scaling from 1 to 16 cores of the U12-2 template on the Portland network for a single iteration with inner-loop parallelism (left) and parallel scaling for 100 iterations of all 10 vertex templates on the H. pylori network with both inner and outer scaling (right).

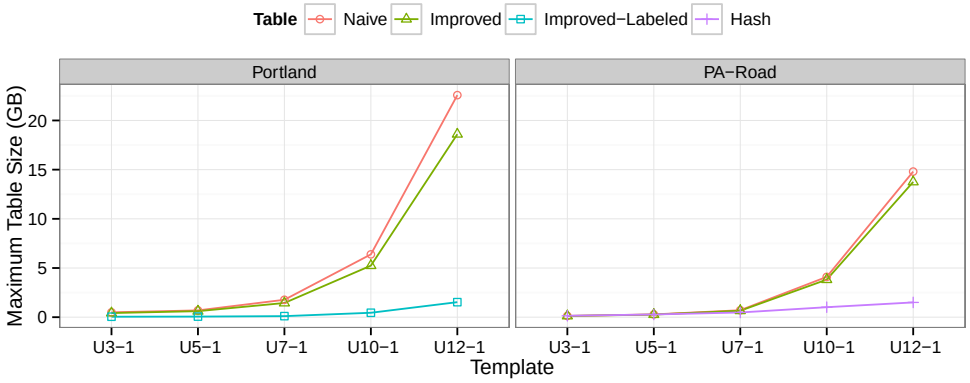

Figure 2.6. Peak memory use reduction on the unlabeled and labeled Portland network with the improved table (left), and memory use reduction that results from using an improved table and hash table on the PA Road network (right).

in the table having to initialize storage for every vertex. However, since the network is sparse, it is unlikely that that every color set will have an embedding, which is why we see such a significant memory use reduction. On a denser network like Portland, it is unlikely there would be much improvement with the hash table, due to the relatively higher number of embeddings relative to the number of vertices in the network.

2.5.2.4 Error Analysis

We next analyze the error in the subgraph counts produced on small and moderate-sized networks. We report the magnitude of relative error (difference in counts divided by true count) in the two figures. In Figure 2.7 (left), we observe the error produced when counting 3 and 5 vertex chain templates on the Enron network. We note that the error falls under 1% after three iterations for both templates. We observe higher error with the larger template. The extremely small number of iterations necessary for low error, on modest-sized networks, mirrors the results seen in prior work [9, 62].

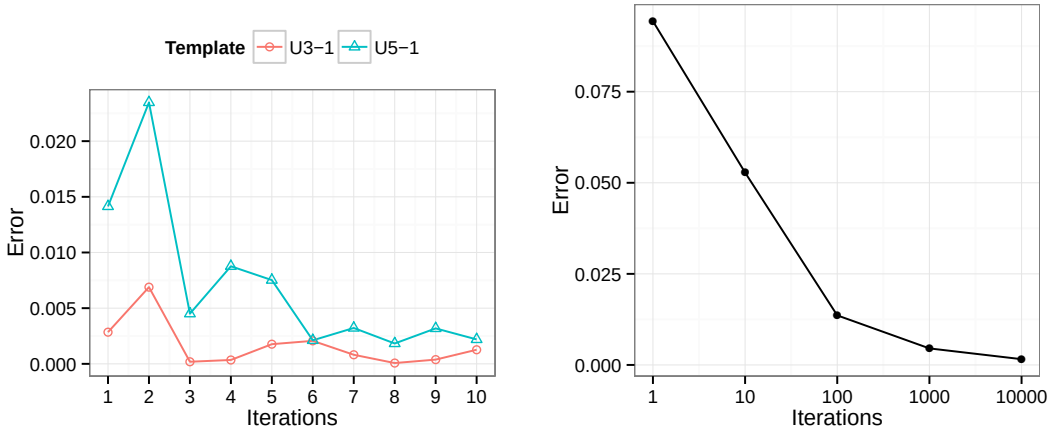

Figure 2.7. Error obtained with the 3 and 5 vertex path templates on the Enron network after a small number of iterations (left) and the average error over all possible 7 vertex templates on the H. pylori network after 1 to 10 K iterations (right).

For the smaller H. pylori network, we note that it takes about 100 iterations to reach about 1% average error across all 7 vertex templates. This network is very small and sparse, and so for large templates, a relatively larger number of iterations are required. Generally, we observe per-iteration error increasing with template size, but decreasing with network size. Also, the greater the number of template embeddings that exist within the graph, the lower the error. This is due to the fact that the random coloring of the graph and the subsequent count scaling has a relatively-smaller impact on the final count estimate. We observe low inter-iteration variation between produced counts on large scale networks. A dynamic stopping criteria based on the variance of produced per-iteration counts is left for future work.

Note that Alon et al. [49] proved that to ensure a computed count is within $\mathcal{C}(1 \pm \epsilon)$ with probability $(1 - 2\delta)$, where $\mathcal{C}$ is the true count, at most $\frac{e^k \log 1/\delta}{\epsilon^2}$ iterations of the dynamic programming scheme are required. However, this upper bound is very loose in practice, as it ignores network size and topology. For example, we require only 100 iterations to compute counts with an error less than 1% for a 7 vertex template on H. pylori ($\epsilon = 0.01$).

2.5.3 Multi-node performance

For the distributed-memory FASCIA implementation, we again analyze performance with regards to running times and parallel scaling. We also demonstrate the inter-node communication reduction with CSR table compression, which also translates to memory savings during the subsequent dynamic programming step. For these experiments, we employ a label propagation-based graph partitioning [47] with random intra-partition vertex reordering to balance overall computation and communication costs.

2.5.3.1 Running times vs. template size

Figure 2.8 gives the running times on the large sk-2005 and Twitter networks for templates between 5 and 10 vertices on 16 nodes of Compton. Due to the large scale of the networks and restricted parallelism on our modest cluster, the memory requirements for the 12-vertex template was too high for us to run to completion. Using more compute compute nodes would reduce per-node memory requirements and allow us to scale to larger network and template sizes. This is left for future work.

From Figure 2.8, we observe that count times for 5-vertex templates complete in seconds, and the larger templates in minutes on these networks. Note that the running times of the binary tree-structured templates are lower than that of the path-like templates in these instances. This is because the computational requirements for the tree templates are higher than the path templates, but they have lower communication and memory costs. This leads to lower performance in shared memory, but faster performance in distributed memory.

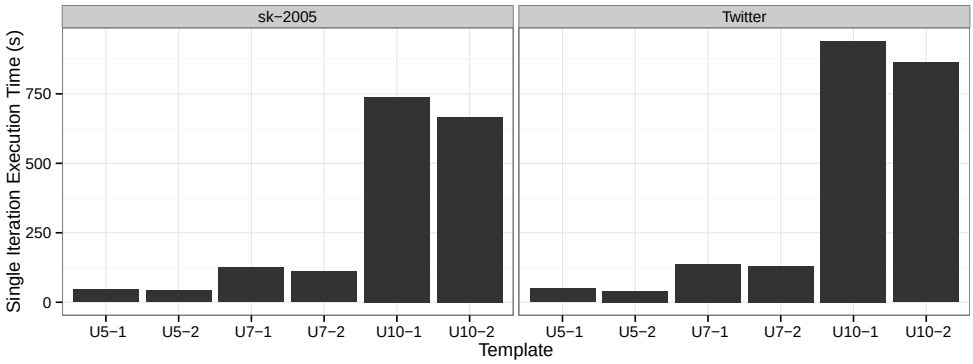

Figure 2.8. Running times on 16 nodes of Compton of tested 5, 7, and 10 vertex templates on the sk-2005 and Twitter networks for a single iteration with partitioned counting and inner loop parallelism.

2.5.3.2 Parallel Scaling

Figure 2.9 gives the scaling of the single-iteration running times on the Orkut network from 1 to 16 nodes, and the scaling of the sk-2005 from 2 to 16 nodes. The Orkut network calculates the counts for the U12-2 templates and sk-2005 network uses the U7-2 network. We show scaling for the total execution time, the portion of time spent in the counting computation phase, and the total time spent in the communication phase.

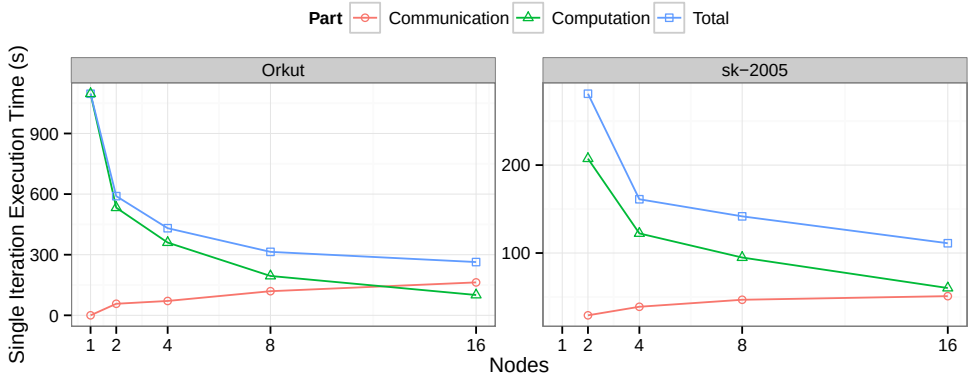

Figure 2.9. Parallel scaling from 1 to 16 nodes of the U12-2 template on Orkut network and the U7-2 template on the sk-2005 network for a single iteration with partitioned counting and inner loop parallelism.

We observe about a 4× overall speedup on the Orkut network, with about 10×

speedup of just computation. Communication increases about 3× from 2 to 16 nodes. We observe higher communication costs with this approach compared to implementation [42], due to a more complex communication phase. However, this higher complexity reduces memory utilization and is necessary in order to calculate the counts on networks as large as sk-2005.

On the sk-2005 network, we observe about 2.5× speedup from 2 to 16 nodes. We observe that there is relatively good scaling with computational time, but at the cost of increasing communication time. Due to the larger-scale and higher overall computation costs, even with a smaller template, the communication costs are slightly less than the computation costs at 16 nodes. With greater parallelism, it is likely that communication costs will dominate. In future work, we will attempt to optimize the All-to-all exchange in order to minimize these costs for larger networks, and achieve better parallel scaling.

2.5.4 Comparisons to Recent Work

SAHAD [9] and PARSE [62] by Zhao et al. both utilize the color-coding approach for determining approximate subgraph counts in distributed environments. PARSE is an MPI-based approach and SAHAD is a newer and more scalable version that uses Hadoop. The parallelization strategies and software environment used in their performance studies are very different from the settings used in our study. Hence it is difficult to perform a head-to-head comparison. To get a sense of relative speedup with our approach, consider the following selection of performance results: The PARSE paper reports an execution time of about an hour on 400 cores of a cluster for a 2 million vertex, 50 million edge network and a 6-vertex chain template. For the same network, a single color-coding iteration with SAHAD for a 10-vertex tree template is reported to take 25 minutes on 42 nodes (1344 cores). With distributed Fascia on 15 nodes (240 cores), we can count occurrences of a 12-vertex chain and 12-vertex tree on a 3 million vertex, 117 million edge network in about 3 and 4 minutes, respectively.

2.6 Conclusions

This chapter presented several new optimizations for implementations of color-coding-based graph algorithms. Using these optimizations, we created shared- and distributed-memory parallelized Fascia, a fast and memory-efficient tool for subgraph counting on both small and large networks. However, subgraph counting by itself offers little insight into the structural characteristics of a given network. In the next chapter, we will introduce several use-cases of subgraph counting for network analysis and show how the counts output by Fascia can be utilized for such analyses.

Chapter 3
Subgraph-based Graph Analysis

3.1 Introduction

In the previous chapter, we introduced FASCIA [43], a fast parallel implementation of the color-coding algorithm for approximate treelet counting. As subgraph counts in isolation give little insight into network topology, a number of analytics introduced over the past decade use subgraph counts as a means for identifying latent structural patterns within networks, for ascertaining possible variations between neighborhoods of individual vertices within a single network, and for comparative analysis of networks. We give an overview of some of these subgraph-based analytics in the next section. Our primary contribution of this chapter is an extensive comparative analysis of real-world networks using subgraph count-based graph analytics.

3.2 Background

3.2.1 Motif Finding

Network motifs are defined as subgraphs that occur more frequently in a network than would be expected by random chance. There has been considerable study of network motifs in bioinformatics [4, 52, 53, 55, 56, 58, 78], usually to discern structurally-significant characteristics in protein-protein interaction and related biological networks. Alon et al. [52] implemented a color-coding based approach for the purpose of determining biological network motifs. Recent work has also focused on applying subgraph counting and network motif finding methods for

social, informational, and other networks [9,62,79]. We perform a similar evaluation in this report.

Motif finding is typically carried out by finding complete subgraph counts for all possible templates of up to a certain size. The relative frequency of each subgraph is determined by scaling the counts by either the total count or the average count. This process is then repeated for perturbations of the network or for synthetic graphs with similar network parameters. From this, subgraphs with exceptionally high or low relative counts can be determined. Prior knowledge about the network can then be used to determine whether the subgraph plays a crucial role related to a functional characteristic. We will demonstrate the resilience of treelet counts for motif finding with noisy or incomplete networks later in this report.

3.2.2 Graphlets

Graphlets are formally defined as small undirected subgraphs between two and five vertices in size. Prior work by Pržulj et al. [1, 51, 59, 60] extensively studies graphlets in the context of biological networks. Pržulj et al. also identified all possible discrete *orbits* within each graphlet. Orbits in this context refer to distinct automorphic vertices in the subgraph; i.e., it can be useful to explicitly differentiate between a vertex in the center of a star versus a vertex on one of the leaves.

Graphlets are also considered to only be *induced* subgraph occurrences. Induced occurrences imply that for every subgraph embedding of a template in a network, edges can exist between vertices in the subgraph if and only if they exist in the template. Most prior work focuses on induced occurrences, as they are computationally less expensive to count. However, Alon et al. [52] argue that non-induced subgraph counts can provide a more accurate analysis of noisy real-world networks. The color-coding technique can only be used to count non-induced occurrences, and verifying its applicability to the comparative metrics is one of the primary focuses of this work.

3.2.2.1 Graphlet Frequency Distance

The graphlet frequency distance (GFD) was proposed by Pržulj et. al [51] as a global comparative measure based on the local structural characteristics of different networks. To calculate the GFD, counts of all $i = 1, 2, \ldots, 29$ graphlets in network

G are determined as $C_i(G)$. Each of the 29 counts are then log-scaled by the total count of all graphlets, with the distance value between networks G and H being the sum of the absolute differences of all scaled counts.

$$S_i(G) = -\log(\frac{C_i(G)}{\sum_{i=1}^{29} C_i(G)}), D(G,H) = \sum_{i=1}^{29} |S_i(G) - S_i(H)|$$

To avoid possible confusion created by using the term *graphlet* outside of its standard context (subgraphs of 2 to 5 vertices), as well as the generic term subgraph, this work will use the term *treelet* in their place where appropriate (for instance, in this work, we will compute treelet frequency distances).

3.2.2.2 Graphlet Degree Distribution

The graphlet degree distribution (GDD) can similarly be used as a global comparative measure between networks based on local structural characteristics [1]. To calculate the GDD of a network, we determine the graphlet counts at every single vertex in the network for all different graphlet orbits. The graphlet distribution for a given orbit is the number of vertices in the network that have a certain number of embeddings, or degree, with the graphlet orbit; e.g. there are 450 vertices with a single embedding (a graphlet degree of one), 230 with two embeddings (a graphlet degree of two), 114 with three embeddings, and so forth. The agreement value for a given orbit between two networks is determined as the euclidean distance between all degree counts normalized with respect to the total area under a curve scaled by the degree number:

$$S_G^j(k) = \frac{d_G^j(k)}{k}, N_G^j(k) = \frac{S_G^j(k)}{\sum_{k=1}^{\infty} S_G^j(k)}$$

$$A^j(G,H) = 1 - \frac{1}{\sqrt{2}}(\sum_{k=1}^{\infty}[N_G^j(k) - N_H^j(k)]^2)^{\frac{1}{2}}$$

Where $d_G^j(k)$ is the number of nodes with degree k of orbit j in graph G, $N_G^j(k)$ is the normalized distribution, and $A^j(G,H)$ is the agreement for orbit j between graphs G and H. The total agreement is either the arithmetic or geometric sum of agreements for all orbits. We consider only the arithmetic sum, as instances of zero

agreement between orbits are observed to occur in treelets larger than 5 vertices with minor frequency.

3.2.2.3 Graphlet Degree Signature Similarity

The graphlet degree signature can give an comparative similarity value between two given nodes in a single network. A vector is created for all nodes in the network containing the counts of embeddings for all graphlet orbits containing that node. It is described as capturing the local topology and interconnectedness of the node in the context of its local neighborhood [60]. The similarity value between two vertices, u and v, for orbit i, with counts u_i and v_i, respectively, is calculated as follows:

$$S_i(u,v) = 1 - w_i \times \frac{|\log(u_i+1) - \log(v_i+1)|}{\log(\max\{u_i, v_i\} + 2}$$

In this equation, w_i is a certain weighting given to that specific orbit. These values are dependent on the number of isomorphisms of smaller graphlets orbits that exist at that orbit in its respective graphlet. The total similarity value is the sum of similarity values divided by the sum of the weightings for all orbits. Although this work does not explicitly utilize the graphlet degree signature similarity, future work will investigate its applicability towards network alignment, clustering, and community detection.

3.2.3 Clustering

Bordino et al. [61] demonstrate that one can use the relative frequency of subgraphs within networks to distinguish and cluster different networks. Using the relative frequencies of undirected subgraphs up to four vertices and other topological properties such as in-degree, out-degree, and PageRank as representative features for a network, they show up to 75% clustering accuracy for networks chosen from seven distinct categories. Using directed edges and 284 features in total, they achieved just over 90% clustering accuracy. Recent work by Rahman et al. [57] implements an approximate graphlet counting algorithm and uses graphlet counts as a vector to cluster various network types. We perform a similar clustering for a larger group of networks using treelet counts.

3.3 Experimental Setup

We performed experiments on various parallel platforms and interactive systems, including Gordon at the San Diego Supercomputer Center, Stampede at the Texas Advanced Computing Center, and the CyberSTAR and Hammer systems at Penn State University. For experiments where execution times are reported, we used the Compton system at Sandia National Laboratories. Code was compiled with the Intel C compiler `icc` using `-openmp` and `-O3` flags.

3.3.1 Networks Analyzed

We analyzed networks from eleven different categories, obtained from many different sources [27,69,80] (see Table 3.1). These include collaboration networks from Arvix and the DBLP Computer Science Bibliography [70,72,81], communication networks of emails and Facebook wall posts [68,81,82], four Erdős-Rényi $G(n,p)$ random graphs, snapshots of the peer-to-peer Gnutella network at various times [81,83], four biological protein-protein interaction (PPI) networks [76], five road networks [27,70], four random scale-free Barabăsi-Albert networks [84], six social networks of online relationships of various types [80,85,86], four random small-world graphs [87], and four web crawls of various universities and Google [70]. In total, 50 different networks were considered.

Network Type	Count	n ($\times 10^3$)		m ($\times 10^3$)	
		min	max	min	max
Collaboration	6	26	425	14	1050
Communication	4	30	63	87	855
$G(n,p)$	4	10	100	100	1000
Peer-to-peer	9	6	63	9.7	77
Bio PPI	4	0.7	22	1.3	22
Road	5	440	1970	530	2800
Scale-free	4	10	100	100	1000
Social	6	60	150	214	5400
Small-world	4	10	100	100	1000
Web Crawl	4	280	875	761	3900

Table 3.1. Networks analyzed in this study: categories, counts, and sizes in terms of the maximum and minimum numbers of vertices (n) and edges (m) for each network category.

All graphs considered are undirected with multiple edges and self loops removed. This mainly only affected the structure of the communication and social networks. The color-coding method can be applied to directed graphs as well. However, the current implementation is unable to do so and this extension is left for future work.

3.3.2 Templates Analyzed

All tree-structured templates between three and nine nodes were considered in our analysis. There is 1 template with 3 nodes, 2 templates with 4 nodes, 3 templates with 5 nodes, 6 templates with 6 nodes, 11 templates with 7 nodes, 23 templates with 8 nodes, and 47 templates with 9 nodes. The templates were created by parsing data output by an online graph generator [88]. Additionally, since treelet degrees were considered for templates of three to seven vertices, it was necessary to copy and modify each of these template files to set a varying root node for the different orbits. In total, 92 subgraphs were considered when calculating motifs and the treelet frequency distances and 83 discrete orbits over 23 different treelets were considered when calculating the treelet degree distribution agreements.

3.4 Results of Network Analysis

In this section, we use color-coding treelet counting in several network analysis methods: motif finding, network type clustering, as well as relative treelet frequencies and treelet degree distributions. The latter two use modifications to existing techniques that commonly use the smaller graphlets, with the changes to calculation methodologies noted where appropriate. We will also study robustness of using treelet counts for analyzing noisy or incomplete networks by perturbing a subset of vertices/edges.

3.4.1 Motif Finding

In this illustrative example for motif finding using results from previous work [10], we analyze five different networks using all eleven different seven node templates. These networks are the Enron email network, the social contact network of Portland, the Slashdot social network, a road network of Pennsylvania, and a $G(n, p)$ random graph created with the same number of nodes and edges as the Enron network. As

our previous work has demonstrated, even on relatively small networks, motifs can become apparent after a very small number of iterations. However, in this example, we run 1000 iterations to minimize any error. The resultant relative counts are given in Figure 3.1.

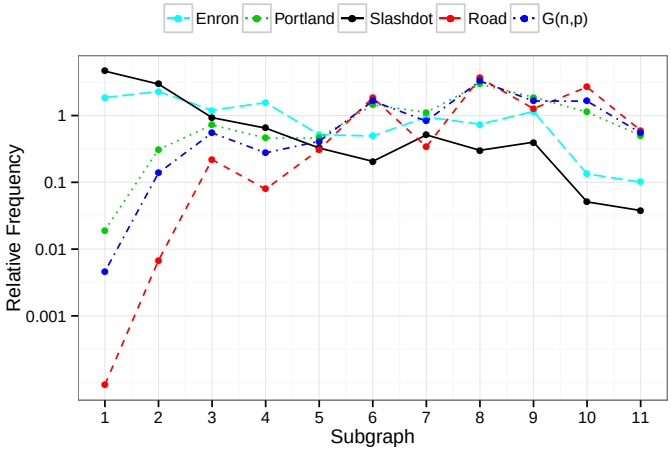

Figure 3.1. Relative frequencies of all seven vertex treelets on five different networks.

Template 1, a star with six edges, shows the largest spread in relative frequencies between all of the networks. It is clear that this template would be considered a very strong anti-motif for the road network. Having knowledge of what the road network is representing (streets are edges and nodes are intersections), we can deduce that these findings make logical sense, seeing as how there are very few six-way intersections commonly encountered.

It is also apparent that the $G(n, p)$ graph, while having the same number of nodes, edges, and average degree of the Enron graph, contains a vastly different local topology. This would once again corroborate existing information, in that it is known that $G(n, p)$ graphs very rarely correspond to real-world networks.

3.4.2 Relative Treelet Frequency Distances

The relative treelet frequency distances were calculated for all networks and groups using all treelets between four and nine vertices in size. We ran each count for 1000 iterations to minimize error in the estimates. The methodology used to calculate these distances is slightly different from the approach described in the background Section. Instead of taking the logarithm of the total count over all subgraphs, we scale values by the total counts for the specific treelet size. This is

done because, even using a log scale, the differences in count magnitudes between four and nine-node treelets is far too large to scale them by the same denominator.

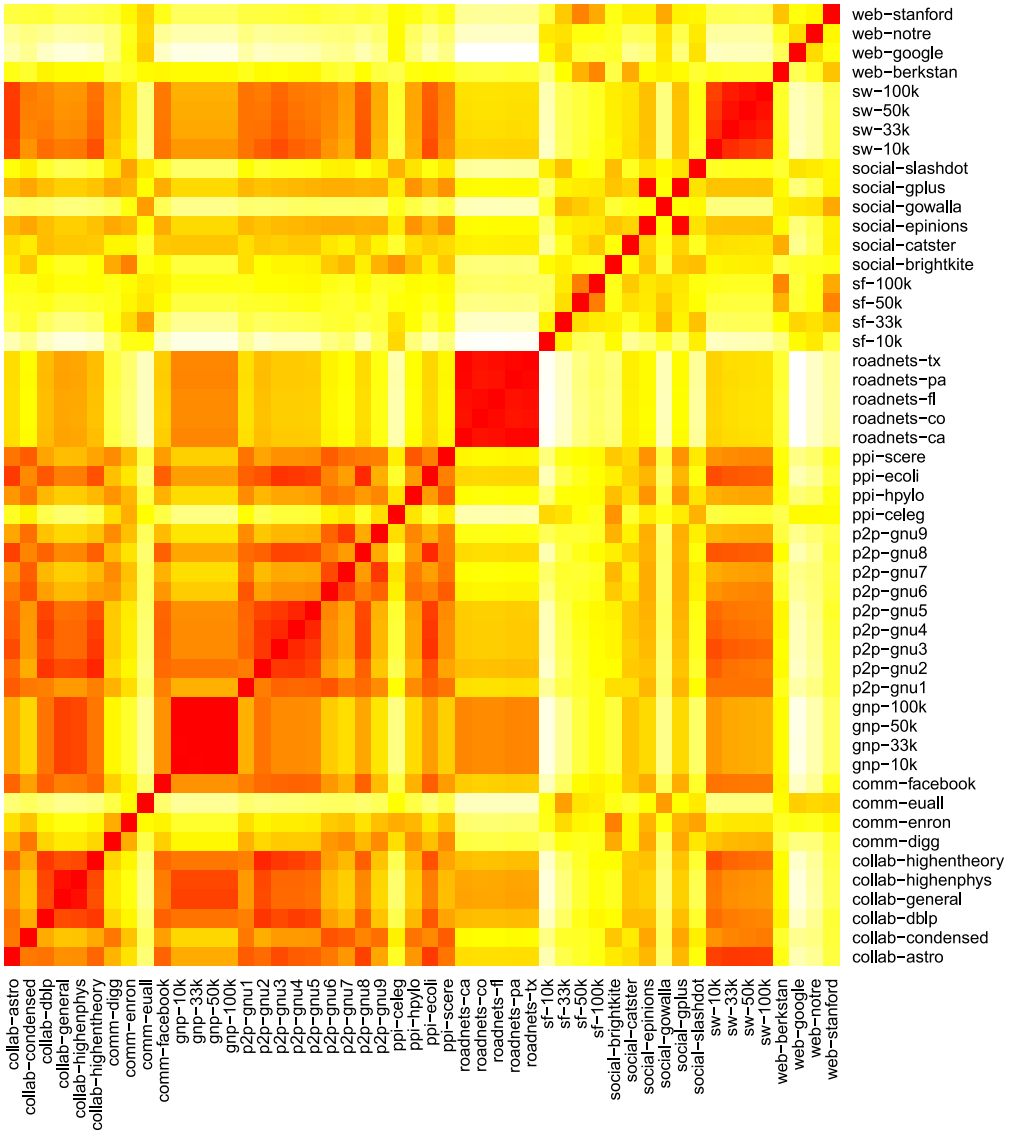

Figure 3.2. Treelet frequency distances between all tested networks. Darker implies a lower distance or higher similarity.

Figure 3.2 demonstrates the results of these calculations on a heatmap. For visualization purposes, the final distance values are log-scaled. Red represents low disagreement while orange, yellow, and finally white show increasingly higher disagreements. Each row-column coordinate represents the distance calculated between the networks (names on the right hand side and bottom). Networks are

ordered by group.

A number of observations are possible by observing Figure 3.2. Several groups show very minimal intra-group disagreement, including the road networks, the small world and $G(n, p)$ random graphs, as well as the collaboration and peer-to-peer networks in a lesser extent. The protein-protein interaction networks show agreement among the unicellular organisms (*E. coli*, *H. pylori*, *S. cerevisiae*), but low agreement with the multicellular organism (*C. elegans*).

The peer-to-peer results are interesting, in that there appears to be two distinct subgroups with high intra-group agreement. Each distinct network is simply a snapshot of part of the same larger network at varying points in time. This might highlight the highly dynamic and fluid nature of peer-to-peer networks, as connections are being constantly made and broken as new content is released and shared. Or it might just be an artifact due to noise and the relatively small sample of the network that was taken at each date.

There is also some agreement between the random small-world networks with the peer-to-peer and collaboration networks, demonstrating a correspondence to the small-world phenomenon that is known to exist in these networks. However, there is no such correlation with the social network crawls, which is surprising and likely suggests that there are other network measures necessary to take into account when determining network similarity beyond subgraph frequency, or that the graph generator parameters or algorithm used to create the small-world graphs need further tuning.

3.4.3 Treelet Degree Distribution Agreements

The treelet degree distribution agreement was calculated on all of the considered networks, for all 83 distinct treelet orbits, on all three to seven vertex tree-structured subgraphs. Again, we ran these counts each for 1000 iterations. These values were calculated using the same methodology as previously described for graphlets without any modification. The heatmap of results for all networks is presented in Figure 3.3 (bottom). Once again, red indicates a high agreement (low disagreement) and white indicates a low agreement.

One of the more interesting observations from this analysis deals with the scale-free networks. The absolute per-vertex counts for these networks were consistently

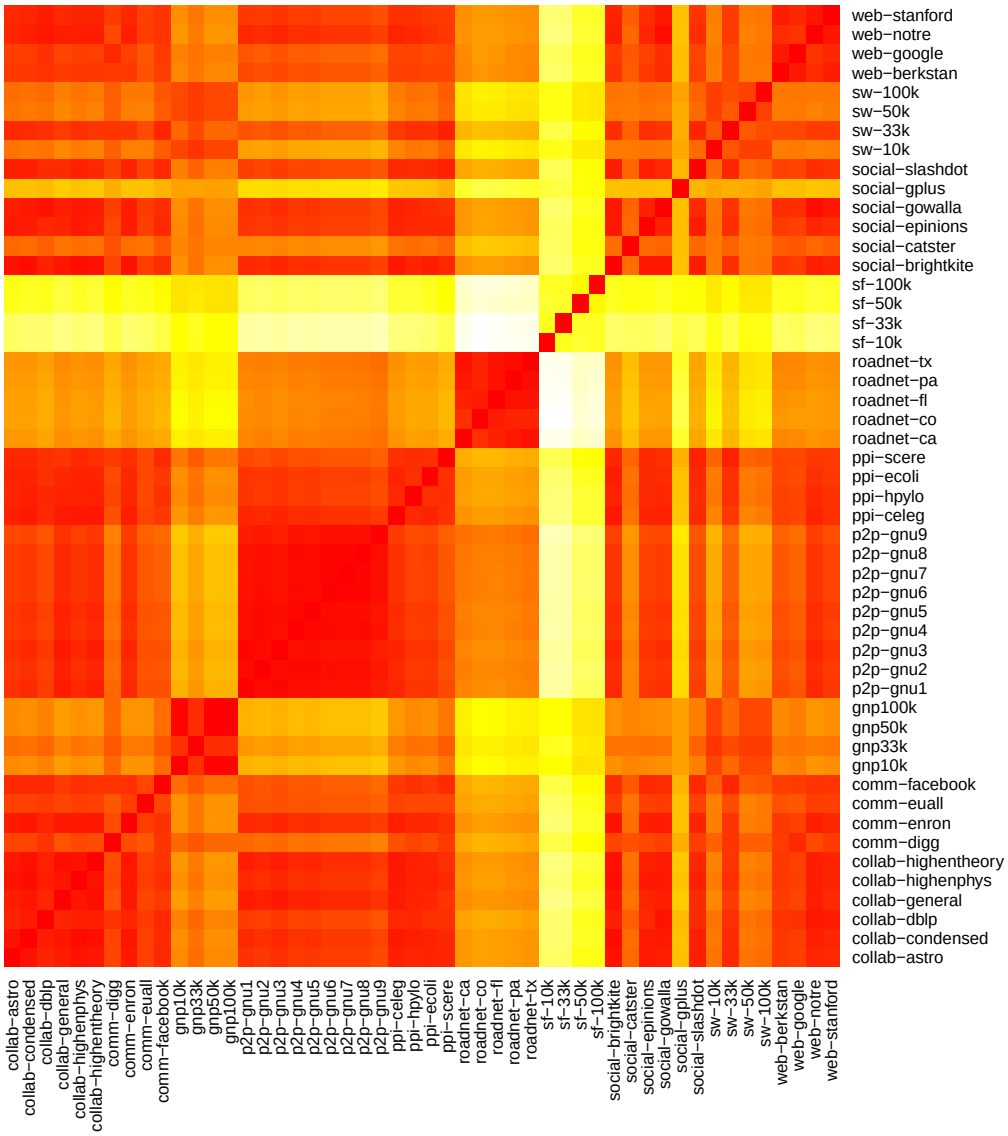

Figure 3.3. Treelet degree distribution agreements between all tested networks. Darker implies a higher agreement.

large, resulting in a very spread out distribution for each treelet orbit, and almost no agreement between any of the other networks. It is likely due to the existence of hubs with a massive degree and small diameter neighborhood that caused this to occur, as the number of treelet embeddings increase combinatorially with vertex degree. Since most vertices are within close range to a hub, their treelet embedding counts are affected as well when the treelet is large enough.

The Google+ social network experienced a similar phenomenon, as did the Digg

communication network to a lesser extent. Their overall distributions are extremely different from that of other networks in their group. It should be noted that the $G(n, p)$ 33K network is different from the other $G(n, p)$, because it was modeled with the same number of nodes and edges as the Enron network, and thus had a much lower average degree than the other random graphs in that group. This difference isn't noticed with treelet frequency distances, however, indicating that treelet degree distributions are more sensitive in this regard.

Figure 3.3 also demonstrates similar results as seen previously with the treelet frequency distances. A lot of network types have low intra-network variance in this instance, notably the road and peer-to-peer networks. The collaboration, communication, peer-to-peer, web, social, and small world graphs also all share a relatively low variance with each other. This could be attributed to the social actions and connections that form these graphs. As was seen with treelet frequencies, the road networks show little agreement with any of the other networks considered, likely due to their vastly different and mostly planar structure.

3.4.4 Clustering Using Treelet Frequency Counts

We also examine whether we can use the treelet occurrence frequencies to cluster networks into categories. The relative frequencies of all 4-9 vertex tree-structured subgraphs for the same ten network groups were considered to be a feature vector, and we used the the k-means and E-M clustering algorithms with the number of clusters set to 10. Approximately 70% and 75% average clustering accuracies were produced using the k-means and E-M algorithms, respectively. This is about the same accuracy as reported by Bordino et al. on undirected graphs (75%). However, they only considered seven network groups at a time, and used other topological features beyond subgraph frequencies. This result indicates that large treelet frequencies are useful features to consider when attempting to classify networks of various types.

3.4.5 Node and Edge Deletion and Edge Rewiring

As most real-world networks are incomplete, noisy, and dynamically evolving, the utility of treelet count for network analysis should be examined under these considerations. We select four networks and introduce varying amounts and types

of network alterations. A 100K vertex $G(n, p)$ graph, the Notre Dame web crawl, the Slashdot social network, and one of the Gnutella peer-to-peer snapshots were all modified by deleting vertices, deleting edges, and randomly rewiring edges. 5%, 10%, 20%, 50%, and 75% modifications of total vertices or edges were performed. The differences in treelet frequency distance between the modified and original network were then noted. As before, 1000 iterations were performed to retrieve the counts for all networks.

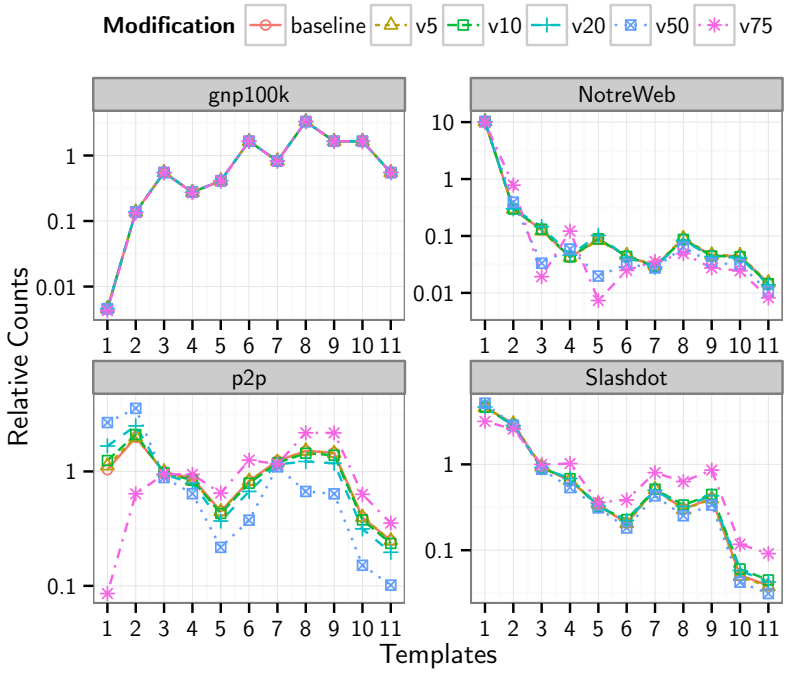

Figure 3.4. Treelet counts after 5%, 10%, 20%, 50%, and 75% **vertices** are deleted.

Figure 3.4 gives the results of vertex deletion on motif plots for all eleven different seven vertex templates (T7-1 to T7-11) on the four aforementioned networks. Vertex deletion has the most pronounced effect on the p2p network and the least effect on the gnp100k network. It is surprising that for some templates, the scaled counts are accurate even with 75% of the vertices deleted. This suggests that combining sampling-based schemes with color-coding might be quite effective to obtain counts for some templates.

Using the methodology to calculate treelet frequency distances (as originally presented with log scaling), we calculate disagreement values for all networks before and after perturbation. The maximal distance between the original and modified counts for any network was with the Notre Dame web crawl, having a disagreement

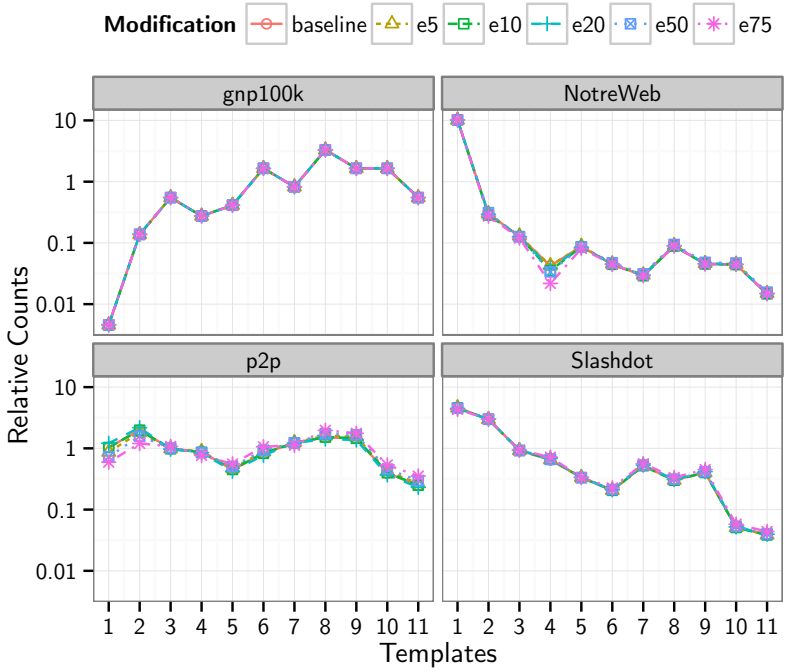

Figure 3.5. Treelet counts after 5%, 10%, 20%, 50%, and 75% **edges** are deleted.

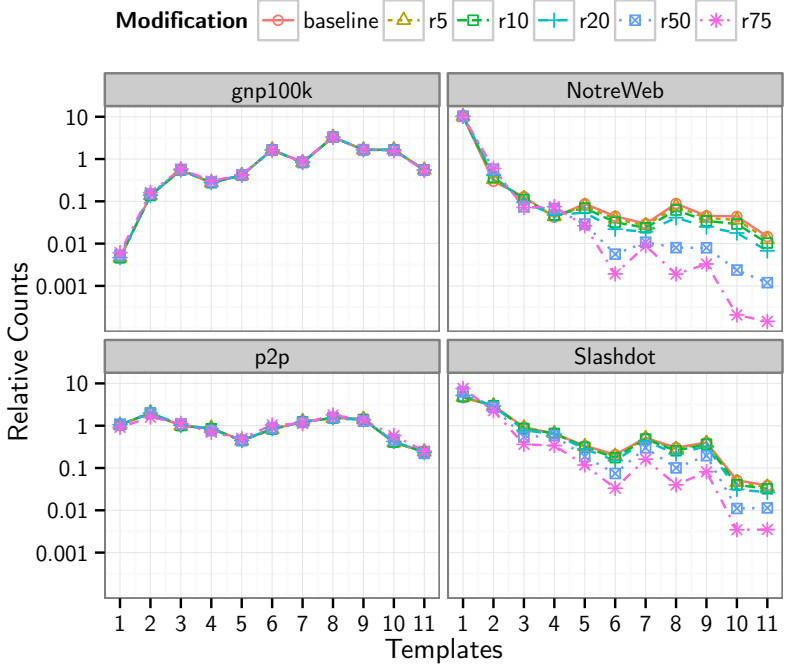

Figure 3.6. Subgraph counts after 5%, 10%, 20%, 50%, and 75% **edges are rewired**.

value of 4.1. With relative count scaling, this can be contrasted to the average disagreement value between all baseline networks, which is 9.2. The minimal

disagreement for the modified networks was with 100K vertex $G(n, p)$ graph, having an absolute disagreement of only 0.6. These results support the assertion that using treelet counts for network analysis can be useful even with incomplete networks.

Figure 3.5 gives results obtained with random edge removal from the networks. Edge removal has a lower impact on treelet counts as expected. The calculated maximal disagreement was with the Gnutella peer-to-peer graph, having a disagreement value of 1.2 with 75% edges removed. All other values were well below 1. This lends further credibility to the use of treelet analysis on networks with a high proportion of known vertices, but a lower confidence in known edges, such as protein interaction networks in computational biology.

We performed another study to observe the effect of randomly rewiring a proportion of edges within the network. These results are shown in Figure 3.6. Uninterestingly, the treelet counts on the random network once again show minimal change, along with the peer-to-peer network. However, the treelet counts on the social network and web crawl are quite different. A high degree of random rewiring seems to have a greater affect on the local topology in the social network and the web crawl.

The disagreement values calculated for Slashdot and the Notre Dame web crawl are relatively high for 75% rewiring (6.6 and 10.4, respectively), and the values calculated at 20% are much lower (1.0 and 2.4). Even with a rather high proportion of 20% spurious edges, the counts obtained on these modified networks are demonstrably similar to that of the original networks, providing further evidence to support the use of non-induced treelet counts in analysis of mildly noisy or incomplete networks.

3.4.6 Comparisons to Recent Work

Rahman et al. have recently designed GRAFT [57], a tool for quickly counting graphlets in large networks using a sampling-based technique. On the com-DBLP network from SNAP [69] ($n = 330K$, $m = 930K$), they report a single node execution time of about 47 seconds to count all 29 graphlets, with approximately 5% error. On the same network and with the same approximate error bound, FASCIA counts all 92 tree-structured templates of size 5 to 9 vertices, in about 78 seconds. Using networks grouped as peer-to-peer, collaboration, road, and

citation, Rahman et al. demonstrated that graphlets could be used for clustering networks, and reported 77% and 91% clustering accuracy rates when using 29 and 18 graphlets, respectively. As mentioned before, counts from Fascia achieve up to 75% accuracy for clustering networks into 10 groups.

3.5 Conclusions

Using the subgraph counting tool Fascia, we conducted an extensive study to highlight the efficacy of using large tree-structured templates to describe network topology, in the same way that smaller subgraphs of varying structures have previously been used. Additionally, we demonstrate the robustness of using these non-induced subgraph counts as graph signatures by quantifying the impact of random edge deletion and rewiring alterations on subgraph counts. Another use-case of subgraph counts is for the network alignment problem, which is finding large topologically-similar portions of disparate networks. The next chapter will introduce the problem and demonstrate how we can utilize the counts output by Fascia for its solution.

Chapter 4
Fast-Align Network Alignment

4.1 Introduction

This chapter will discuss the use of the subgraph counts output by Fascia for the purpose of network alignment. We introduce FastAlign, which is our implementation of a prior alignment algorithm that uses the counts of small subgraphs for alignment of biological networks. Through using the larger subgraphs countable by Fascia, our FastAlign algorithm is able to produce network alignments of a higher quality and with lower computational effort relative to the prior approach.

4.2 Network Alignment

The NP-Complete network alignment problem is determining a vertex to vertex mapping between two disparate networks that minimizes some cost function. This cost function is usually calculated based on how well the mapping aligns vertices and edges between the networks. This can be considered a generalization of the graph or subgraph isomorphism problems, where the cost is zero if an exact mapping is found or infinity otherwise. Also, similar to the subgraph isomorphism problem, determining alignments between networks can often reveal latent functional similarities between the network structures. Network alignments can be cheaper to calculate than exact subgraph matches (depending on the algorithm and subgraph size), so determining alignments of very large networks is feasible.

In previous chapters, we introduced graphlets and some of their associated uses. Graphlets are formally defined as all possible undirected induced subgraphs

from 2-5 vertices [51]. Using graphlets counts, one can calculate a global similarity measure between networks as the total distances between all different graphlet counts. This is termed as the graphlet frequency distance [59]. Similarly, a distance can be calculated based of graphlet degree distributions, where the graphlet degree of a vertex is the number of graphlet embeddings rooted at that vertex [1]. By taking a single vertex and calculated the degrees for all possible graphlets rooted at that vertex, a vector can be constructed (graphlet degree signature) for inter-vertex comparisons [60]. Finally, by examining the graphlet degree signatures for nodes within and between two distinct networks, an alignment between the networks can be determined [89–93].

We previously demonstrated that treelet counts produced from Fascia are more computationally efficient to compute and can take the place of graphlet counts for a number of the aforementioned graphlet-based analytics [42]. The primary focus of this chapter will be to demonstrate the possibility of using a treelet degree signature vector, calculated through approximate treelet counts, as a means to align biological networks. We utilize the GRAAL alignment algorithm, explained in [89, 90], to develop our own implementation, FastAlign. We then do a comparison in alignment qualities produced between using treelet or graphlet counts. We demonstrate that treelet counts can be more useful than graphlet counts for the alignment of moderate scale biological networks.

Other relevant work has examined the applicability of using the color-coding approach for network alignment in gene regulatory and protein interaction networks [63, 64, 94, 95]. These approaches utilize a clever extension of the color-coding dynamic programming phase based on local sequence alignment algorithms (such as Smith-Waterman) to account for the possibility of insertions and deletions in the alignment. Due to high memory requirements which scale exponentially with the size of the query network, query sizes greater than 12 vertices soon become infeasible with this general approach.

4.3 Background

4.3.1 Graphlets

As previously stated, graphlets are formally defined as small undirected induced subgraphs between two and five vertices in size. Prior work by Pržulj et al. [1, 51, 59, 60] extensively studies graphlets in the context of biological network analysis. Pržulj et al. also identified all possible discrete *orbits* within each graphlet. Orbits in this context refer to distinct automorphic vertices in the subgraph; i.e., it can be useful to explicitly differentiate between an embedding rooted on a vertex in the center of a star versus a vertex on one of the leaves. All possible graphlets and identified orbits are given in Figure 4.1.

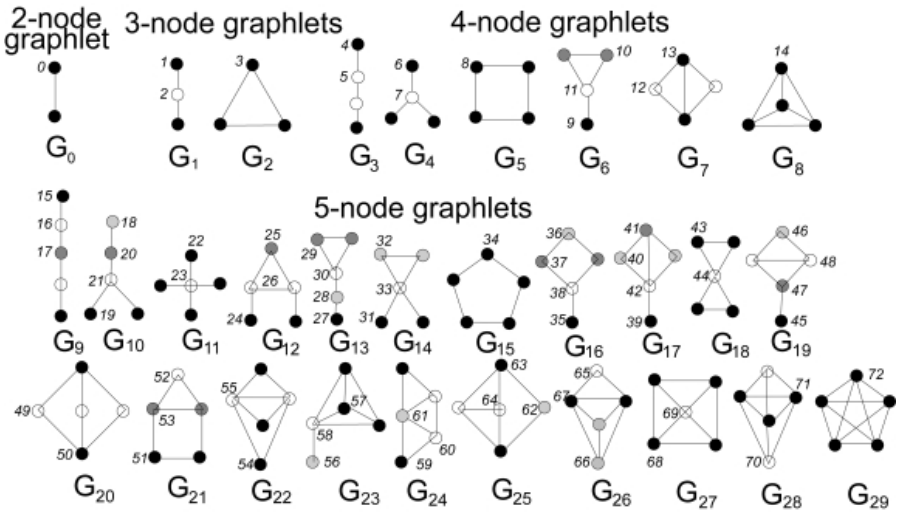

Figure 4.1. All possible graphlets and orbits. From [1].

4.3.1.1 Graphlet Degree Signature Similarity

The graphlet degree signature similarity was previously described in Chapter 3, but we formally define it again here. The graphlet degree signature similarity is a per-vertex metric that allows comparison between two disparate vertices in the same or separate networks. This metric is based on a vector created with the counts for all possible graphlet orbits rooted at the vertices. It is described as capturing the local topology and interconnectedness of the node in the context of its local

neighborhood [60]. The similarity value between two vertices, u and v, for orbit i, with counts u_i and v_i, respectively, is calculated as follows:

$$S_i(u,v) = 1 - w_i \times \frac{|\log(u_i+1) - \log(v_i+1)|}{\log(\max\{u_i, v_i\}+2)}$$

In this equation, w_i is a certain weighting given to that specific orbit. These values are dependent on the number of isomorphisms of smaller graphlets orbits that exist at that orbit in its respective graphlet. The total similarity value between vertices is the sum of similarity values divided by the sum of the weightings for all orbits.

4.3.2 Treelets

Treelets in the context of this work are formally defined as all possible 3-7 vertex tree-structured non-induced subgraphs. As mentioned, previous work [42] has demonstrated the applicability of using treelets in lieu of graphlets to benefit from the much lower possible running time bounds. Counting graphlets requires an upper bound of $O(n^5)$ time where n is the number of vertices in the graph. There have been sampling methods introduced to improve time to solution (e.g. [57]), but these still do not improve upon the upper bound, have relatively high error, and have not yet been demonstrated as applicable for producing per-vertex counts, only global counts. Counting treelets, on the other hand, scales linearly in $O(m)$ time using the color-coding approach, where m is the number of edges in the network, and can be used to count the number of treelet embeddings rooted at each vertex in a graph. Note that the bound given here is even looser than it appears, since as network size increases, the necessary number of iterations needed to retain the same approximation error decreases [10, 43].

4.3.3 GRAAL

The GRAAL (GRAph ALignment) algorithm and its variants [89, 91–93] constitute various approaches for the use of graphlet degree signature similarity between vertices of different networks to compute an optimal alignment. An overview of the baseline GRAAL algorithm implemented in this work is given by Algorithm 4.1, as described in [90]. The algorithm proceeds as follows. Firstly, a cost matrix C is

created between all possible vertices v and u in between the two networks G and H based on the following function:

$$C(u,v) = 2((1-\alpha) \times \frac{v_d + u_d}{G_{max} + H_{max}} + \alpha \times S(u,v))$$

In this function, v_d and u_d are the degrees of vertex v and u, G_{max} and H_{max} are the maximal degrees of graphs G and H, $S(v,u)$ is the graphlet degree signature similarity between v and u, and α is a control parameter between [0-1] that varies the influence of the vertex degrees versus signature similarity on the overall cost. An α value of 0 would specify that only vertex degrees are to be utilized, while an α value of 1 would result in only vertex counts being utilized.

Algorithm 4.1 GRAAL Alignment Algorithm

```
procedure GRAAL(G, H)
    C ← allCosts(G, H)
    A ← ∅
    p ← 1
    while G, H not fully aligned do
        (u, v) ← findSeed(G^p, H^p)
        A ← (u, v)
        r ← 1
        repeat
            R_G ← getRadius(G^p, u, r)
            R_H ← getRadius(H^p, v, r)
            A ← align(R_G, R_H, C)
            r ← r + 1
        until R_G or R_H = ∅
        if r ≥ 3 and p < 3 then
            p ← p + 1
return A
```

Using the cost matrix, an initial *seed* is selected as the minimal value pair in C and added to the alignment A. The networks are then iteratively and greedily aligned (based on minimal cost) outward from this pair of vertices on a per-radius basis (e.g. 1 hop from u and v, 2 hops, 3 hops, etc.) until no more vertices are available for alignment in one of the graphs at a given radius.

If the resultant radius is greater than or equal to 3, the graphs are taken to the next *power*. In this instance, power refers to a graph that is created by adding edges to the graph between all vertices having some shortest paths length between them up to some value in the original graph. For example, a power 1 graph would just be the original graph (i.e. $G^1 = G$), while a power 2 graph would have additional edges

between all vertices that are at most 2 hops away from each other on the original graph. This allows for inexactness in the alignment, similar to how additions or deletions function in sequence alignment.

New seeds are selected and the greedy iterative alignment is again performed for each radius. The alignment then continues with new seeds being selected and the graphs incremented as necessary up to a maximal power as 3 until all possible vertices in the smaller of G and H have been fully aligned.

4.3.4 Alignment Evaluation

There are three metrics commonly used to evaluate the quality of alignment between two biological networks. These are edge correctness, node correctness and interaction correctness. Given an alignment in terms of a mapping function M from vertex sets $V_G \in G$ to $V_H \in H$, we can define edge correctness as the following:

$$EC = \frac{|(u,v) \in E_G : (M(u), M(v)) \in E_H|}{|E_G|}$$

where E_G and E_H are the edge sets of G and H, respectively. Edge correctness can be simply stated as the ratio of the number of edges that exist in G that equivalently end up mapped to an existing edge in H over the total number of edges in G.

When labels exist for each vertex, e.g. proteins existing in disparate protein interaction networks, node correctness is equivalently defined as:

$$NV = \frac{|u \in V_G, v \in V_H : M(u) = v, L(u) = L(v)|}{|V_G|}$$

where L defines the labeling for each vertex. This is simply stated as the number of matched labels when mapping the vertices in G to H over the number of vertices in G.

Using a combination of vertex mappings and labels, we can finally define interaction correctness as:

$$IC = \frac{|(u,v) \in E_G : (M(u), M(v)) \in E_H, L(u) = L(M(u)), L(v) = L(M(v))|}{|E_G|}$$

Network	n	m	Source
Yeast	5.1 K	22 K	[76]
Human	3.7 K	5.1 K	[76]
Yeast_lc	5.0 K	22 K	[76]
Human_lc	2.8 K	4.4 K	[76]
Yeast2	2.3 K	16 K	[97]
Human1	9.1 K	41 K	[77]
Yeast2_lc	2.0 K	16 k	[97]
Human1_lc	8.9 K	41 K	[77]
Anthrax	2.6 K	3.1 K	[98]
Bubonic Plague	3.3 K	4.1 K	[98]
Mesorhizobium	1.8 K	3.1 K	[99]
Synechocystis	1.9 K	3.1 K	[99]

Table 4.1. 12 networks comprising the 8 alignments that were used for testing. The four bottom networks were all aligned to the Human1 network.

Which is the number of interactions between identified vertices in G that are correspondingly mapped to the same interaction in H. It follows that $EC \geq NC \geq IC$. Because not all of the network datasets used for this work included explicit node labels, edge correctness is the primary metric used to evaluate alignment quality in this work.

4.4 Experimental Setup

Experiments and reported execution times were retrieved from *Compton* at Sandia National Labs. FASCIA was built with the Intel C++ compiler (version 13), used OpenMP for multithreading, the `-O3` optimization parameter, and utilized the environment variable `KMP_AFFINITY` for controlling thread to core pinning. The GRAAL executables were retrieved from [96]. The alignment algorithm was re-implemented by us as FASTALIGN due to very slow execution times and high memory overheads of the supplied executable. The supplied graphlet counting tool, however, was still used as-given.

Several different alignments were attempted and the networks utilized are listed in Table 4.1. The top 8 networks were aligned correspondingly to the network co-listed, while the bottom four networks were all aligned the the Human1 network. All networks listed are protein-protein interaction (PPI) networks. The networks were

downloaded from the Database of Interacting Proteins [76] and the supplementary website for the C-GRAAL paper [93, 100]. To test for the influence of noise on alignment quality, the Human, Human1, Yeast, and Yeast2 networks were also parsed to extract only the largest connected component from the graph. This had mild structural impact with the exception of the Human network, where about 35% of vertices and edges were removed.

4.5 Results

We are going to compare the performance of both graphlet and treelet counts for network alignment utilizing the original GRAAL algorithm. Comparisons are going to be made with regards to execution time required for subgraph count calculation and resultant alignment quality in terms of edge correctness. Additionally, to determine the significance of results, we also utilize a *Random* approach, which was created by replacing the similarity value $S(u, v)$ in the cost function with a random `float` instead of having it calculated as an explicit difference in graphlet or treelet counts between vertices.

Note that the overall GRAAL algorithm is not optimal or deterministic, so results can be variable on a run-to-run basis. To extract the best alignment, one would need to perform a large number of runs and determine the best output from this. However, since this analysis is considering overall trends as opposed to optimality, and the different outputs are usually within only about 10% of each other, the results reported are from only a single run.

4.5.1 Execution Times

Figure 4.2 gives the execution times of both Fascia and the graphlet counting portion of GRAAL for the Human, Human1, Yeast, and Yeast2 networks and their largest connected components. Reported execution time is the sum total for counting all 73 graphlets orbits or 83 treelet orbits. Fascia is run to 1000 iterations, which previous work [10, 42, 43] has established is more than sufficient to minimize approximation error for networks of this scale.

From Figure 4.2, it is apparent why there is a desire to utilize treelets in lieu of graphlets for network alignment. Fascia demonstrates superior scalability due to

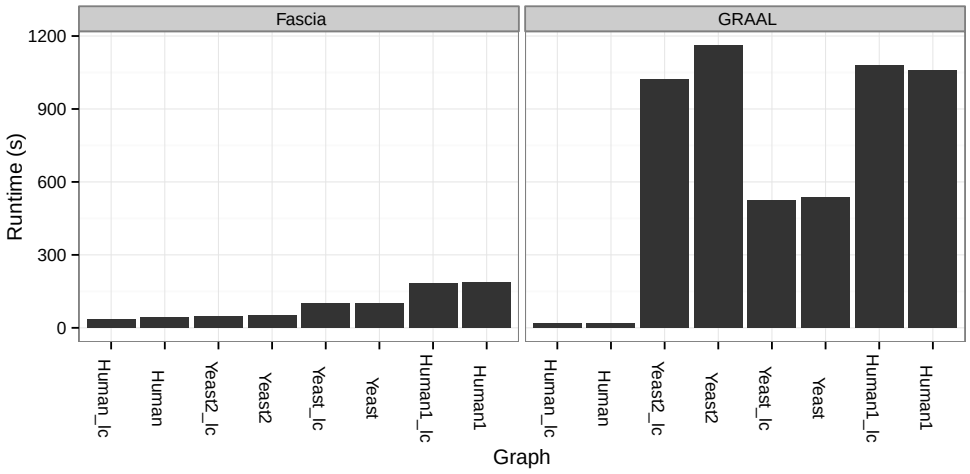

Figure 4.2. Execution times for both FASCIA and GRAAL to count several networks.

the $O(m)$ versus $O(n^5)$ execution time bounds. Note that the graphs are listed in increasing edge count from left to right, and that, although the Yeast network is larger than the Yeast2 network, it has lower relative execution time with GRAAL due to a lower vertex count.

4.5.2 Alignment Analysis

The previous GRAAL work has established that an α value in the cost function of about 0.8 often results in the best alignment with GRAAL. Figure 4.3 demonstrates the alignment quality in terms of edge correctness for all 8 alignments with α fixed at 0.8. From this Figure, it appears that, relative to the random cost alignment, FASCIA's treelet counts demonstrate good performance on the moderately-sized pathogen and bacteria alignments while graphlet counts shows equivalent or better performance on the larger and more complex Human1 and Yeast2 alignments.

Worth noting is the increase in alignment quality that results between the HumanYeast and Human1Yeast2 alignments and their largest connected components. There are two factors at play here. The dominant one is the reduction in noise, which will increase the core alignment of the largest component while simultaneously decreasing the denominators of the edge correctness calculations, resulting in an increase in edge correctness as seen. However, an additional factor is observed with the especially disconnected Human network. Aligning this network will result in good alignments of all of its disconnected components, since a number

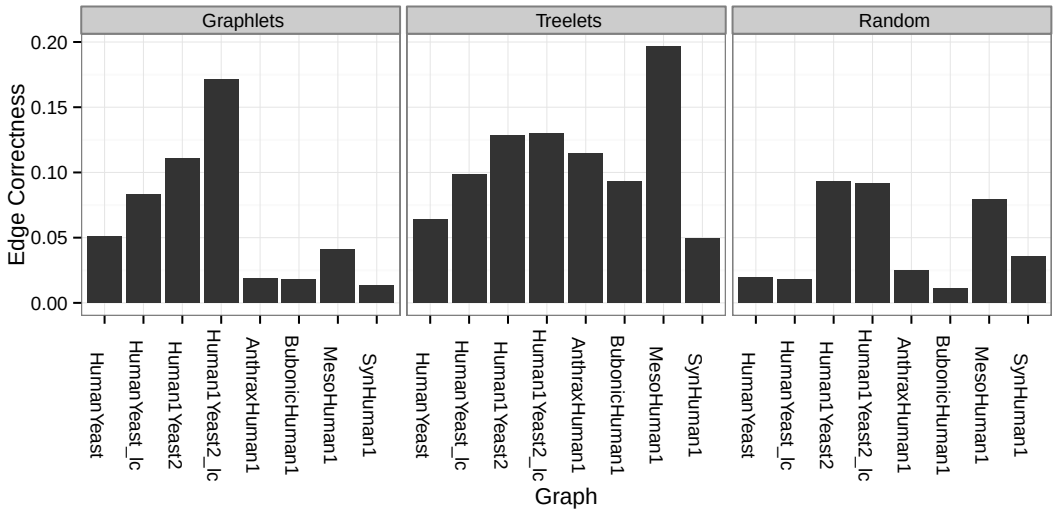

Figure 4.3. Edge correctness across all network alignments with the α parameter fixed at 0.8.

of them are only one hop in width and therefore have a good chance of perfectly aligning with the implemented algorithm.

To analyze the effect of the α parameter, we then look at the alignments across alpha values from 0.0 to 1.0 and compare treelets, graphlets, and random edge correctnesses that results on all network alignments. This is given in Figure 4.4.

The results in Figure 4.4 appear to follow those in Figure 4.3, in that graphlets shows good performance on the larger networks while treelets demonstrates superior alignments on the smaller pathogen and bacterial networks. In general, it appears that alignment quality decreases with increasing influence of subgraph counts for a number of networks. Overall, one can take the alignment quality at $\alpha = 1.0$ to be the impact of subgraph counts themselves on the ability of this approach to align networks.

Finally, we consider the possibility that Fascia counting accuracy can have an effect on alignment quality. To test this, we perform additional counts at 1, 10, 100, and 10K iterations. We analyze both the Human1Yeast2_lc alignment, where Fascia and treelets demonstrated near equivalent performance to random and worse than graphlets for most α values, and the AnthraxHuman1 alignment, where treelets demonstrated superior performance to both graphlets and random. We compare alignment quality from these to that output with the random approach in Figure 4.5.

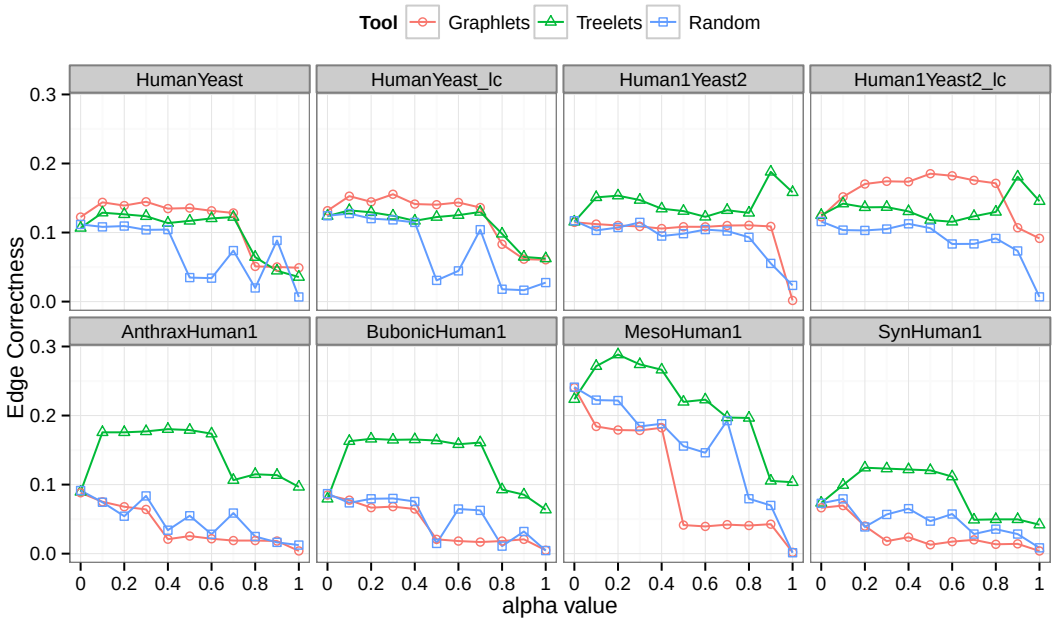

Figure 4.4. Edge correctness across all alignments with a variable α parameter.

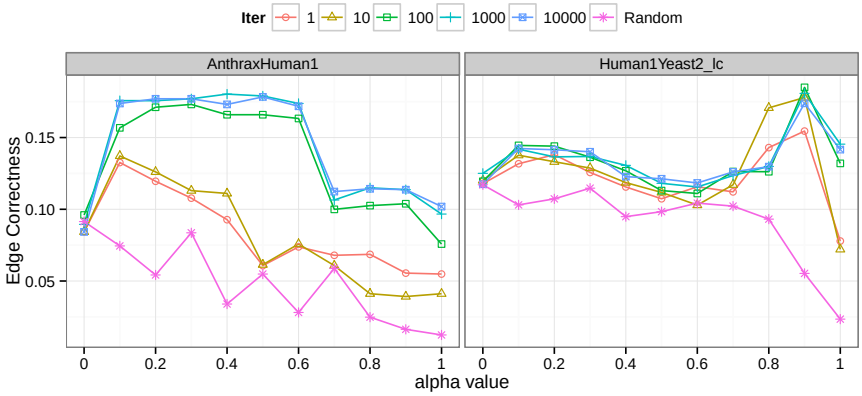

Figure 4.5. Edge correctness across all alignments with a variable α parameter.

From Figure 4.5, we can observe that about 100 iterations were necessary to produce appropriate counting accuracy for the AnthraxHuman1 alignment. Further iterations beyond 100 appear to have minimal effect. On the Human1Yeast2_lc alignment, we observe that the edge correctness produced is fairly independent on counting accuracy, but still better than random. Without directly observing graph structure or the explicit alignment being produced, it is an educated guess that the

alignment is highly dependent on a few major vertices in the core of the networks. As touched on previously, counting accuracy increases with network size and density, and only a few iterations are necessary to produce very low approximation error. Note that the Anthrax network is both considerably smaller and less dense than the Human1 and Yeast2 networks, so correspondingly requires a greater number of iterations.

4.6 Conclusions

This chapter introduced our implementation of the GRAAL graph alignment algorithm, FastAlign, which uses subgraph counts to greedily align two distinct networks. We compared the alignment qualities of both graphlet and treelet counts across a range of biological network alignments. Based on the limited experiments, it appears the graphlet counts can be better suited for aligning larger and more dense networks, while treelets are effective with smaller and less dense networks. Both approaches were significantly better than random on most alignments.

Subgraph counting is only one possible application of the color-coding approach. In the next chapter, we will introduce our extension of our baseline color-coding implementation for the solution of the minimum-weight path finding problem and demonstrate its usefulness for detecting signaling pathways in protein interaction networks.

Chapter 5
FastPath Minimum Weight Path Finding

This chapter demonstrates how the optimizations we introduced with Fascia are generalizable to other color-coding-based algorithms with our parallel minimum weight path finding tool, FastPath. As we will show, FastPath uses the core techniques of Fascia to find minimum weight paths more quickly than prior work and with less overhead.

5.1 Introduction

Consider the NP-hard optimization problem [101] of finding the *minimum-weight simple path* of path length L in a weighted graph with positive edge weights. This problem is of considerable interest in bioinformatics, specifically in the analysis of paths in protein interaction networks. With an appropriately-defined edge weight scheme, paths with the minimum weight, or in general close to the minimum weight, often have vertices that belong to biologically-significant subgraphs such as signaling and metabolic pathways [102, 103]. As in the case of subgraph counting, color-coding can only offer an approximate solution to this NP-hard problem. With some confidence and error bounds, it is guaranteed to return simple paths with a weight close to the minimum path weight. The low-weight paths returned through color-coding on protein interaction networks are shown to be good candidates for signaling pathways [101]. We present a shared-memory parallelization of the approximate low-weight path enumeration strategy, which we term as FastPath.

5.2 Related Work

Scott et al. were the first to use the color-coding technique to find low-weight paths, with the use case of detecting signaling pathways in protein interaction networks [101]. Vertices in these networks are proteins, and edge weights are the negative log of the probability that the two proteins interact. Thus, simple paths with low weights correspond to chains of protein that would interact with high confidence. Hüffner et al. [66] expanded on this initial work by offering several optimizations to the baseline algorithm to improve running times, including choosing an appropriate number of colors to decrease iteration counts and implementing a pruning strategy that complements coloring. More recently, Gabr et al. [65] further decreased the number of iterations required for a given confidence bound through per-iteration examination of graph colorings. Color-coding has also been used for querying linear pathways in protein interaction networks by Shlomi et al. [95]. This work was expanded for more complex bounded tree-width queries by Dost et al. [94]. Similar to the aforementioned Gabr et al. work, Gülsoy et al. [63, 64] speeds up pathway and small bounded tree-width querying by also reducing the number of iterations required for a given confidence bound.

5.3 Enumerating low-weight simple paths with FastPath

We now present a color-coding based scheme to enumerate simple paths of length L in a graph with positive edge weights. Finding the minimum-weight path is an NP-hard problem, but color-coding gives us an approximation algorithm whose cost is linear in the number of edges in the graph, but exponential in the value L. The main idea is the same as the subgraph counting case: instead of enumerating all paths of length L and looking for a simple path with the minimum weight, we instead only search for *colorful paths* by randomly coloring vertices. There are prior approaches and tools that implement this strategy. However, prior work has primarily focused on reducing running time by limiting the required number of iterations for a given confidence bound [65, 101], with the exception of the approach of Hüffner et al. [66]. Here, we will only consider minimizing per-iteration costs

through the previously-described optimizations utilized for FASCIA (combinatorial table indexing, memory-reducing optimizations, partitioning, multithreading). Our approach can be combined with topology-aware coloring methods to further reduce end-to-end running time.

Algorithm 5.1 FASTPATH: Enumerating low-weight simple paths using color-coding.

```
Initialize all entries of a min heap H of size n_L to ∞
for it = 1 to Niter do                                   ▷ Outer loop parallelism
    Color G(V, E) with k colors
    Initialize all Weights[1][v ∈ V_1][1 ··· c_1] ← ∞
    for i = 2 to L + 1 do
        for all v ∈ V_i do                               ▷ Inner loop parallelism
            for all color sets C do
                min_w ← ∞
                for all C_a, C_p ∈ C do
                    for all u ∈ N(v) do
                        w_a ← GetEdgeWeight(u, v)
                        w_p ← Weights[i − 1][u][C_p]
                        if w_a + w_p ≤ min_w then
                            min_w ← w_a + w_p
                if min_w < H.max then                    ▷ Critical section
                    if i = L + 1 then
                        insert min_w into H
                    else
                        Weights[i][v][C] ← min_w
Return H as output.
```

Algorithm 5.1 gives an overview of the general approach for finding low-weight simple paths. The algorithm is similar to the general color-coding template for subgraph counting. Since this implementation only considers simple paths (which can be considered a tree template) rather than a more complex template, we can simplify the partitioning phase. We avoid partitioning completely by assuming that we already performed a one-at-a-time partitioning, and have set the active child as the single cut vertex at each step in the partitioning tree.

To simplify the description of the algorithm, we only show weights of the n_L least-weight colorful paths being stored in Algorithm 5.1. In our actual implementation, we also store the corresponding vertices in the low-weight path as an array of integers. In prior work, paths have also been stored using compressed representations [66,101].

We use a min heap H of size n_L to store the best weights and the corresponding paths.

Algorithm 5.1 has L inner loop iterations. At each step, we are attempting to find over all $v \in V_i$ the least-weight colorful path that ends at v, for every possible color set C. Initially, weights for all vertices and colorsets are set to ∞ for a single vertex path. For succeeding steps, we look at the sum of weights of all previously discovered paths ending on neighbors u of v, while considering adding the weight of the edge between u and v. For each color set C, we take the minimum and store the summed weight of the path in Weight$[i][v][C]$.

We can also compare the weights found during each step to the current highest value in the min heap, and store the path only if it is one the current n_L lowest-weight paths. These paths are inserted into the heap in the final step of the inner loop ($i = L + 1$). We update the heap over subsequent iterations, storing better paths if we find them. This decreases memory requirements for subsequent iterations by avoiding unnecessary storage of heavy paths in the Weights array.

There are FastPath-specific issues to note with regards to memory utilization. Storing the actual paths for all color sets for all vertices can increase memory costs considerably. However, the biological networks and path lengths examined are usually both small enough that memory is not a concern. Additionally, there is often a predefined directivity in the input paths (e.g. finding a path between membrane proteins and transcription factors), and this allows us to restrict the size of the table for each step i by only placing a subset of possible vertices into V_i with per-vertex initializations. Using a min heap with a small n_L value will also substantially decrease memory requirements after the first few iterations.

Finally, note that we implement both inner-loop and outer-loop parallelism here, similar to Fascia. For the size of biological networks commonly considered for the minimum-weight path problem, outer loop parallelism performs considerably better. If every outer loop thread maintains its own min heap, we can avoid the synchronized heap insertions that inner loop parallelism requires. After all iterations are complete, we can simply examine all heaps and return the n_L-best unique paths.

Network	n	m	d_{avg}	d_{max}	$\widetilde{D}$	Source
Human	9.0 K	22 K	5.0	322	14	[104]
Caenorhabditis	3.2 K	5.5 K	3.4	186	14	[104]
Drosophila	7.2 K	21 K	5.9	176	12	[104]
Mammalia	8.8 K	19 K	4.4	323	18	[104]

Table 5.1. Network sizes and average and maximum degrees and approximate diameter for the networks used in our analysis.

5.3.1 Experimental Setup

Experiments were performed on the Compton testbed at Sandia National Laboratories. We evaluate performance of our FastPath implementation on a collection of biological networks, listed in Table 5.1. We considered the weighted Human, Caenorhabditis (C. elegans genus), Drosophila (fruit flies), and Mammalia protein interaction networks from the Molecular INTeraction database [104].

5.4 FastPath performance

In this section, we analyze our FastPath implementation for finding minimum-weight paths in the weighted Human, Drosophila, Caenorhabditis, and Mammalia protein interaction networks from the MINT database. We compare our per-iteration running times to the current state-of-the-art serial code of Hüffner et al. (HWZ) [66]. We do not directly compare to other work, as most other work has focused on reducing the number of color-coding iterations, rather than improving per-iteration performance.

Figure 5.1 gives the running times for FastPath and HWZ to determine the 100 best 4-9 length paths over 500 search iterations. We report serial and parallel performance of FastPath, with our parallel code run across 16 cores on a Compton node. We include both the Hüffner et al. baseline color-coding approach (HWZ), as well as their dynamic programming heuristic technique (HWZ-Heuristic). We also analyzed outputs, and noticed that all four approaches find the same paths with the same minimum weights.

From Figure 5.1, we observe that our parallel FastPath implementation demonstrates considerable speedup across all test instances. We note that the running times of serial FastPath are close to that of HWZ-Heuristic on most

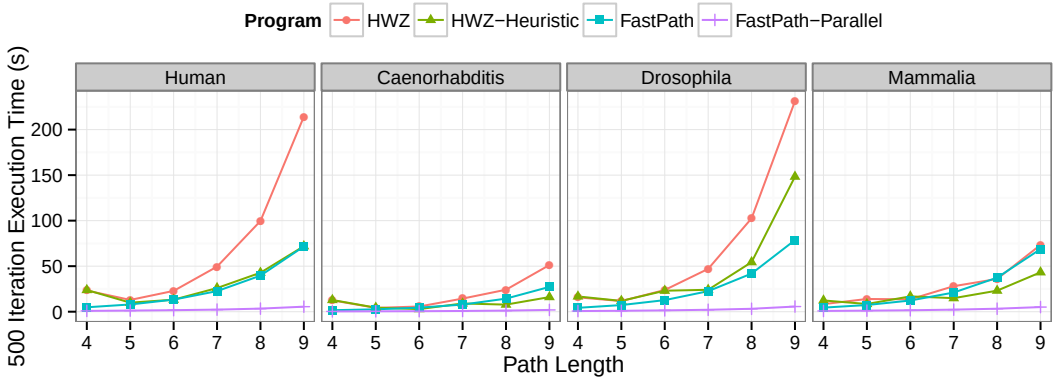

Figure 5.1. Absolute running times for 500 iterations of finding path lengths 4 through 9 using the Hüffner et al. baseline and heuristic methods, as well as FastPath in serial and on 16 cores.

tests, and that the heuristic offers improved speedup to the HWZ baseline. Our current version of FastPath does not implement the HWZ heuristic, but future work combining both our and Hüffner et al.'s optimizations could lead to improved performance.

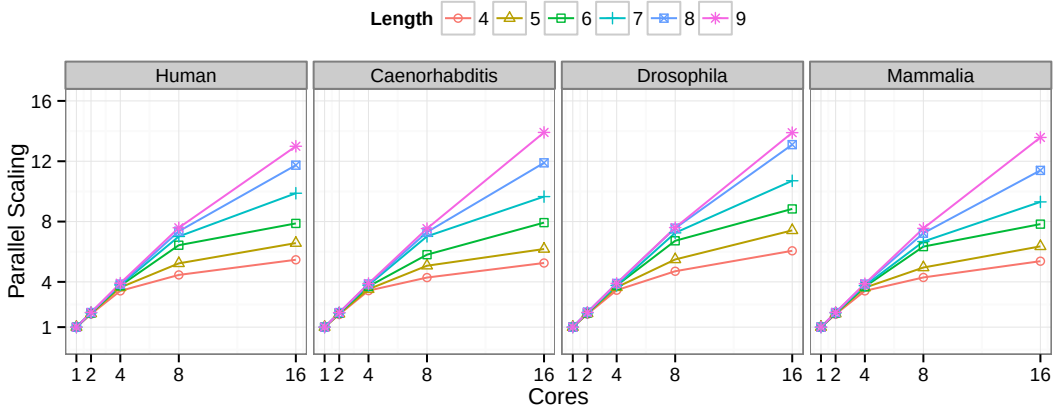

Figure 5.2. Speedup for FastPath from 1 to 16 cores for path lengths 4 through 9.

Figure 5.2 gives the parallel speedup from 1 to 16 cores for all the tested path lengths and networks. We observe that 500 iterations of path length 9 takes about 5 seconds on the Human network, with a parallel speedup of about 12.5×. We note that speedup is dependent on the length of the path being searched, since longer paths generate more overall work and greater relative parallelism.

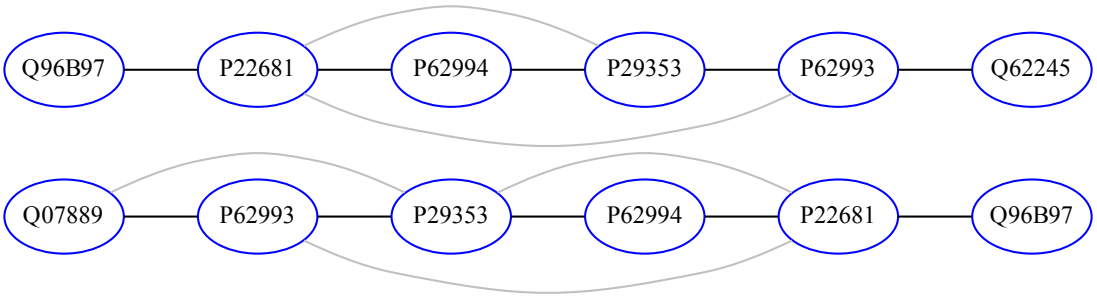

Figure 5.3. Sample minimum-weight paths of path length five found in the MINT Human PIN using FastPath (top) and FASPAD (bottom). The path weight is 0.0211329 in both cases.

We also employ FastPath to see if we can determine the minimum path weight in the Human-MINT network for different values of L. We performed a search with paths of length 5 using FastPath and FASPAD [105], a tool based on the Hüffner et almethod. We found several minimum weight paths in the network in just a few iterations. Two paths are shown in Figure 5.3, one generated using FastPath and the other using FASPAD. The simple path detected is shown with black edges, and other edges connecting proteins in the path are shown in grey. Further analysis using DAVID [106, 107] reveals that proteins in the high-scoring paths appear in the well-studied chronic myeloid leukemia KEGG pathway [108]. We also compared our results with those presented by Gabr et al. [65] for different path lengths, and despite the fact that we don't explicitly restrict the search space from membrane proteins to transcription factors in our test, we notice the same proteins appearing in both our works.

Finally, we demonstrate the statistical significance of the paths found using FastPath. We use the standard score metric, also known as a z-score, which gives the number of standard deviations a given path weight is from the mean path weight determined over some sample of paths. The z-score is calculated as $z = \frac{x-\mu}{\sigma}$, where x is a given single path weight and μ and σ are the mean and standard deviation path weights over the sample, respectively.

Using all four networks and path lengths from 3 to 8, we take the mean (μ) and standard deviation (σ) of weights from 1000 randomly-selected paths, and calculate the z-scores (z) using the lowest weight (x) returned by FastPath after 500 iterations. Table 5.2 gives these results. The statistical significance of the paths we find is apparent. The z-scores obtained on the Drosophila and Caenorhabditis

Network	Path Length	x	μ	σ	z
Human	3	0.003	3.37	0.87	3.80
	4	0.012	4.55	1.07	4.22
	5	0.021	5.67	1.22	4.61
	6	0.045	6.76	1.45	4.61
	7	0.065	7.92	1.59	4.92
	8	0.086	9.03	1.72	5.19
Caenorhabditis	3	0.65	3.73	0.27	11.1
	4	1.49	4.98	0.32	10.6
	5	2.51	6.23	0.33	11.2
	6	3.27	7.46	0.37	11.2
	7	4.03	8.71	0.38	12.2
	8	4.86	9.95	0.41	12.2
Drosophila	3	1.12	3.76	0.25	10.3
	4	1.80	5.02	0.24	13.1
	5	2.65	6.27	0.29	12.3
	6	3.64	7.53	0.23	16.6
	7	4.37	8.80	0.23	19.0
	8	4.81	10.0	0.26	19.7
Mammalia	3	0.046	3.42	0.98	3.44
	4	0.063	4.62	1.20	3.80
	5	0.116	5.70	1.48	3.77
	6	0.158	6.85	1.56	4.29
	7	0.178	8.04	1.68	4.68
	8	0.286	9.34	1.88	4.82

Table 5.2. The lowest weight paths obtained with FastPath for several networks and path lengths, along with its z-score calculated using the mean and standard deviation of a random sample of paths.

networks are especially high. As was similarly noted by Gabr et al., we observe that z-score consistently increases with path length. This demonstrates the importance of scalable methods for finding low-weight paths of large L. In future work, we will perform a detailed exploration of the biological significance of the paths found, and see if they correlate with the apparent statistical significance.

5.5 Conclusions

This chapter introduced FastPath, a tool for finding minimum weight simple paths in weighted edge networks. We utilize the color-coding method along with optimizations borrowed from Fascia to demonstrate good serial and parallel performance relative to prior art. We also demonstrate the applicability of using such a tool for network analysis by finding several relevant and known pathways in protein interaction networks. In the next chapter, we will move away from

color-coding and subgraphs and path-based problems onto the topic of graph connectivity.

Chapter 6
Multistep Method for Graph Connectivity

6.1 Introduction

This chapter introduces the Multistep method, which is a shared-memory parallel algorithm for the decomposition of a directed graph into its strongly connected components. We reduce the strongly connected components decomposition problem into numerous subroutines. We optimize each of these subroutines for multicore platforms and demonstrate the effectiveness of our approach relative to prior art. Again, our techniques are very generalizable, as with minimal modification we are able to use the same subroutines for other related connectivity problems.

6.2 Strongly Connected Components

The problem of strongly connected components (SCCs) decomposition refers to detection of all maximal strongly connected subgraphs in a large directed graph. Informally, a strongly connected subgraph is a subgraph in which there is a path from every vertex to every other vertex. SCC decomposition is a useful preprocessing and data reduction strategy when analyzing large web graphs [11] and networks constructed from online social network data [12]. SCC detection also has several uses in formal verification tools, including model checking in state space graphs [15]. Other application areas include computer-aided design [109] and scientific computing [16].

Tarjan's algorithm [18] is an efficient serial algorithm for SCC detection. It uses a recursive depth first search (DFS) to form a search tree of explored vertices. The roots of subtrees of the search tree form roots of strongly connected components. Although it is possible to extract parallelism from DFS while retaining proper vertex ordering, this is often met with limited success [110]. Therefore, most parallel SCC algorithms have avoided its use.

The Forward-Backward (FW-BW) method [19] and Orzan's color propagation method [15] are two SCC detection algorithms that are amenable to both shared-memory and distributed-memory implementation. These methods use very different subroutines and were proposed in different contexts, FW-BW for graphs arising in scientific computing and color propagation in the context of formal verification tool design. We observe that baseline shared-memory parallelization of both methods perform poorly in comparison to the serial algorithm for SCC decomposition in irregular and small-world graphs, such as social networks and web crawls.

In this chapter, we present a new parallel algorithm for SCC, the MULTISTEP method, and problems related to SCC, connected components (CC) and weakly connected components (WCC). Finding the connected components of an undirected graph is to find all maximal subgraphs in the graph graph where there is a path between all vertices. Weakly connected components are the equivalent of connected components but in a directed graph where edge directivity is ignored. Known efficient serial and parallel algorithms for these problems can be very different from SCC detection algorithms. However, we observe that the approaches we present are easily extended to solve these problems as well.

6.2.1 Contributions

Our new MULTISTEP method is designed for SCC detection in large real-world graphs, such as online social networks and web crawls, using current shared-memory multicore platforms. We utilize variants of FW-BW and Orzan's color propagation methods in subroutines. We minimize synchronization and avoid use of fine-grained locking. Our BFS subroutine incorporates several recently-identified optimizations for low-diameter graphs and multicore platforms [111–113]. We perform an extensive experimental study on a 16-core Intel Xeon server with Sandy Bridge processors, and the following are our main observations:

- For low-diameter networks (e.g., Twitter, LiveJournal crawls), the single-threaded MULTISTEP approach is significantly faster than the serial Tarjan algorithm.
- MULTISTEP is faster and exhibits better scaling than our implementations of the FW-BW and coloring algorithms.
- MULTISTEP is up to 8.9× faster than the state-of-the-art Hong et al. method [114] on ItWeb, a network with a large number of SCCs (30 million).
- MULTISTEP modified for CC is consistently faster than the coloring-based algorithm implementation in Ligra [115].
- For WCC on real-world RDF graphs, MULTISTEP is faster than a color propagation-based approach.
- Our modified atomic-free and lock-free BFS averages a traversal rate of 1.4 GTEPS (Giga traversed edges per second) over all tested networks.

6.3 Background

6.3.1 Strongly Connected Components

There are several existing serial and parallel algorithms that are used to determine the SCCs of a graph. This section will give a general overview of a number of important ones. Serial methods include Tarjan's and Kosaraju's algorithms, while parallel methods are the Forward-Backward algorithm and color propagation.

6.3.1.1 Serial Algorithms

Two common serial algorithms used for SCC detection are Tarjan's [18] and Kosaraju's [116] algorithms. Both of these algorithms perform linear $O(n + m)$ work in the RAM model of computation, where n is the number of vertices and m is the number of edges in an input graph. However, since Tarjan's algorithm only requires a single DFS as opposed to Kosaraju's two, it is often faster in practice.

6.3.1.2 Forward-Backward

The Forward-Backward (FW-BW) algorithm [19] (see Algorithm 6.1) can be described as follows. Let V denote the set of vertices in the graph, $E(V)$ the set of outgoing edges, and $E'(V)$ the set of incoming edges. Given the graph $G\,(V, E(V))$, a *pivot* vertex u is selected. This can be done either randomly or through simple heuristics. A BFS (or DFS) search is conducted starting from this vertex to determine all vertices which are reachable from u (the *forward sweep*). These vertices form the *descendant* set (D). Another BFS is performed from u, but on $G\,(V, E'(V))$. This search (the *backward sweep*) will find the set (P) of all vertices than can reach u, called the *predecessor* set. The intersection of these two sets forms an SCC ($S = D \cap P$) that has the pivot u in it. If we remove all vertices in S from the graph, we can have up to three remaining disjoint vertex sets: $(D \setminus S), (P \setminus S)$, and the *remainder* R, which is the set of vertices that we have not explored during either search from u. The FW-BW algorithm can then be recursively called on each of these three sets.

Algorithm 6.1 Forward-Backward Algorithm

```
1:  procedure FW-BW(V)
2:      if V = ∅ then
3:          return ∅
4:      Select a pivot u ∈ V
5:      D ← BFS(G(V, E(V)), u)
6:      P ← BFS(G(V, E'(V)), u)
7:      R ← (V \ (P ∪ D)
8:      S ← (P ∩ D)
9:      new task do FW-BW(D \ S)
10:     new task do FW-BW(P \ S)
11:     new task do FW-BW(R)
```

There is parallelism on two levels. As the three sets are disjoint, they can each be explored in parallel. Also, note that we do not require any vertex ordering within each set, just reachability. Therefore, each of the forward and backward searches can be easily parallelized or run concurrently. For graphs with bounded constant vertex degree, FW-BW is shown to perform $O(n \log n)$ expected case work.

A routine called *trimming* is commonly performed before executing FW-BW. The trimming procedure was initially proposed as an extension to FW-BW [117] to

remove all trivial SCCs. The procedure is quite simple: all vertices that have an in-degree or out-degree of zero (excluding self-loops) are removed. Trimming can also be performed recursively, as removing a vertex will change the effective degrees of its neighbors. In this chapter, we refer to a single iteration of trimming as just *simple* trimming and iterative trimming as *complete* trimming. This procedure is very effective in improving the performance of the FW-BW algorithm, but can be beneficial for other approaches as well.

6.3.1.3 Color Propagation

The color propagation algorithm for SCC decomposition is given in Algorithm 6.2. This algorithm is similar to FW-BW in that it uses forward and backward traversals. However, the approach is also quite different, as it uses multiple pivots in the forward phase and only looks at a subset of edges for each pivot in the backward phase.

Algorithm 6.2 Color Propagation Algorithm

```
1:  procedure ColorSCC(G(V, E))
2:      while G ≠ ∅ do
3:          for all u ∈ V do Colors(u) ← u
4:          while at least one vertex has changed colors do
5:              for all u ∈ V in parallel do
6:                  for all ⟨u, v⟩ ∈ E do
7:                      if Colors(u) > Colors(v) then
8:                          Colors(v) ← Colors(u)
9:          for all unique c ∈ Colors in parallel do
10:             V_c ← {u ∈ V : Colors(u) = c}
11:             SCV_c ← BFS(G(V_c, E'(V_c)), u)
12:             V ← (V \ SCV_c)
```

Assume that the graph vertices are numbered from 1 to n. The algorithm starts by initializing elements of the array $Colors$ to these vertex identifiers. The values are then propagated outward from each vertex in the graph, until there are no further changes to $Colors$. This effectively artitions the graph into disjoint sets. As we initialized $Colors$ to vertex identifiers, there is a unique vertex corresponding to every distinct c in $Colors$. We consider $u = c$ as the root of a new SCC, SCV_c. The set of reachable vertices in the backward sweep from u of vertices of the same color(V_c) belong to this SCV_c. We then remove all these vertices from V and

proceed to the next color/iteration. The two subroutines amenable to parallelization are the color propagation step and the backward sweep. In a graph with a very large SCC and high diameter, the color of the root vertex has to be propagated to all of the vertices in the SCC, limiting the efficiency of the color propagation step.

6.3.1.4 Other Parallel SCC Approaches

There has been other recent work aimed at improving FW-BW and color propagation. One example is the OBF algorithm [118] of Barnat et al., which, like color propagation, aims to further decompose the graph into multiple distinct partitions at every iteration. The OBF decomposition step can be performed much quicker than color propagation. However, it does not necessarily result in as many partitions. Barnat et al. implement OBF, FW-BW, and color propagation on NVIDIA GPUs [23] and demonstrate considerable speedup over equivalent CPU implementations.

More recently, Hong et al. [114] present several improvements to the FW-BW algorithm and trimming procedure by expanding trimming to find both 1-vertex and 2-vertex SCCs, decomposing the graph after the first SCC is found by partitioning based on weakly connected components, and implementing a dual-level task-based queue for the recursive step of FW-BW to improve runtimes by reducing overhead for the task-based parallelism. We present detailed comparisons to their approach in Section 6.6.

6.3.2 Connected and Weakly Connected Components

The connected components of an undirected graph are the maximal subgraphs where every vertex in the subgraph has a path to every other vertex. Weakly connected components in a directed graph are the equivalent to connected components in undirected graphs if edge directivity is ignored.

The approaches for determining connected components and weakly connected components in graphs are similar. There are two primary parallel methods, using techniques similar to those described in the preceding sections. First, a parallel BFS can be used for connected components. Any vertices reachable by the BFS traversal will be in the same component. We continue selecting new unvisited vertices as BFS roots until all vertices have been visited and all connected components identified.

The procedure is the same for weakly connected components, but it is required to examine both in and out edges.

We can also use a color propagation approach. Each vertex is initialized with a unique color, and the maximal colors are propagated throughout the network. Once the colors reach a stable point, all vertices contained in each discrete component will have the same color. The number of propagation iterations is bounded by $O(\log n)$ when using the pointer-jumping technique. There is also load balancing phase to handle giant connected components [119].

6.4 Applying the Multistep Method

This section provides more implementation details about various phases, and also discusses extensions for WCC, CC, and articulation point detection. All our algorithms were implemented in C++, using OpenMP for multithreading. We use the compressed sparse row (CSR) representation for graph storage, and use additional arrays for storing incoming edges. To avoid modifying the graph, we have a boolean array termed *valid* which signifies if a vertex is yet to be placed in an SCC. We also have an additional integer array which gives a numeric identifier of the SCC to which each vertex belongs. We avoid locking or atomic operations when possible through thread-owned queues, mitigation of race conditions, and by utilizing various techniques to reduce work.

6.4.1 Trim Step

We consider two different approaches for parallel trimming. We use a boolean array to mark vertices that are trimmed. Simple trimming requires the degrees of vertices in the original graph. Therefore, it requires a single pass through all vertices to find their in/out degrees, and flip their *valid* boolean if either degree is zero.

Complete trimming is a bit more complex. To speed up parallel complete trimming, in addition to the boolean trimmed array, we create current and future queues and an additional boolean array of values (mark) to signify if a vertex is currently placed in the future queue. All vertices are in the current queue to begin with. We then determine the effective in- and out-degrees for all vertices in the current queue and mark trimmed vertices as such. In addition, any untrimmed

child or parent of the trimmed vertex is placed in the future queue and marked as such. Once the current queue is empty, the queues are swapped with the marks reset.

This process is repeated for as many iterations as necessary. The queues avoid having to look through all vertices at each iteration, as it has been observed that long tendrils of vertices in several real-world graphs [11] tend to result in numerous iterations where only a few vertices are removed at a time. The marking is done to prevent a vertex from being placed in the future queue multiple times. To avoid the synchronization overhead that would be required with a parallel queue, we maintain separate queues for each thread and combine them into the next level queue at the end of each iteration of complete trimming.

Although complete trimming is easily parallelizable and can be quite fast, with the queues and marking being done similar to our BFS and color propagation steps (described below), it does not offset the additional cost. We note that the simple trimming step removes the vast majority of vertices that can be removed by trimming, and the additional iterative steps have a high overhead.

6.4.2 Breadth-First Search

The main subroutine in the FW-BW step is the parallel breadth-first search. We utilize a level-synchronous and hybrid bottom-up parallel approach, with threads concurrently exploring the vertices in the current frontier. Further, each thread maintains a queue of visited vertices to represent the frontier, and these queues are merged at the end of each iteration. Using thread-local queues instead of a shared queue avoids the synchronization overhead of insertions.

A key data structure required in BFS is a lookup array of size n, to check if a vertex has been visited or not. A typical BFS optimization is to use a bitmap (1 bit per vertex) to avoid further exploring visited vertices. A bitmap will fit completely in the last-level cache of modern server-grade CPUs for graphs of up to tens of millions of vertices. However, as we observed, a boolean *visited* array (one byte per vertex) actually outperforms a bitmap in our test environment. The likely reason for this is three-fold: less arithmetic to figure out the vertex index within the bitmap, the additional accesses needed for a SCC algorithm as opposed to running a pure BFS, and guaranteed atomic reads/writes at the byte level on

our test system [120]. A much more complicated read/write function is required to guarantee atomic updates for a bitmap [113]. We note that the effectiveness of a bitmap, in practice, will depend on last-level cache utilization, which is dependent on the size and structure of the network being explored.

Recent results show that for certain levels of a BFS in low-diameter graphs, it is more efficient to look in the reverse direction [112]. In this *direction-optimizing* approach to BFS, all unvisited vertices attempt to find a parent that is in the frontier, instead of the typical way of inspecting adjacencies of frontier vertices. We used this optimization with similar settings as the original paper ($\alpha = 15, \beta = 25$) and notice considerable speedup. However, we had to maintain the thread queues in the bottom-up hybrid as opposed to explicitly rebuilding the queue from scratch when we turn the hybrid mode off. This is due to the fact that we do not maintain the BFS tree and lack the ability to track BFS level on a per-vertex basis, as we only maintain the visited array for determining the SCC.

We also investigated a per-socket graph partitioning and exploration scheme similar to the ones described in Agarwal et al. [111] and Chhugani et al. [113]. Although these partitioning approaches improved parallel scaling, it was only in a limited number of instances that actual runtimes improved due to the additional overhead. We do not include it in our final results. Overall, our BFS implementation achieves an mean traversal rate of 1.4 GTEPS (billion traversed edges per second) on the graphs given in Table 6.1.

6.4.3 Color Propagation

The pseudocode for the parallel vertex color propagation step MS-Coloring is given in Algorithm 6.3. Initially, all active vertices are assigned a color which is the same as their vertex identifier and placed into a frontier queue Q. The adjacencies of all vertices in Q are inspected in parallel, and we check to see if an adjacency's color is lower than the color of v, the current vertex. If it is, the color is passed to the child, and both the parent and child are placed in the thread's next level queue Q_t and globally marked as such. Although the pseudocode of Algorithm 6.3 indicates that Q_t is of unlimited size and is only emptied at the end of each iteration, in practice it is usually better to limit Q_t to a fixed size (dependent on cache and other hardware considerations) and empty it into a global next level queue when

it becomes full. We omit showing this for clarity and space considerations of the given algorithm.

Algorithm 6.3 Pseudocode for MS-Coloring

```
1: for all v ∈ V do
2:     Color(v) ← v
3:     Add v to Q
4:     Visited(v) ← false
5: while Q ≠ ∅ do
6:     for all v ∈ Q do in parallel on thread t
7:         for all ⟨v, u⟩ ∈ E(V) do
8:             if Color(v) > Color(u) then
9:                 Color(u) ← Color(v)
10:                if Visited(u) = false then
11:                    Visited(u) ← true
12:                    Add u to Q_t
13:        if any u changed color then
14:            if Visited(v) = false then
15:                Visited(v) ← true
16:                Add v to Q_t
17:    for all v ∈ Q_t do in parallel on thread t
18:        Visited(v) ← false
19:    Barrier synchronization
20:    Q ← ∪_t Q_t                          ▷ Master thread performs merge
```

We place the parent in the next level queue after an update occurs to avoid explicit locking. It is possible that two parents will have higher colors than a shared child, creating a race condition. Both parents will once again examine their children in the next iteration to make sure that either the color that was given by them, or a higher one, has been placed. Additionally, since only a higher color can be assigned, we can ignore the race condition created if a parent has their own color overwritten before they assign their previous one to the child.

We also tried to avoid locks by trying a bottom-up scheme: having children look at their parents' and own color and take the largest, avoiding the race condition entirely. However, this is much slower in practice, because either all vertices need to be examined at each iteration, or the out vertices of the child need to be examined to create the queue, effectively doubling the amount of memory transfers for each iteration.

Our parallel SCC finding on the reverse step is fairly standard, as it is a trivial algorithm to parallelize. We simply determine the root vertices by finding all unique colors in the graph, and then run a parallel multi-source BFS using each unique root as a source. We propagate SCC labels to descendants of the roots which have the same initial color as the root.

6.4.4 Serial Step

We use the recursive Tarjan's algorithm for the serial step. Previous work has demonstrated little difference in runtime between recursive and non-recursive implementations [121]. Additionally, Tarjan's runtime should ideally be within a small factor of the runtime of a plain DFS, and our implementation runs within a factor of 1.65× on average across our test graphs in Table 6.1.

We experimentally determined that a cutoff of about 10,000 to 100,000 remaining vertices is a relatively good heuristic for switching to the serial algorithm, although this is hardware and graph-specific. Some graphs benefit from running color propagation all the way to completion, while some others would benefit more from switching to serial sooner. However, determining this cutoff without prior knowledge of the graph may be quite difficult. The cutoff threshold can be set based on the number of steps needed to fully color the graph, and we will investigate this in future work.

6.4.5 Connected Components and Weakly Connected Components

Our Multistep method can be easily extended to detect weakly connected components in directed graphs, and connected components in undirected graphs. We initially trim any vertices with an effective degree of 0, then determine the massive (weakly) connected component through a single parallel BFS from the pivot (instead of two in case of SCC), and finally perform color propagation on the remaining vertices.

6.5 Experimental Setup

Experiments were performed on *Compton*. All programs were compiled with the Intel C++ compiler, version 13.1.2. The `-O3` optimization parameter was used with the `-openmp` flag. The environment variable `KMP_AFFINITY` was used to control thread locality when needed.

For comparison to recent work, we also run SCC code provided by Hong et al. [114] and CC code from the Ligra framework, released by Shun and Blelloch [115]. We used the same compilation procedures and runtime environment when possible, with the exception of using Cilk Plus for parallelizing Ligra code instead of OpenMP. This was observed to be faster in practice.

Several large real-world and synthetic graphs were used in the course of this work (see Table 6.1). The first twelve graphs listed in the table are undirected while the rest are directed. These graphs were retrieved from a number of sources, namely the SNAP database [69], the Koblenz Network Collection [80], the 10th DIMACS Implementation Challenge [27], and the University of Florida Sparse Matrix Collection [75]. The R-MAT [87] and $G(n, p)$ networks were generated with the GTGraph [122] suite using the default parameters.

Friendster, LiveJournal, Orkut, and Twitter are crawls of social networks [70, 72, 73]. Italy Web is a web crawl of the .it domain [74]. WikiLinks is the cross-link network between articles on Wikipedia [80]. XyceTest is a Sandia National Labs electrical simulation network and Cube is 3D coupled consolidation problem of a cube discretized with tetrahedral finite elements. R-MAT_20/22/24 are R-MAT graphs of scale 20, 22, and 24, respectively. RDF_Data is constructed from RDF triples in a *data.gov* data set (# 1527), and RDF_linkedct is a semantic data set from clinical trials. Note that these RDF datasets contain no non-trivial SCCs because they are mostly bipartite and acyclic. WCC detection is a useful preprocessing step for partitioning these data sets. The Kron_21 graph is a SCALE 21 graph created from the Kronecker generator of the Graph500 benchmark [123, 124]. Finally, GNP_1 and GNP_10 refer to Erdős-Rényi random graphs with 1 and 10 large SCCs, respectively.

These graphs were selected to represent a wide mix of graph sizes and topologies. The number of SCCs/CCs and max SCC/CC both play an important role in the general performance of decomposition algorithms, while the average degree and

Network	n	m	deg		$\widetilde{D}$	(S)CCs	
			avg	max		count	max
Twitter	53M	2000M	37	780K	19	12M	41M
ItWeb	41M	1200M	28	10K	830	30M	6.8M
WikiLinks	26M	600M	23	39K	170	6.6M	19M
LiveJournal	4.8M	69M	14	20K	18	970K	3.8M
XyceTest	1.9M	8.3M	4.2	246	93	400K	1.5M
RDF_Data	1.9M	130M	70	10K	7	1.9M	1
RDF_linkedct	15M	34M	2.3	72K	13	15M	1
R-MAT_20	0.56M	8.4M	15	24K	9	210K	360K
R-MAT_22	2.1M	34M	16	60K	9	790K	1.3M
R-MAT_24	7.7M	130M	17	150K	9	3.0M	4.7M
GNP_1	10M	200M	20	49	7	1	10M
GNP_10	10M	200M	20	49	7	10	5.0M
Friendster	66M	1800M	53	5.2K	34	70	66M
Orkut	3.1M	117M	76	33K	11	1	3.1M
Cube	2.1M	62M	56	69	157	47K	2.1M
Kron_21	1.5M	91M	118	213K	8	94	1.5M

Table 6.1. Information about test networks. Columns are # vertices, # edges, average and max. degree, approximate diameter, # of (S)CCs, and size of the largest (S)CC.

graph diameter can have a large effect on the BFS subroutine that is necessarily used for these algorithms.

6.6 Experimental Results

In this section, we compare our Multistep SCC algorithm execution time and scaling to our implementations of the baseline FW-BW and color propagation algorithms, as well as the Hong et al. SCC algorithm. Furthermore, we compare our Multistep CC algorithm to baseline color propagation and the color propagation approach implemented in the Ligra graph processing framework. We then compare our weakly connected components algorithm to the color propagation-based parallel approach, and our biconnected components algorithm to the optimal serial algorithm. We justify algorithmic choices and measure their influence on parallel performance for different graphs.

Network	Execution time (s)					MS Speedup	
	Serial	MS	Hong	FW-BW	Color	Serial	All
Twitter	33.0	**1.60**	2.6	120.00	40.0	20.0×	1.6×
ItWeb	6.7	**1.80**	16.0	1400.00	7.1	3.6×	3.6×
WikiLinks	4.9	**0.90**	0.98	270.00	9.3	5.5×	1.1×
LiveJournal	1.3	**0.11**	0.20	4.10	1.6	12.0×	1.9×
XyceTest	0.2	**0.04**	0.08	0.07	0.37	4.7×	1.9×
R-MAT_24	2.4	**0.25**	**0.25**	0.62	2.4	9.5×	1.0×
GNP_1	7.2	**0.15**	0.30	1.60	6.5	47.0×	1.9×
GNP_10	5.5	2.90	5.10	**1.20**	3.5	1.9×	0.6×

Table 6.2. Comparison of serial Tarjan's algorithm with parallel MULTISTEP, Hong et al. , Naïve FW-BW, and color propagation, running on 16 cores.

6.6.1 Strongly Connected Component Decomposition

Table 6.2 gives the absolute execution time on 16 cores for baseline color propagation, FW-BW with complete trimming, MULTISTEP with simple trimming and the Hong et al. *Method 2* on several directed graphs. The fastest method for each network in highlighted in bold. We also give the speedup achieved by MULTISTEP over the serial approach and the fastest approach for that network.

Both MULTISTEP and Hong et al. are considerably faster than the parallel FW-BW and color propagation approaches. The performance of the baseline approaches is also very dependent on graph structure. The graphs with a large proportion of their vertices in the massive SCC, such as the $G(n, p)$, R-MAT, and Xyce graphs, show very poor performance with color propagation, due to the long time needed to fully propagate the colors. Further, networks with a large absolute number of SCCs show poor performance with FW-BW, due to the recursive and tasking overhead. FW-BW demonstrates the strongest performance on GNP_10, as this graph was designed to result in very even partitions for each recursive call. It should be noted that with the Hong et al. code on the GNP_10 graph, we utilize their BFS subroutine for the recursive SCC calls instead of their standard DFS, as the BFS is better suited to finding large-scale SCCs and also performs better.

Although the Hong et al. method attempts to minimize the impact of the recursive and tasking overhead with a partitioning step based on WCCs and a smart tasking queue, on graphs with a very high number of small but non-trivial SCCs, such as ItWeb, the overhead inherent in the FW-BW algorithm can still

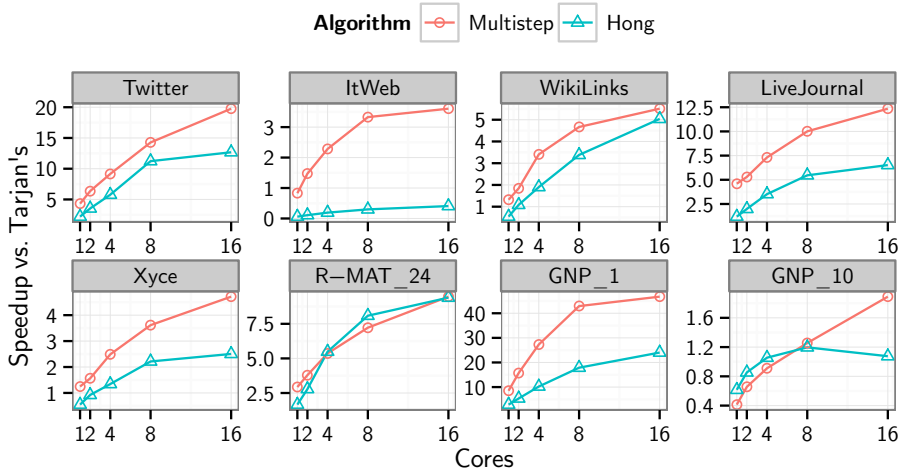

Figure 6.1. Finding SCCs: Parallel scaling of MULTISTEP and Hong et al. relative to Tarjan's serial algorithm.

dominate the running time. It can also be noted that our color propagation step will, at each iteration, partition the graph into *at least* as many discrete partitions that their WCC decomposition will. Overall, for 16-core runs, MULTISTEP gives a geometric mean speedup of 1.92× over Hong et al. on these graphs.

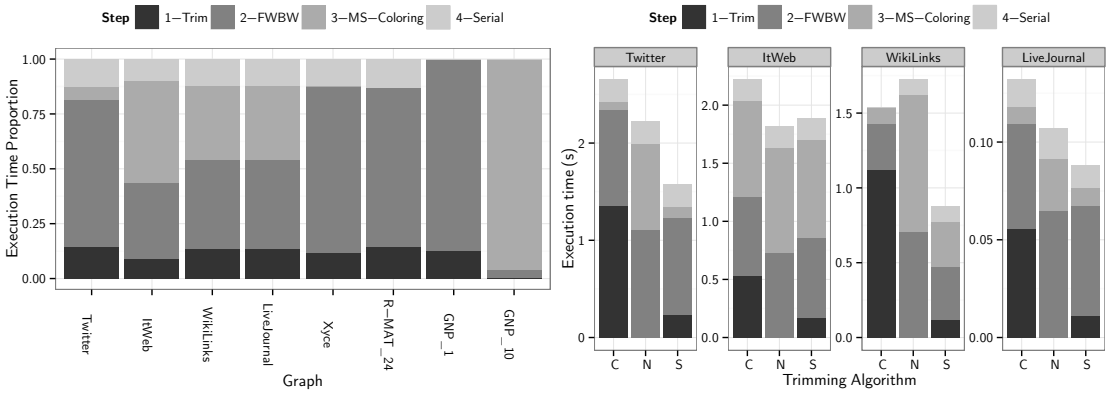

Figure 6.2. Left: Proportion of time spent in each subroutine of the MULTISTEP algorithm. Right: Comparing possible trimming procedures (S: Simple, N: None, C: Complete) in MULTISTEP for several networks.

Figure 6.1 gives the scaling of MULTISTEP and Hong et al. for parallel runs, relative to the serial Tarjan implementation. Both MULTISTEP and Hong et al. demonstrate good scaling on most test instances, and the overall speedup on 16 cores is dependent on the single-threaded performance. The Hong et al. running time on ItWeb is greatly affected by the number of SCCs. Additionally, on ItWeb,

there are long strings of trivial and non-trivial SCCs, which results in a relatively long time spent in the multiple trimming iterations that are in the Hong et al. approach, as well as longer time spent in their WCC decomposition step.

Figure 6.2 gives the breakdown for each stage of MULTISTEP as a proportion of total parallel running time. We observe that the execution time proportion for the FW-BW and color propagation steps is mostly dependent on graph structure, with color propagation taking a larger proportion of time for graphs for graphs with a higher diameter (e.g., ItWeb vs Twitter). In case of GNP_10, most of the time is spent in the serial step due to the fixed cutoff employed in our case.

Figure 6.2 also gives further justification for our choice of doing simple trimming versus complete trimming with Multistep. In general, the time spent doing iterative trimming does not sufficiently decrease the execution times of the FW-BW or color propagation steps for the overall running time to be lower. As is shown on LiveJournal and Twitter, doing no trimming at all can end up being faster than fully trimming the graph with our MULTISTEP approach. ItWeb shows that no trimming can even be faster than simple trimming, although this appears to be an exception. While running MULTISTEP across a wide variety of graphs, fully trimming the graph never improved execution times versus only doing a single iteration. However, complete trimming is important for naïve FW-BW.

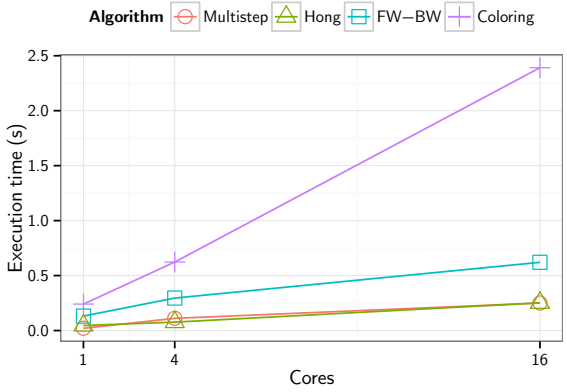

Figure 6.3. Approximate weak scaling of MULTISTEP compared to color propagation and naïve FW-BW on R-MAT graphs.

Figure 6.3 gives approximate weak scaling for three R-MAT test graphs (R-MAT_20/22/24). The test graphs' number of vertices, edges, number of SCCs, and size of largest SCC all increase by approximately a factor of 4×. From Figure 6.3, we see that MULTISTEP scales better than simple FW-BW or color propagation,

and Hong et al. performance is comparable to MULTISTEP for this instance.

6.6.2 Connected and Weakly Connected Component Decomposition

We also compare our approach to Ligra for the problem of determining connected components, for the four undirected networks in our collection. Ligra implements a parallel color propagation-based algorithm. We show scaling relative to a serial DFS. From Figure 6.4, we observe that MULTISTEP greatly outperforms the other approaches on all tested graphs.

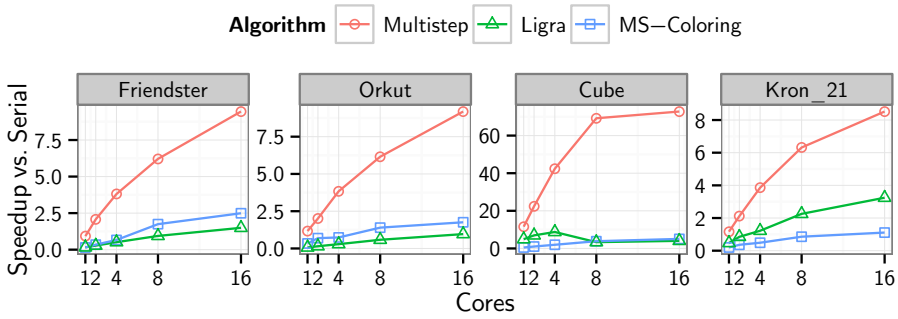

Figure 6.4. Finding CCs: Parallel scaling of MULTISTEP CC, Ligra, and MS-Coloring relative to the serial DFS approach.

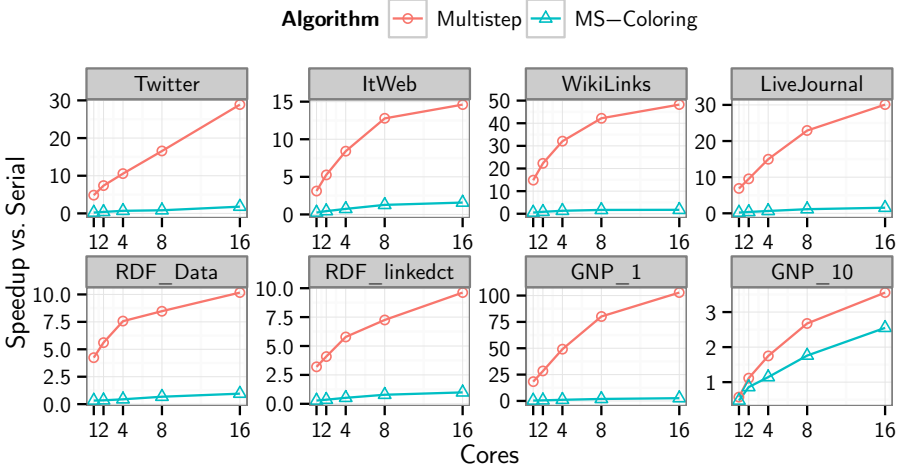

Figure 6.5. Finding WCCs: Comparison of WCC-MULTISTEP and MS-Coloring scaling relative to the serial DFS approach.

Figure 6.5 gives the speedup of the MULTISTEP method and our color propagation approach for determining the weakly connected components of several graphs.

We give speedup relative to the serial DFS approach. Once again, we observe good scaling of MULTISTEP relative to both color propagation and the serial code.

6.7 Conclusion

This chapter presented the MULTISTEP method for SCC detection and its extensions for solving related problems (CC and WCC). We demonstrate significant speedup over the current state-of-the-art methods on a multicore server and present scaling results on a wide variety of networks. The MULTISTEP method uses optimized BFS and color propagation subroutines and several heuristics to achieve this performance. Using these basic subroutines, we can solve other connectivity problems as well. In the next chapter, we will demonstrate BFS and color-propagation algorithms for finding the biconnected components of an undirected graph.

Chapter 7
Biconnectivity Algorithms for Multicore

7.1 Introduction

This chapter builds upon the parallel techniques described in the previous MULTI-STEP chapter. Using the same breadth-first search and color-propagation-based subroutines that can effectively decompose a graph into its strong, weak, and connected components, we introduce two new parallel algorithms for biconnected components decomposition of an undirected graph. Again, we show considerable performance improvements relative to prior art with these general purpose techniques and optimizations.

7.2 Biconnected Components

The biconnected component (BiCC) decomposition of an undirected graph refers to determining all *maximal biconnected subgraphs* or *blocks* contained within the graph. A biconnected subgraph is a graph which remains connected with the removal of any single vertex and all edges incident on it. *Articulation vertices* or *articulation points* are vertices that, when removed, disconnect the graph into multiple connected components. These vertices belong to two or more biconnected components. Finding the articulation vertices in a graph is one of main purposes of BiCC decomposition, as these vertices represent links that are critical for overall connectivity. A *bridge* is defined as an edge that, when removed, would disconnect

the graph into multiple connected components. BiCC decomposition also gives us a disjoint partitioning of all edges. Each edge in the graph can belong to only one maximal biconnected subgraph.

Identifying large and non-trivial biconnected components, articulation vertices, and bridges are useful in the analysis and characterization of new graph data. In the field of networking, when designing communication networks and physical infrastructure networks, identifying articulation points and minimizing bridges is relevant to network robustness and redundancy. In large virtual networks such as social networks and web crawls, BiCC decomposition gives insight into network structure and has potential to be a useful preprocessing step in data analysis [13,14].

7.2.1 Contributions

This chapter introduces two new shared-memory parallel approaches for finding the biconnected components of large sparse graphs. Both approaches use a breadth-first spanning tree. The first method is based on executing multiple truncated breadth-first searches (BFSes). The second method uses the *color propagation* [15] technique. We used this strategy in the prior chapter for detecting strongly connected components in parallel. Similar to previous serial and parallel algorithms, the output of both our algorithms is a labeling of edges into disjoint biconnected components and a classification of vertices into articulation and non-articulation points.

We analyze and implement both algorithms and perform an experimental study on an Intel multicore platform. Both approaches demonstrate good parallel strong scaling across a wide range of real-world and synthetic test cases, with the truncated BFS-based approach (the first method) offering the best speedups relative to a serial implementation. Additionally, due to the fact that the first method uses the hybrid direction-optimizing BFS algorithm of Beamer et al. [112] as a subroutine, a substantial fraction of the graph edges are pruned or untouched in some cases. This results in the single-threaded performance of our new approach being faster than the linear-work Hopcroft-Tarjan DFS-based algorithm.

7.3 Background

There is substantial prior work on serial and parallel algorithms for BiCC decomposition. We review three algorithms that we use for comparison and discuss relevant implementation details in this section.

7.3.1 Hopcroft-Tarjan Algorithm

The serial algorithm for BiCC decomposition, introduced by Hopcroft and Tarjan [17], is optimal in the RAM model and based on a single Depth-First search (DFS). It runs in $O(n+m)$ time (n is the number of vertices and m the number of edges) and linear space. A recursive implementation to identify articulation points is given in Algorithms 7.1 and 7.2.

Algorithm 7.1 Hopcroft-Tarjan biconnectivity algorithm to identify articulation points.

```
1: procedure HT(G(V, E))
2:     for all v ∈ V do
3:         preorder(v) ← −1
4:         low(v) ← −1
5:         articulation(v) ← false
6:     global count ← 0
7:     root ←selectRandomRoot(V)
8:     HT-DFS(G, root, root)
```

The algorithm maintains two arrays of size n in addition to the standard DFS stack of visited vertices. One array, the *preorder*, records the order in which vertices are discovered in the DFS. The second array *low* tracks the lowest DFS depth of the adjacencies or children of the current vertex. When there are no more adjacencies remaining to be explored from the vertex on top of the stack, this vertex is removed. If this vertex does not connect to any vertices lower on the stack other than the vertex immediately preceding it on the stack, then we know that this vertex is an articulation point and can mark it as such.

The listing is only for identifying articulation points. If we would like to partition edges into components, we need to maintain an additional stack. As each edge is first touched, it is placed on this stack. When an articulation point is identified, all of the edges contained in the biconnected component can then be removed from

Algorithm 7.2 Recursive DFS used in Hopcroft-Tarjan algorithm.

```
1: procedure HT-DFS(G, u, v)
2:     children ← 0
3:     preorder(v) ← count++
4:     low(v) ← preorder(v)
5:     for all ⟨w, v⟩ ∈ E do
6:         if preorder(w) = −1 then
7:             children ← children + 1
8:             HT-DFS(G, v, w)
9:             low(v) ←min(low(v), low(w))
10:            if low(w) ≥ preorder(v) and u ≠ v then
11:                articulation(v) ← true
12:            else if w ≠ u then
13:                low(v) ←min(low(v), preorder(w))
```

the stack and appropriately labeled. Our serial implementation is closely based on this recursive DFS-based algorithm. We will refer to this as the HT method.

7.3.2 Tarjan-Vishkin Parallel Algorithm

The PRAM BiCC algorithm by Tarjan and Vishkin [20] requires $O(\log n)$ time using $O(n+m)$ processors. An overview of the main steps is given by Algorithm 7.3.

Algorithm 7.3 Tarjan-Vishkin algorithm to identify articulation points.

```
1: procedure TV-BiCC(G(V, E))
2:     T ← SpanningTree(G)
3:     L ← EulerTour(T)
4:     pre, size ← ListRank(L)
5:     low, high ← DetermineMinMaxPreorder(G, T, pre, size)
6:     G′ ← BuildAux(G, T, low, high)
7:     C ← ConnectedComponents(G′)
8:     B ← BiconnectedComponents(C, G)
```

The Tarjan-Vishkin algorithm has several key subroutines. First, a spanning tree T is created from the input graph G. This can be computed using any traversal. T is then rooted at an arbitrary vertex and a Euler tour is found to create an ordered list of vertices L. List ranking is then performed on L, which in turn gives the preordering numbering of vertices $pre(v)$ for all $v \in T$. The size of each subtree rooted at v in T, $size(v)$, is also found.

Similar to the serial algorithm, two values are then obtained for each vertex $v \in G$, a *low* and a *high* value. Using the preorder numbering from the previous step, we determine the *low* and *high* values as the lowest and highest preorder numbering of all descendants of or neighbors of descendants of v that are not connected in T. The next steps utilize these labelings to build an auxiliary graph G', whose connected components C are the biconnected components of G. The problem of finding biconnected components in the original graph is recast as finding connected components in this auxiliary graph. A final step can then be performed to create an explicit set B of the edges and articulation points defining each biconnected component in G.

7.3.2.1 Cong-Bader TV-Filter Algorithm

An experimental study by Cong and Bader [125] presents an improvement to the Tarjan-Vishkin algorithm that leads to a significant reduction in the size of the auxiliary graph. They suggest a preprocessing step that filters out certain *non-essential* edges (i.e., edges that do not impact the biconnectivity of G). The use of this preprocessing step also reduces execution time of other subroutines in the Tarjan-Vishkin algorithm for the graph instances studied in [125]. We also independently notice up to a 4× speedup over the Tarjan-Vishkin (TV) algorithm, and hence we just focus on TV-Filter in this chapter.

Note that this preprocessing step is similar to the approach for finding k-connected subgraphs in a j-connected graph, where $j > k$, as described by Nagamochi and Ibaraki [126]. However, it is important to note that the filtering step finds only a k-connected subgraph in a k-connected graph, specifically where $k = 1$, by constructing $k + 1$ spanning forests (in this case, a spanning tree and then a spanning forest of the graph with that spanning tree removed). As a result, this preprocessing step preserves the biconnectivity of the original graph by maintaining articulation points, bridges, and the biconnected components themselves.

Algorithm 7.4 provides an overview of Cong and Bader's approach. A BFS is first performed from an arbitrary vertex to obtain T (the spanning tree) and P (parent information). Edges in T are then filtered from G. A spanning forest F is obtained using another traversal and by executing connected components. The union of edges in F and T is shown to contain the essential edges needed to determine the biconnected components B using the standard Tarjan-Vishkin

Algorithm 7.4 Cong-Bader algorithm to identify articulation points.

```
1: procedure CB-BiCC(G(V, E))
2:     T, P ← BFS(G)
3:     F ← SpanningForest(G \ T)
4:     B ← TV-BiCC(F ∪ T)
5:     for all e = ⟨u, v⟩ ∈ G − (F ∪ T) do
6:         label e as in BiCC containing v and P(v)
7:         B ← (B ∪ e)
```

algorithm. The rest of the edges are non-essential. Once the Tarjan-Vishkin algorithm completes, the non-essential edges are labeled and added back to obtain the complete BiCC output. Correctness proofs and the data structures used to store the intermediate results are given in [125].

In this work, we use an updated version of the code first developed by Cong and Bader for their study [127], which was designed using the SIMPLE POSIX threads-based framework [128]. We changed their code for execution on our test platform, primarily modifying storage of common structures and eliminating or globalizing some thread-owned structures in an effort to reduce memory utilization. Despite these changes, memory usage when creating the auxiliary graph limits running the code on our two largest test instances.

7.3.3 Related Work

There are several other known parallel algorithms for BiCC. One of the earliest CREW PRAM parallel algorithms was presented by Eckstein [129]. This approach runs in $O(d \log^2 n)$ time with $O((n+m)/d)$ processors on the CREW PRAM model. This work was the first to note that the structure of BFS trees can be utilized to find articulation points, and our work can be considered a related extension of that work for modern architectures. Savage and JáJá [130] designed two PRAM algorithms, taking $O(\log^2 n)$ and $O(\log^2 n \log k)$ time and requiring $O(n^3/\log n)$ and $O(mn+n^2 \log n)$ processors, respectively, where k is the number of biconnected components. A CREW PRAM algorithm by Tsin and Chin [131] runs in $O(\log^2 n)$ time with $O(n^2/\log^2 n)$ processors.

Another class of proposed biconnectivity algorithms utilize open ear decompositions. An original approach was described by Maon et al. [132]. This approach was expanded upon by Miller and Ramachandran [133], with a related implementation

for solving 2-edge connectivity problems on an early massively parallel MasPar system [134].

In terms of parallel implementations, the Tarjan-Vishkin approach has received the most attention, with Cong and Bader being the first to demonstrate parallel speedup on an SMP system over the serial HT algorithm with their TV-Filter algorithm. Edwards and Vishkin [110] also implemented the TV and HT algorithms using a new programming model, and demonstrated scalability and parallel speedup on the Explicit Multi-Threading (XMT) manycore computing platform.

Most recently, Ausiello et al. developed a MapReduce-based biconnected components detection algorithm [135, 136] under a streaming data model. This method is based on previous work by Westbrook and Tarjan [137], but utilizes a lightweight *navigational sketch* of the input graph to hold biconnectivity information for the full graph.

7.4 New Parallel Algorithms

We present two new methods in this section that are both based on a breadth-first spanning tree. However, unlike TV and TV-Filter, we do not construct an auxiliary graph. Instead, we identify some properties that articulation vertices must satisfy and use them to decompose the graph. Our two methods use BFS and color propagation as underlying subroutines, and so we refer to them as BiCC-BFS and BiCC-Coloring. The output obtained is identical to prior algorithms. In both the methods, we initially assign two integer labels to each vertex and progressively update them. The final vertex labels of all vertices can be inspected to determine the component to which an edge belongs to. These algorithms assume that we begin with an undirected graph with a single connected component.

7.4.1 BFS-based BiCC method

We first describe the BFS-based approach to identify articulation points, similar to that used by Eckstein [129]. The key steps are listed in Algorithms 7.5 and 7.6. Assume that G has only one connected component with no multi-edges and self loops. Choose an arbitrary vertex r, designate it as the root, and perform a BFS. Store the BFS output in two arrays, P and L. For all $v \in V$, $P(v)$ stores a parent

of v (i.e, the BFS tree) and $L(v)$ gives the depth of v in the BFS tree, or the distance of v from r. For all $v \in V$, we define a *child* w of v to be any adjacency of v such that $P(w) = v$. Thus, the set of all children of a vertex is a subset of its adjacencies. Our algorithm is based on the following proposition.

Proposition 7.4.1. *A non-root vertex v in the BFS tree $\langle P, L\rangle$ is an articulation vertex if and only if it has at least one child w that cannot reach any vertex of depth at least $L(v)$ when v is removed from G.*

The above statement is equivalent to the following.

Proposition 7.4.2. *A non-root vertex v in the BFS tree $\langle P, L\rangle$ is not an articulation vertex if and only if all its children w in the BFS tree ($P(w) = v$) can reach all other vertices in the graph G when v is removed from G.*

Proof. An articulation vertex, by definition, is a vertex whose removal will decompose G into two or more connected components. Consider a vertex v and all its adjacencies. The BFS output splits the adjacencies of any non-root vertex into three disjoint subsets: its children (as defined above), non-child adjacencies at depth $L(v)+1$, and adjacencies at depth $L(v)$ or $L(v)-1$. If v were an articulation point, then the adjacencies of v would be split up such that at least two of them lie in different connected components upon removal of v. If v is not an articulation point, there must be an alternative path in the graph from adjacencies of v to every other vertex, and thus, between every pair of adjacencies of v as well. To show that v is an articulation point, it suffices to inspect just the children of v in the BFS tree and show that at least one of them cannot reach an alternate vertex at the same level as v (and thus is disconnected from some part of the graph). Non-child adjacencies of v have a path through their parent in the BFS tree to other vertices, and so we do not need to consider them explicitly. If a child can reach some vertex at the same level as v, then it can reach all other vertices by tracing a path back to the root. □

As a more general extension of the above propositions, we have the following Corollary.

Corollary 7.4.3. *If a traversal from any $u_i \in V$ ($P(u_i) = v$) is not able to reach* all *other $u_j \in G$ ($P(u_j) = v$) when v is removed from the graph, then v is an*

articulation point. Further, if the only path in G between u_i and u_j requires v, then u_i and u_j are in separate biconnected components with v as an articulation point. We term v as the parent articulation vertex.

Algorithm 7.5 BFS-based algorithm to identify articulation points in BiCC decomposition.

```
1:  procedure BFS-ArtPts(G(V, E))
2:      for all v ∈ V do
3:          Art(v) ← false
4:          visited(v) ← false
5:      Select a root vertex r
6:      P, L ← BFS(G, r)
7:      for all u(≠ r) ∈ V where P(u) ≠ r do
8:          v ← P(u)
9:          if Art(v) = false then
10:             l ←BFS-L(G, L, v, u, visited)
11:             if l ≥ L(v) then
12:                 Art(v) ← true
13:     Check if r is an articulation point
```

Algorithm 7.6 Truncated BFS subroutine in the BFS-ArtPts algorithm.

```
1:  procedure BFS-L(G(V, E), L, v, u, visited)
2:      Insert u into Q
3:      Insert u, v into S
4:      visited(u) ← true
5:      visited(v) ← true
6:      while Q ≠ ∅ do
7:          for all x ∈ Q do
8:              Remove x from Q
9:              for all ⟨w, x⟩ ∈ E where visited(w) = false do
10:                 if L(w) < L(u) then
11:                     for all s ∈ S do visited(s) ← false
12:                     return L(w)
13:                 else
14:                     Insert w into Q
15:                     visited(w) ← true
16:     return L(u)
```

We now describe the algorithm in more detail. Parallelization of the initial

BFS computation required to construct P and L is well-studied. We use the parallel BFS from our prior work [21]. This implementation maintains a boolean vector for tracking visited vertices, thread-local queues, and further utilizes a direction-optimizing search [112]. These optimizations have been demonstrated to considerably speed up parallel BFS computations on the small-world graphs we are considering.

To identify the articulation points, we consider every vertex u and its parent $v = P(u)$. Instead of performing pairwise reachability queries from u, we perform a BFS from u after removing v from G and track the level of vertices that are reached, using the previously computed L values. If any vertex w with level $L(v)$ or less is reached, we can exit. This step (i.e., step 7 of Algorithm 7.5) can also be parallelized, with each thread maintaining a separate *visited* bit vector. We also use another temporary stack S in the BFS-L subroutine to track visited vertices.

The root vertex r must be handled separately. There are several ways to determine whether it's an articulation point. The simplest way is to select a vertex that is definitely not an articulation point (vertex of degree 1), or is certainly an articulation point (a degree-2 vertex, with one of its neighbors having a degree of one). If no such vertices exist, then there are two options. One is to create a new spanning tree using an alternate root and run the second stage of the algorithm with any children u of the original root r, $P(u) = r$. Because our optimized BFS subroutine is quite fast, this is not entirely impractical. The second option is to run a BFS on $G \setminus r$ from a single u where $P(u) = r$ and track whether all other w where $P(w) = r$ are also reachable. If they are all reachable from u, then r is not an articulation vertex as per Corollary 7.4.3.

7.4.1.1 Identifying Biconnected Subgraphs

We now extend the previous algorithm and make it work-efficient to label edges. The new method is given in Algorithms 7.7 and 7.8.

The primary goal of this approach is to determine for all $v \in V$ two labels, $Par(v)$ and $Low(v)$. The Par value is the highest-level articulation point (parent articulation vertex) separating v from the root. Low signifies the lowest-value vertex identifier (vertices are numbered from 0 to $n - 1$) among all vertices contained in the biconnected component of v. We can then use these two vertex labels to uniquely label all edges.

Algorithm 7.7 BFS-based algorithm to perform BiCC decomposition.

```
1: procedure BFS-BiCC(G(V, E))
2:     for all v ∈ V do
3:         Art(v) ← false
4:         visited(v) ← false
5:         Low(v) ← v
6:         Par(v) ← v
7:     Select a root vertex r
8:     P, L, LQ ← BFS(G, r)
9:     for all Q_i ∈ LQ_{m···1} do
10:        for all u ∈ Q_i, where Par(u) = u do
11:            Remove u from Q_i
12:            v ← P(u)
13:            l, vid_low, V_u ←BFS-LV(G, L, v, u, visited)
14:            if l ≥ L(u) then
15:                Art(v) ← true
16:                visited(v) ← false
17:                for all w ∈ V_u do
18:                    Low(w) ← vid_low
19:                    Par(w) ← v
20:                    visited(w) ← false
21:                    Remove w from V
22:    for all e = ⟨u, v⟩ ∈ E do
23:        if Low(v) = Low(u) or Par(u) = v then
24:            BiCC(e) ← low(u)
25:        else
26:            BiCC(e) ← low(v)
```

The first step in Algorithm 7.7 is similar to Algorithm 7.5. A BFS is performed in order to determine the level $L(v)$ and parent $P(v)$ for each vertex v in V. Additionally, we store the output of the level-synchronous BFS as a list of queues LQ. Each $Q_i \in LQ$ contains all the vertices at a distance i from the root vertex. We inspect the queues in reverse order, from maximum level m through level 1 (level 0 is considered as the level containing the root). If u has already been assigned a Par value, we know that the biconnected component that contains u has already been discovered via another child of $P(u)$. Otherwise, we perform a similar BFS as in Algorithm 7.5. However, in this search, we also track the lowest vertex identifier encountered, vid_{low}, as well as a list of all unique vertices encountered, V_u.

If we determine that v is an articulation vertex based on the retrieved minimum

Algorithm 7.8 Truncated BFS subroutine in BFS-BiCC to identify articulation points and track component vertex set.

```
1: procedure BFS-LV(G(V, E), L, v, u, visited)
2:     Insert u into Q
3:     Insert u into V_u
4:     visited(u) ← true
5:     visited(v) ← true
6:     vid_low ← u
7:     while Q ≠ ∅ do
8:         for all x ∈ Q do
9:             Remove x from Q
10:            for all ⟨w, x⟩ ∈ E where visited(w) = false do
11:                if L(w) < L(u) then
12:                    return (L(w), ∅, ∅)
13:                else
14:                    Insert w into Q
15:                    Insert w into V_u
16:                    visited(w) ← true
17:                    if w < vid_low then
18:                        vid_low ← w
19:    return (L(u), vid_low, V_u)
```

level l, we then proceed to label all Par and Low of all encountered vertices w. $Par(w)$ is set to point to the BiCC parent articulation vertex of v, while $Low(w)$ is set to vid_{low}. This ensures a unique and consistent labeling across vertices within the components. Note that it is not necessary to create the Par values, since simply tracking the articulation vertices is sufficient to correctly label edges. We choose a Par array as opposed to a boolean array or vertex list in order to be consistent with the output of prior BiCC algorithms.

Once a component is identified, V_u, which is the discovered component minus the articulation vertex, is considered removed from G (line 21 of Algorithm 7.7). We do not modify G. Instead, we maintain $valid$, a shared boolean array of size n that signifies the current state of a vertex. When we remove a v from G, we set $valid(v) = false$. Because we are working from the highest-level leaves of the tree to the root in T, we can safely do this. On any given level i, all articulation vertices on level $i - 1$ will be discovered. This is because separate biconnected components cannot exist through articulation vertices by Corollary 7.4.3, and all such articulation vertices will be discovered by Proposition 7.4.1. It is also

guaranteed that no vertices in the biconnected component will have been previously removed, as there is no way they could have been encountered by a successful articulation vertex search.

The final step is to label all edges between all u and v. Although it is possible to do this in the inner BFS loop, it is simpler and more cache-friendly to do the separate step. For labeling, if both vertices have the same *Low* value, we know they were discovered during the same search, and therefore the edge is contained in their component. If the edge is between a child and its parent articulation point, we apply the *Low* label of the child. We don't further explicitly label vertices as belonging to a component, since articulation vertices exist in multiple components, and the information is readily retrievable by examining *Low* and *Par* arrays.

7.4.1.2 Parallelization

Our primary avenue for parallelization is across all vertices in the current queue level (line 9 in Algorithm 7.7). We could also parallelize the BFS-LV search (line 13 or Algorithm 7.8). However, in our implementation, the BFS-LV is only parallelized once we reach level 1, since we observe that in in most real-world graphs that have a massive biconnected component, a randomly-selected root vertex is likely to be contained in the giant component. We do not require additional synchronization while updating the *Low* and *Par* values. Should the parent be an articulation vertex and the children be contained in the same component, the same *Par* and *Low* values will subsequently be given to all vertices in the component. Thus, all the concurrent writes are benign races.

7.4.1.3 Algorithm Analysis

The dominant step in the algorithm is the number of invocations of BFS-LV and the cumulative number of edge inspections performed through BFS-LV, with the naïve approach requiring an upper bound of $O(nm)$ work. The rest of the steps (initialization, initial BFS, *Low* and *Par* labeling) require $\theta(n + m)$ work. The naïve approach to determine articulation points is somewhat inefficient, since there is no ordering imposed on invocations of BFS-L. However, in the full algorithm, we inspect vertices in level-synchronous order, and once a biconnected component is identified, all visited vertices are marked as invalid. Thus, there are no further

unnecessary traversals. We also truncate BFS-LV as soon as we encounter a vertex at the level of the parent. We observe that the cumulative number of edge examinations is a small constant multiplicative factor of the total number of edges, and so the work performed is linear in practice. The level-synchronous approach of examining the vertices implies that the parallel time would be proportional to the the graph diameter.

7.4.2 Coloring-based BiCC Method

Instead of potentially performing a full BFS from each vertex in T, it is also possible to compute the same Par and Low values using the *color propagation* technique. Color propagation is an iterative strategy that is similar to recursive doubling, and we have previously used it to develop practical parallel algorithms for connected components in undirected graphs as well as the weakly and strongly connected components in directed graphs [21].

We define the *lowest common ancestor* (LCA) p of any two neighboring vertices u and v in a BFS tree $T\langle L, P\rangle$ to be the lowest-level vertex that both vertices share some ancestral relationship with in T. Should a parent-child relationship exist between these two vertices, $P(v) = u$ or $P(u) = v$, then the lowest common ancestor is simply the parent vertex.

Our Coloring-based approach is based on the following observation for biconnected components with at least three vertices.

Proposition 7.4.4. *In a biconnected component with at least three vertices, determine the LCA of all pairs of neighboring vertices. At least two vertices in the component will have their lowest-level LCA set to the parent articulation point.*

Proof. Any biconnected component containing at least three vertices has the requirement that the articulation vertex has at least two children. If the articulation vertex has only one child, then the removal of this child would disconnect the component and the component is therefore not biconnected. Additionally, the lowest-level LCA for each of the these two child vertices will always be articulation vertex, as per Proposition 7.4.1. It should also be noted that for any biconnected component of size larger than three, it is highly likely that two neighboring vertices of higher levels connected through an edge not in T have each of their lowest level mutual parents set as the parent articulation vertex as well. □

7.4.2.1 Identifying BiCC with Color Propagation

Our approach for determining biconnected components is given by Algorithm 7.9. This algorithm once again determines the same Par and Low values as the BFS approach. However, instead of propagating the values to vertices within the same BiCC through a search, we simply propagate them to their neighbors one iteration at a time under certain constraints.

The initialization steps are the same as before. We select a root and perform the BFS to create the parent and level arrays. We use this information to initialize the Par values for each vertex v to the lowest-level LCA among it and all of its neighbors $\langle u, v\rangle \in E$. We omit pseudocode for LCA, as it is a well-known algorithm.

Once initialization is complete, we begin our primary coloring loop. The goal of coloring is to color all vertices in a biconnected component, $v \in B$ under a parent articulation vertex of p, with $Par(v) = p$. Additionally, we want to color all $Low(v)$ as the vertex in B with the lowest vertex identifier.

A Par value is propagated from a vertex v to a neighbor u if the level of $Par(v)$ is less than the level of $Par(u)$. To ensure that no Par value is passed down from an articulation vertex to its child, we don't propagate Par values from a parent to a child unless the Par value of the child is not equal to the parent (i.e., we know there is a path from the child to a vertex of lower level than the parent, so by Proposition 7.4.1, we know the parent is in the same biconnected component as its child). We only propagate Low values between vertices that have the same Par value, as the same Par value already indicates that they are in the biconnected component.

We know that in any non-trivial biconnected component there must be at least two vertices with their Par value initialized to the parent articulation vertex for the component. These lowest-level Par values will freely propagate to all vertices in the biconnected component, with the exception of vertices not initialized to a Par besides their parents, which are vertices that have no immediate non-tree connection to a vertex of a lower level. However, these vertices do have a path to their grandparent vertex (which may be the parent articulation vertex) through either a vertex at the same level, non-tree edge (which also must be a child of the same parent) or one of their children. Using a recursive argument along this path, the directional restriction we have on Par value propagation will eventually be lifted as this lower level Par will finally reach the original vertex. When this

Algorithm 7.9 Color propagation-based algorithm to perform BiCC decomposition.

```
1: procedure COLOR-BICC(G(V, E))
2:     for all v ∈ V do
3:         Art(v) ← false
4:         Low(v) ← v
5:         Par(v) ← v
6:         Insert v into Q
7:         queued ← true
8:     Select a root vertex r
9:     P, L ← BFS(G, r)
10:    Init-LCA(G, P, L, Par)
11:    while Q ≠ ∅ do
12:        for all v ∈ Q do
13:            Remove v from Q
14:            for all ⟨u, v⟩ ∈ E do
15:                if Par(u) = v then
16:                    continue
17:                if L(Par(v)) > L(Par(u)) then
18:                    Par(u) ← Par(v)
19:                    if queued(u) = false then
20:                        queued(u) ← true
21:                        Insert u into Q
22:                if Par(v) = Par(u) then
23:                    if Low(v) < Low(u) then
24:                        Low(u) ← Low(v)
25:                        if queued(u) = false then
26:                            queued(u) ← true
27:                            Insert u into Q
28:            if any u got queued and queued(v) = false then
29:                queued(v) ← true
30:                Insert v into Q
31:        for all v ∈ Q do
32:            queued(u) ← false
33:    for all e = ⟨u, v⟩ ∈ E do
34:        if Low(v) = Low(u) or Par(u) = v then
35:            BiCC(e) ← low(u)
36:        else
37:            BiCC(e) ← low(v)
```

Algorithm 7.10 Initialize the LCA for all neighbors using parents and level information.

```
1: procedure INIT-LCA(G(V, E), P, L, Par)
2:     for all v ∈ V do
3:         for all ⟨u, v⟩ ∈ E do
4:             w = LCA(v, u, G, P, L)
5:             if L(w) ≤ L(Par(v)) then
6:                 Par(v) = w
```

happens, the lowest-level parent articulation vertex Par will be able to reach this vertex as well through any path. We also know that if the Par value reaches all vertices in the component, then so must the correct Low value, as it is unique and will begin propagating immediately when the vertex which has the Low vertex identifier value gets their Par set, or initialized to the final correct value.

We rely on a queue Q to avoid having to examine every vertex at every iteration. We further rely on a *pushing* as opposed to *pulling* form of coloring. While visiting vertex v, we examine and overwrite the colors of all its neighbors, u. With a pulling methodology, we would only overwrite the color of v with the best color from all of u. We empirically find pushing to be faster. We mitigate the race condition created by two vertices both overwriting the color of u by adding both v and u to the queue. If v had attempted to push the superior color but got it overwritten, on the next iteration v will attempt to push it again and succeed.

7.4.2.2 Parallelization

As with the BFS-BiCC algorithm, parallelization of the first stage of the algorithm is straightforward as it utilizes a standard BFS. Parallelization of the coloring stage is a bit more involved, but still relatively simple to implement. We parallelize over the queue (line 12 in Algorithm 7.9), with each thread examining and propagating colors from a limited subset of vertices. To avoid the overhead associated with writing to a shared queue, we instead have each thread place their vertices in a thread-owned queue. Once a thread completes its iteration, its queue is copied into the global queue for the next iteration. As these operations are non-blocking, there is minimal overhead. Like the previous approach, the final labeling step over all edges is easy to parallelize.

7.4.2.3 Algorithm Analysis

The key step in Color-BiCC is the coloring phase, and the work performed depends on the number of times a vertex is inserted into the queue and cumulative number of edges inspected. The rest of the steps can be performed in $\theta(n+m)$ work. The upper bound on work for coloring-based connected components and strongly connected components algorithms is $O(n^2)$ [15], but the observed performance is linear in the number of edges for real-world, low-diameter graph instances [21, 118]. Because we impose additional constraints in this case for BiCC and use precomputed LCA information to direct the color propagation, the work performed is input-dependent. We quantify the overhead of coloring through the additional edge inspections required (a multiplicative factor) over the baseline value of m. We report this value for all the test instances in the next section. Note that concurrency depends on the size of the queue for each iteration, and is not dependent on graph diameter.

7.5 Experimental Setup

Experiments were performed on *Compton*. OpenMP was used for parallelization. Several large real-world graphs were used in our study. These are listed in Table 7.1. These graphs were retrieved from a number of sources, namely the SNAP database [69], the Koblenz Network Collection [80], the 10th DIMACS Implementation Challenge [27], and the University of Florida Sparse Matrix Collection [75]. The R-MAT [87] and $G(n, p)$ (GNP) networks were generated with the GTGraph [122] suite using the default parameters. Only the largest connected component for each graph was taken. Directed edges were considered undirected. Multiple edges and self loops were removed. This was done to reduce noise in results and simplify analysis and comparison between tested algorithms.

Friendster, LiveJournal, and Orkut are social networks [70, 72, 73]. WikiLinks is the cross-link network between articles on Wikipedia [138]. Cube is 3D coupled consolidation problem of a cube discretized with tetrahedral finite elements [139]. R-MAT_24 is an R-MAT graph of scale 24. The Kron_21 graph is a scale 21 graph created from the Kronecker generator of the Graph500 benchmark [124]. Finally, GNP_1 and GNP_10 refer to Erdős-Rényi random $G(n, p)$ graphs with 1 and 10 large biconnected components, respectively. The GNP_10 network was

Network	n	m	d_{avg}	d_{max}	$\widetilde{dia}$	# Bi	Max Bi
LiveJournal	4.8M	43M	18	20K	21	1.1M	3.7M
Orkut	3.1M	117M	76	33K	11	68K	3.0M
WikiLinks	26.0M	543M	42	4.3M	86	3.5M	22M
ItWeb	41.0M	1.0B	50	1.3M	46	5.0M	33M
Friendster	63.0M	1.6B	53	5.2K	34	13M	49M
Cube	2.1M	59M	56	67	157	1	2.1M
Kron_21	1.5M	91M	118	214K	8	238K	1.3M
R-MAT_24	7.7M	133M	35	257K	11	2.2M	5.4M
GNP_1	10.0M	200M	40	152	7	1	10M
GNP_10	10.0M	200M	40	80	19	19	5M

Table 7.1. Network sizes and parameters for all networks. The columns are #vertices, #edges, average and max-degree, approximate diameter, # of BiCCs and size of the largest BiCC.

created to have 10 large biconnected components connected through 9 bridges. The components were generated independently with geometrically decreasing sizes (5.0M, 2.5M, 1.25M, ...) with the bridges added manually by connecting the independent components via single edges.

7.6 Results

We compare the performance of our two new algorithms to Cong and Bader's improvement to the Tarjan-Vishkin algorithm, as well as the Hopcoft-Tarjan serial algorithm. We will demonstrate absolute speedups relative to prior work, strong scaling of our algorithms, an analysis of each of our algorithms with respect to the basic graph computations on which they are based, as well as a more general analysis of the biconnected component size and count distribution between real world and synthetic graphs.

7.6.1 Execution times and Scaling

Table 7.2 gives the serial execution time of the Hopcroft-Tarjan algorithm, the parallel times of our coloring and BFS algorithms with 32 threads, the TV-Filter implementation run with 32 threads, as well as our best speedup over TV-Filter. All reported times are the averages over five independent tests. We select the highest

Network	HT	TV-Filter	BFS-BiCC	Color-BiCC	Speedup vs. TV-F
LiveJournal	2.1	1.6	**0.38**	0.61	4.2
Orkut	3.4	1.8	**0.49**	0.93	3.6
WikiLinks	25	24	**7.0**	20	3.4
ItWeb	19	–	50	**3.3**	–
Friendster	79	–	**20**	48	–
Cube	1.2	0.64	**0.17**	0.51	3.8
Kron_21	1.8	2.3	**0.60**	2.2	3.8
R-MAT_24	4.7	5.8	**1.5**	5.6	3.9
GNP_1	11	5.8	5.0	**4.8**	1.2
GNP_10	6.5	5.9	12	**4.0**	1.5

Table 7.2. Execution time (seconds) result comparison between the serial Hopcroft-Tarjan algorithm, TV-Filter algorithm on 32 threads, and the new BFS-BiCC and Color-BiCC approaches on 32 threads.

out-degree vertex as the root for reasons that will be noted. The fastest time over all approaches is given in bold. As previously mentioned, memory limitations for the TV-Filter algorithm prevented us from obtaining time results on the two largest graphs, ItWeb and Friendster.

From Table 7.2, it is apparent that the BFS-based algorithm demonstrates the fastest absolute execution times for a majority of test cases. Our coloring algorithm demonstrates the best performance on the remaining test cases. Both of our algorithms demonstrate considerable speedup over both the serial algorithm as well as the TV-Filter algorithm. As will be explained, the relative performance benefit between the BFS and coloring algorithms is highly dependent on graph structure.

Figure 7.1 gives the strong scaling of the BFS, coloring, and TV-Filter algorithms from 1 to 32 threads on all 10 test graphs. Reported speedups are relative to the execution time of Hopcroft-Tarjan. Similar results as Table 7.2 can be observed from Figure 7.1, with the BFS algorithm demonstrating the fastest execution time and best speedups on a majority of tests.

7.6.2 Breadth-First Search Analysis

Table 7.3 gives the execution time of the BFS-BiCC algorithm with 1, 16, and 32 threads. Additionally, the running time of just a single BFS exploration of

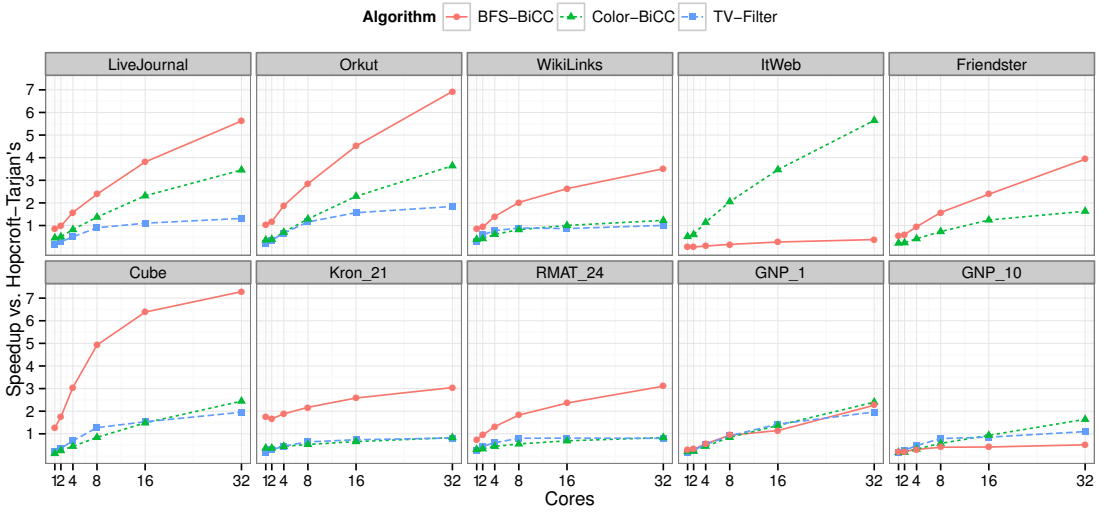

Figure 7.1. Parallel scaling of BFS and Coloring approaches as well as Cong and Bader's implementation relative to the serial Hopcroft-Tarjan algorithm.

Network	HT	BFS	(1)	(16)	(32)	Edge Ratio
LiveJournal	2.1	0.032	2.5	0.56	0.38	1.0
Orkut	3.4	0.025	3.3	0.75	0.49	0.60
WikiLinks	25	0.40	29	9.3	7.0	1.3
ItWeb	19	0.41	460	70	50	1.6
Friendster	79	0.46	150	33	20	0.95
Cube	1.2	0.042	0.98	0.20	0.17	0.031
Kron_21	1.8	0.030	1.1	0.71	0.60	0.015
R-MAT_24	4.7	0.12	6.2	2.0	1.5	0.042
GNP_1	11	0.082	37	10	5.0	3.1
GNP_10	6.5	0.15	31	16	12	19

Table 7.3. Execution time (in seconds) comparison between the serial algorithm, a standard BFS run, and the BFS-BiCC algorithm. Additionally, a ratio of the average number of edges examined during the inner-loop BFS is given.

the entire graph is given along with the serial algorithm for comparison. We also report a work estimate termed *edge ratio* in the final column. The edge ratio is the total number of edges explored during the second stage of the BFS-BiCC algorithm (where articulation vertices are being searched for) divided by the total number of edges in the graph. This essentially gives the multiplicative factor of additional/lesser work being performed in comparison to the Hopcroft-Tarjan algorithm, where all m edges are necessarily examined.

As can be observed, the graphs that have low edge ratios demonstrate the best performance with BFS-BiCC relative to the serial approach. In fact, for all four networks where the edge ratio is considerably less than 1.0, such as Orkut, Cube, Kron, and R-MAT, single thread execution time for BFS-BiCC is faster than the time for Hopcroft-Tarjan. As explained before, this is because of the direction-optimizing BFS that is very effective for certain low-diameter graphs.

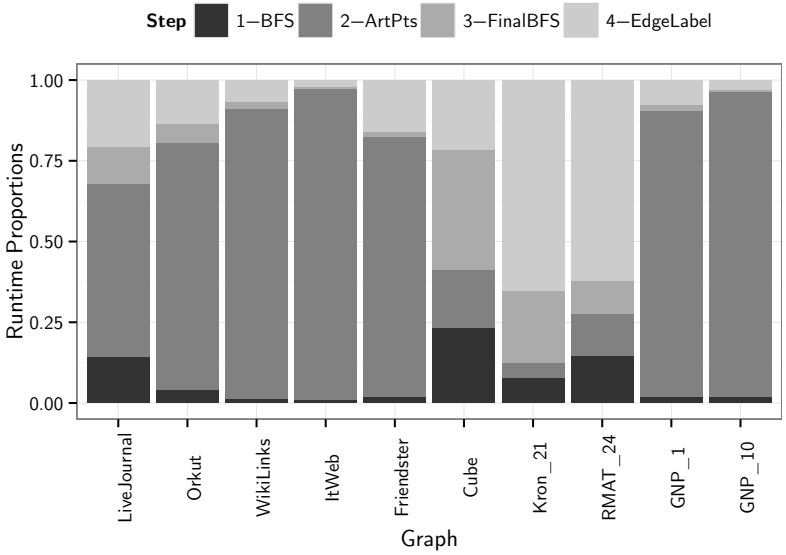

Figure 7.2. Per-step execution time breakdown of the BiCC-BFS approach.

Figure 7.2 gives the ratio of time spent in each of the four stages of the BFS-BiCC algorithm (initial BFS tree creation, articulation point identification, final level BFS run(s), edge labeling). For most graphs, the second stage dominates the overall execution time, as expected. The exception to this is a few of the graphs mentioned previously that have a low edge ratio. For those, the necessary $O(m)$ work required by edge labeling is the largest portion of execution time.

7.6.3 Color Propagation Analysis

We will now look at similar performance metrics obtained for the Color-BiCC algorithm. Table 7.4 gives the execution time of Color-BiCC with 1, 16, and 32 threads, as well as the execution time of a color propagation algorithm for identifying connected components in a graph.

We also compute an edge ratio metric in this case. It is calculated as the total number of edge propagations divided by the total number of edges. There

Network	HT	Color	(1)	(16)	(32)	Edge Ratio
LiveJournal	2.1	0.48	4.7	0.92	0.61	0.10
Orkut	3.4	0.44	9.4	1.5	0.93	0.063
WikiLinks	25	4.6	63	24	20	0.30
ItWeb	19	–	36	5.4	3.3	0.083
Friendster	79	21	360	63	48	0.063
Cube	1.2	0.19	9.7	0.84	0.51	0.17
Kron_21	1.8	0.43	5.0	2.8	2.2	0.62
R-MAT_24	4.7	1.2	16	6.8	5.6	0.57
GNP_1	11	1.8	53	8.3	4.8	0.064
GNP_10	6.5	2.0	37	7.0	4.0	0.062

Table 7.4. Execution time (in seconds) comparison between the serial algorithm, a color propagation algorithm for connected components, and the Color-BiCC algorithm. Additionally, the total number of color propagations divided by the number of edges in the network is reported.

is a moderate correlation between the edge ratio and performance of the coloring algorithm. This is especially apparent for the Kron_21 and R-MAT graphs, which demonstrate the highest edge ratio, and correspondingly are the only networks where fully-parallel coloring offers no advantage over the serial algorithm. It is noted that this ratio does not account for the initialization of the LCAs, which does make a considerable contribution to the overall time.

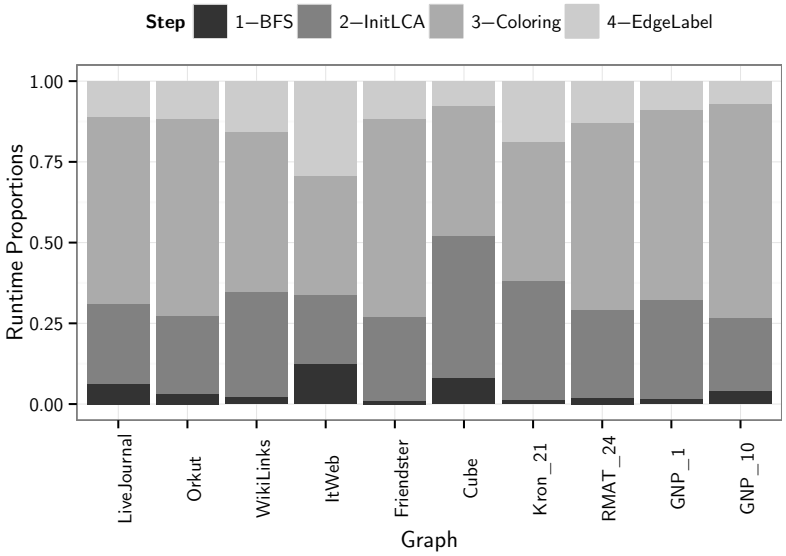

Figure 7.3. Per-step breakdown of the Coloring approach.

Network	BFS-BiCC Speedup	BFS-Color Speedup
LiveJournal	2.7	0.98
Orkut	2.7	0.94
WikiLinks	930	1.1
ItWeb	1.8	0.97
Friendster	3.7	1.0
Cube	0.92	0.98
Kron_21	3.5	1.1
R-MAT_24	12	0.96
GNP_1	1.1	1.0
GNP_10	0.42	1.0

Table 7.5. Speedups resulting for both the BFS and coloring algorithms with the heuristically-chosen root vertex compared to the average result over 20 randomly selected root vertices.

Figure 7.3 gives a per-step breakdown as Figure 7.2, with the four steps as the initial BFS tree creation, the initialization of the LCAs for all vertices, the primary coloring stage, and the final edge labeling. Compared to the breakdown for the BFS algorithm, the coloring algorithm has much more consistent ratios of times spent in all four stages across all networks. As is also observed, a majority of time is spent in the LCA initialization and coloring stages, with the coloring stage taking approximately twice as much time as the initialization stage.

7.6.4 Performance impact of root vertex choice

For the breadth-first search algorithm, the vertex selected as the initial root can considerably impact the overall running time. Table 7.5 demonstrates this difference. By selecting the vertex with the highest out-degree, the work required during the inner loop of the BFS algorithm can be minimized. This is because the final large biconnected component is usually found by the first vertex encountered in the final level queue. For networks containing a node with an exceptionally large out degree, such as WikiLinks (4.3M), the resultant difference in running time can be quite considerable. As the effort to find this vertex is minimal and often tracked during graph creation, the extra work required for this simple heuristic is minimal with regards to the possible payoff.

For coloring algorithms, there is minimal correlation between the initial root

vertex and the execution time of the algorithm. Ideally, the selected root should minimize the number of traversals required to initialize the LCA for all vertices, as well as minimize the number of the color propagations required. Selecting such a vertex seems like a challenging problem to be solved and the solution with the highest degree heuristic does not seem satisfactory.

7.7 Conclusions

This chapter introduced two novel shared-memory parallel algorithms for finding the biconnected components of an undirected graph. Since they both use simple and well-known subroutines, practical and efficient parallel implementations are much more feasible compared to prior algorithms. Additionally, our implementations of both of these algorithms offer considerable speedup over the TV-Filter implementation, while being more memory-efficient. As this chapter and the prior chapter demonstrated, identifying low level subroutines such as breadth-first search or color propagation can facilitate the development of novel algorithms that utilize these subroutines. Optimizing these subroutines for specific architectures can therefore facilitate their widespread use. While these previous two chapters focused on implementations targeted for multicore platforms, the next chapter will extend this general idea onto manycores processors such as GPUs and Intel Xeon Phis.

Chapter 8
Graph Processing on Manycores

8.1 Introduction

This chapter will build upon a number of the ideas introduced in the previous chapters, such as how to develop general purpose techniques for effective and performant graph processing on modern hardware. While previous chapters mostly focused on multicore CPUs, this chapter will shift focus onto optimizing for manycore processors such as GPUs and Intel Xeon Phis. The greater parallelism available on these processors introduces considerable challenges for processing irregular graphs. This chapter will discuss these challenges and introduce potential solutions. Again, our solutions will prove generalizable to a large class of graph algorithms, which we will demonstrate through straightforward and high performance implementations of the various subroutines of the Multistep algorithm.

8.2 Manycore Processing

This chapter will attempt to answer the following questions in the context of graph computations and manycore processors:

1. Can we identify frequently-used optimization strategies from the large and growing collection of tuned parallel graph computation implementations (e.g., [25, 140–143]), and create a structured methodology for designing new parallel algorithms? If one were to build a new framework for high-performance domain-specific graph computations, what would be the key optimization strategies to consider, and best practices to follow?

2. In addition to *parallel-for*, data-parallel scans, reductions, and sorting methods, what are some common abstractions used to design parallel graph algorithms?

3. Is it possible to develop performance-portable implementations of graph algorithms using advanced parallel programming libraries and frameworks with the optimizations and abstractions identified above?

We begin by observing that several recent graph algorithms and their efficient implementations follow the loop nest structure shown in Algorithm 8.1. The first point to note in Algorithm 8.1 is that the listing uses only simple, array-based data structures. Current state-of-the-art parallel implementations for several graph problems use array-based stacks, queues, and priority queues, as these structures are more amenable to applying data-parallel operations such as scans and reductions. Implementations of this general template differ in terms of graph representation, data structure access patterns, number of iterations of the outer loop, graph topology-based heuristics to reduce total work, synchronization overhead, etc. Intra- and inter-iteration dependencies hinder automatic compiler-based loop transformations such as unrolling, coalescing, collapse, and fusion.

Algorithm 8.1 A template followed by several serial and parallel graph algorithms operating on a sparse graph $G(V, E)$. $m = |E|$, $n = |V|$, and $m = O(n \log n)$.

```
Initialize temp/result arrays A_t[1..n], 1 ≤ t ≤ l.          ▷ l = O(1)
Initialize S_1[1..n].
for i = 1 to niter do                                        ▷ niter = O(log n)
    Initialize S_{i+1}[1..n].                                ▷ Σ_i |S_i| = O(m)
    for j = 1 to |S_i| do                                    ▷ |S_i| = O(n)
        u ← S_i[j]
        Read/update A_t[u], 1 ≤ t ≤ l.
        for k = 1 to |E[u]| do                               ▷ |E[u]| = O(n)
            v ← E[u][k]
            Read/update A_t[v].
            Read/update S_{i+1}.
        Read/update A_t[u].
```

Consider level-synchronous parallelizations of Breadth-First Search (BFS). *niter*, the number of outer-loop iterations, is bounded by the graph diameter. The arrays S_i correspond to the vertices in the current frontier, and the adjacencies of these

vertices can be visited in parallel. Arrays of size n (A_t) are used to store parent information, whether a vertex has been previously visited or not, and distance from the source vertex. Finally, there is a barrier synchronization before every iteration of the outer loop. For low-diameter (diameter is $O(\log n)$) graphs, the overhead of barrier synchronization is insignificant in comparison to the work performed in the inner loops. The arrays S_i store vertices in an arbitrary order for BFS. For other algorithms, such as ones for single-source shortest paths (SSSP), the ordering of vertices in S_i may be important to reduce the outer-loop iteration count. The direction-optimizing heuristic in a recent BFS algorithm [112] and the push-pull optimization in a recent parallel SSSP algorithm [144] can be viewed as work-reducing heuristics to switch between alternate representations of S_i in the inner loop.

The label propagation community detection heuristic [145] and the PuLP graph partitioning strategy [47] also fit within this general template. They differ in the access patterns of the temporary arrays and the result arrays in the inner loops. Algorithms that are similar to level-synchronous BFS, such as betweenness centrality [146], approximate diameter [147], strongly connected components [21], and biconnected components [44] also follow a similar structure. In fact, several PRAM graph algorithms can be viewed as instances of this template, and they would result in polylogarithmic parallel time algorithms (assuming low-diameter graphs and/or a $O(\log n)$ bound for the outer-loop iterations). This template is not restricted to shared-memory algorithms. Distributed-memory approaches for K-core decomposition also use a label propagation-like strategy [148], and implementations differ in the number of outer-loop iterations. Finally, open-source software packages providing state-of-the-art parallel implementations of graph algorithms, such as Parallel Boost graph library [149], MTGL [150], Galois [151], SNAP [69], PowerGraph [33], Ligra [115], NetworKit [152], etc., include several programs that fall under the Algorithm 8.1 template.

So we hypothesize that the first step towards creating an efficient parallel implementation of a graph algorithm would be to recast it such that it fits the general template shown in Algorithm 8.1. The focus of the current work is efficient graph analytics on manycore platforms such as NVIDIA and AMD GPUs and the Intel Xeon Phi MIC coprocessor. We do not want to reinvent the wheel for data-parallel subroutine implementations and parallel-for support. Hence, we use

an emerging node-level library and programming model called Kokkos [153], that lets us write code that is portable to GPUs, Intel Xeon Phi, as well as Intel and AMD x86 CPUs. In Section 8.3, we discuss key Kokkos features that enable us to quickly develop and compare alternate implementations.

As the next step, we develop several manycore implementations for the graph problems of BFS, color propagation, and strongly-connected components (SCC), expressing them in the template shown in Algorithm 8.1. The inner loop nests of BFS and color propagation have several differences, and so we explore both of these problems. The general manycore SCC algorithm is based on the prior Multistep SCC code [21]. Use of Kokkos lets us develop several alternatives for each problem and conduct a methodical evaluation of optimizations.

In Section 8.4, we present the third step, the key optimizations that are critical to manycore performance and portability. We primarily evaluate several tuned loop transformation strategies for the inner loop nest, and we customize these strategies for our use case of small-world graph analytics. These loop transformations, in essence, improve load balance and reduce irregular memory accesses. Our proposed strategies are similar to a compiler-based loop collapse [154,155]. However, compilers cannot automatically do this because of loop-carried dependencies. To the best of our knowledge, this is the first work to explore the loop transformations and manycore optimizations for SCC and color propagation problems. We arrive at this portable and performant manycore implementation of SCC using our algorithmic template, a Kokkos-based implementation, and architecture-aware optimizations.

The main observations from our empirical performance evaluation (see Section 8.5) are as follows:

- Our new loop collapse strategy, termed *Local Manhattan Collapse*, is very effective on GPUs and consistently results in the highest-performing variant for several problems.

- A GPU SCC implementation using the Local Manhattan Collapse strategy demonstrates up to a 3.25× speedup relative to a state-of-the-art parallel CPU implementation running on a dual-socket compute node.

- We find our GPU BFS implementation averages 1.74 GTEPS across a suite of 12 test graphs, comparable to the current state-of-the-art, without any BFS-specific tuning.

8.3 Portable Graph Algorithms for Manycore

8.3.1 The Kokkos Programming Model

The Kokkos library [153] was originally developed as a back-end for providing portable performance for scientific computing frameworks, but has since been extended to a more general-purpose library for parallel execution. The two primary capabilities of Kokkos include polymorphic multidimensional arrays optimized for varying data access patterns/layouts in different architectures and thread parallel execution that allows for fine-grained data parallelism on manycore devices.

The parallel execution model follows a dispatch model, where a single master CPU thread divides some N units of work to be processed on GPU. Each unit of work is executed by a single *thread* or *thread team*. On GPU, a thread team is comprised of multiple warps each executing on the same multiprocessor. This team of threads operates in a data parallel SIMT fashion, and is able to "communicate" via shared memory. In addition to optimizing the data layout in different devices, Kokkos also provides us the option to use the hierarchy of memory in manycore devices, such as thread block shared memory and texture cache. We use these features of Kokkos for the appropriate data structures. We also use the fast atomic operations and "thread team level" scan and reduction operations to synchronize between different threads in a thread block. One of the key design decision we made, using low-level simple array-based data structures, helps us when using Kokkos, as the layout of these simple arrays is then optimized by Kokkos in different devices (CPUs, GPUs and Phis), different types of memory (e.g., shared memory), and different access patterns (e.g., coalesced access). Use of custom data structures that are optimized for any single architecture would have prevented Kokkos level optimizations, in turn affecting portability.

8.3.2 Breadth-first Search

BFS is one of the most widely used basic graph subroutines, appearing in a vast number of more complex graph analytics. The goal of BFS is commonly to determine from a given root vertex either reachability status, distance, or BFS tree parent-child relationships for all or some subset of vertices in a graph. On each iteration $i \in niter$ of the algorithm, the status for all vertices that are distance i

away from the root is updated. Described in terms of Algorithm 8.1, we would first initialize S_1 to contain only the root. To determine distances, our result array A would be initialized to -1 for all vertices except for the root, which would be initialized to 0. Each iteration will examine all edges of all vertices in S_i. When a vertex v is encountered such that $A[v] < 0$, we update $A[v]$ to i and place v into S_{i+1}. There has been a lot of recent work focused on optimizing both CPU and manycore-based implementations of BFS [23, 25, 26, 112, 156, 157]. We don't explicitly consider fully optimizing BFS itself through our framework, but rather show how close we can get to state-of-the-art traversal rates by only considering simple techniques that affect per-unit-work assignments and which are applicable to a much broader class of algorithms.

8.3.3 Color Propagation

Color propagation is an iterative procedure that is useful for many different graph connectivity problems [15, 21, 44, 115]. An overview of the general algorithm is given by Algorithm 8.2. Note how it also follows the Algorithm 8.1 template, where S_i is our current queue and the result array A can be considered as the current color assignments for all vertices in the graph. We initialize A to be unique vertex identifiers and S_1 as all vertices in the graph. We then examine all edges, and when there exists a source vertex that has a higher color than one of its neighbors, that vertex propagates its color to the neighbor. The next work set S_{i+1} is comprised of vertices that have had their color altered. This process continues iteratively until no further propagations occur. As with BFS, we implement color propagation in a straightforward manner within our general framework.

8.3.4 Strongly Connected Components

The problem of computing strongly connected components (SCCs) in large directed small-world graphs is a common analytic for social networks [12] and a preprocessing step in scientific computing (among other uses) [16]. Using either BFS or coloring, straightforward parallel strongly connected component decomposition algorithms can be implemented [15, 19]. Combining both subroutines into an efficient Multistep procedure can result in considerable speedup for small-world graphs [21]. We use the BFS and color propagation subroutines implemented in our framework to

Algorithm 8.2 Color Propagation pseudocode.

```
A[1..n] ← 1..n                                  ▷ Set A[i] = i, 1 ≤ i ≤ n
S_1[1..n] ← 1..n                                ▷ Set S_1[i] = i, 1 ≤ i ≤ n
i ← 1
while S_i ≠ ∅ do
    S_{i+1} ← ∅
    for j = 1 to |S_i| do
        u ← S_i[j]
        for k = 1 to |E[u]| do
            v ← E[u][k]
            if A[u] > A[v] then
                A[v] ← A[u]
                Add v to S_{i+1}
    i ← i + 1
```

perform graph SCC decomposition via the Multistep procedure (we refer the reader to [21] for a more detailed description). Once again, outside of a few changes to initializations and the very innermost loops, few alterations need to occur to the original BFS and color propagation codes for the SCC problem.

8.4 Optimization Methodologies

In this section, we describe the optimization techniques used to achieve scalable performance on manycore architectures. These techniques are applicable to any algorithm that fits the template described in Algorithm 8.1. Furthermore, the optimizations are general enough for architectures that share similar characteristics, such as a very high core count, hierarchical memories, and small amounts of memory per thread. These characteristics of present day GPUs is expected to hold or become increasingly important in future manycore architectures. As a result, the optimizations described here are critical for scalable algorithms on modern and future systems.

8.4.1 Thread Teams, Local Synchronization, Shared and Global Memory

Current GPUs are organized as a number of streaming multiprocessors (for instance, 16 in a NVIDIA Maxwell GM204), each with a number of smaller cores (e.g., 128 CUDA cores in GM204). The number of threads that can be scheduled in a single streaming multiprocessor of a GPU can be up to 2048. The number of warps per streaming multiprocessor is 48-64, and the number of thread blocks is 8-16, depending on the microarchitecture. A similar hierarchy is also seen in the multiple hyperthreads per core of a Xeon Phi coprocessors, along with NUMA effects due to placement of cores near different memory regions. This results in multiple levels of parallelism for which algorithm developers need to design. The programming model in Kokkos abstracts this to a *thread team*, where a thread team corresponds to a thread block on GPU. In order to effectively utilize a streaming multiprocessor (SM), it is crucial to be able to schedule multiple thread blocks in each SM. Within each thread block, there is enough concurrency for thousands of threads, so that multiple warps can be kept busy at the same time. All of our algorithms use the thread teams concept to synchronize locally and utilize shared memory to communicate within a team when necessary.

The number of thread blocks that can be scheduled concurrently in a single SM is determined by the amount of shared memory used by each thread block (or a thread team). As all the threads in a team use the shared memory to synchronize among themselves before synchronizing to the global memory, the amount of shared memory used is an important resource. Increasing its size would reduce the number of writes to global memory by doing more local synchronizations, but it would also decrease the number of concurrent thread blocks that can be scheduled. Our approach balances shared memory usage with the parallelism available within each thread block. Finally, it is important that reads and writes to global memory are coalesced. Essentially, we want reads and writes for a single warp to be to be performed at neighboring global memory addresses to reduce total memory transfer, improve cache utilization, and ensure that threads in the warp are not idle waiting for the memory request of a single thread.

8.4.2 Hierarchical Exploration to Improve SM Utilization

A common optimization technique for algorithms dealing with irregular graph structure is special handling of the fringe cases, i.e., vertices with degree much larger than the average [23, 25]. This can be done at the granularity of a single level or through considering multiple classes of vertices in a hierarchy. In the GPU context, this might translate to a thread block working together to explore the edges of a vertex with an out-degree greater than the number of threads in the thread block, while a warp would explore vertices with an out-degree greater than the number of threads in a warp, but smaller than the number of threads in a block. Smaller vertices would be handled by individual threads. This general hierarchical technique has been previously used for irregular graph problems, referred to as the *deferring outliers* [23] and the *CTA+Warp+Scan* [25, 141] approaches.

Our implementation of this technique, which we term as *Hierarchical Expansion*, is given by Algorithm 8.3. In the Kokkos model, we consider parallelism at three hierarchies: team-level, warp-level, and thread-level. For each iteration of our algorithm, we remove a chunk of vertices V_T from the input work set S_i and pass it to a Kokkos thread team T. For good team-level work balance and multiprocessor utilization, the size of V_T is usually within a small factor of the size of T. The threads in each team work to process their input set, placing the high-degree vertices they encounter into a team-shared queue (when the degree is greater than the size of the thread team $|T|$) or warp-shared queue (when the degree is smaller than the size of a thread team, but larger than the size of a warp $|W|$). Smaller vertices get placed into a small thread-owned buffer Q_t for later serial expansion. The vertices in the team-level queue Q_T are collectively expanded by all threads, with potential updates to S_{i+1} kept in team-level shared memory. Once the Q_T queue is exhausted, the warp queue Q_W is examined. Each warp removes a vertex from the queue, and cooperatively expands its adjacencies. Finally, each individual thread serially expands the vertices in its buffer. Once all work is exhausted, the team collectively pushes their updates to the next iteration's work set S_{i+1}. We use team-level scans and reductions whenever possible to minimize global synchronizations.

The primary benefit to this type of approach is that it allows fine-grained warp utilization by limiting the serial expansion of high degree vertices by a single thread or warp. This leads to better load balance at the thread and warp level. This

Algorithm 8.3 Hierarchical Expansion.

```
Initialize A and S_1
for i = 1 to niter do
    Initialize S_{i+1}[1..n].
    for all Thread Teams do                          ▷ Team-level parallelism
        Retrieve subset V_T from S_i
        for j = 1 to |V_T| do                        ▷ Thread-level parallelism
            v ← V_T[j]
            if |E[v]| > |T| then
                Add v to Q_T                         ▷ Team-shared Queue
            else if |E[v]| > |W| then
                Add v to Q_W                         ▷ Warp-shared Queue
            else
                Add v to Q_t                         ▷ Thread-owned Queue
        Team-level synchronization
        for j = 1 to |Q_T| do
            v ← Q_T[j]
            for k = 1 to |E[v]| do                   ▷ Thread-level parallelism
                u ← E[v][k]
                Read/update A[u]
        Warp-level synchronization
        for j = 1 to |Q_W| do                        ▷ Warp-level parallelism
            v ← Q_W[j]
            for k = 1 to |E[v]| do                   ▷ Thread-level parallelism
                u ← E[v][k]
                Read/update A[u]
        for j = 1 to |Q_t| do                        ▷ Serial expansion by thread
            v ← Q_t[j]
            for k = 1 to |E[v]| do
                u ← E[v][k]
                Read/update A[u]
        Team-level synchronization
        Update S_{i+1}
```

approach also allows the use of shared memory to create new queues for the next iteration of an irregular graph problem, reducing the number of global synchronizations required. However, as vertices are assigned to a single team statically, there can still be some imbalance at the highest level. All teams might finish their work long before the team owning a highly-skewed vertex completes, delaying the start

of the next iterations and vastly under-utilizing available processing resources.

8.4.3 Loop Collapse for Better Load Balance

As shown by the template in Algorithm 8.1, many graph algorithms follow the pattern of two nested loops, where the outer loop is over the vertices and the nested inner loop is over the edges. Often, these two loops are not perfectly nested, as the vertex contents of the outer loop determine the start and end indices for the edges examined of the inner loop. There might be additional operations within the outer loop, such as changing the properties of vertices, adding vertices to the next queue, etc. While perfectly nested loops are great candidates for compiler-based optimizations, loops containing these other operations cannot be automatically optimized by compilers.

The importance of collapsing these loops increases when both loops are parallelizable and when the work in different outer loop iterations is heavily unbalanced. In graph analytic algorithms on graphs with skewed degree distributions, when the work in the outer loop varies based on the degree of the vertices, collapsing the inner loop is critical. When there are few threads, like in the CPU, a simple dynamic scheduling runtime can alleviate the problem [21]. However, it is hard to scale this approach to the thousands of threads in manycore devices. In our framework, we do the optimization a compiler might do, and collapse the two loops over vertices and edges into a single loop over all possible edges.

We employ the *Manhattan Collapse* [155], where a prefix-sum operation, easily parallelizable on GPUs with a scan-based procedure, is used to compute the bounds of each outer loop iteration. With the results of the prefix sums, a binary search is then used to compute the indices of the original inner-loop and outer-loop within the collapsed loop (referenced as HighestLessThan in Algorithms 8.4 and 8.5). The overhead associated with reverse-engineering the vertex information is offset by the good load balance achieved by each thread. This general approach has been explored before by Merrill et al. for GPUs in the context of BFS [9] and by Davidson et al. for SSSP [141]. As with the work of Davidson et al., we consider two forms of the Manhattan Collapse, implementing it at both the global and local level.

8.4.3.1 Local Manhattan Collapse

For our local implementation, we do not require any additional global storage, apart from the queues and work arrays updated in the algorithm. An overview of this approach is given by Algorithm 8.4. We statically partition our work set S_i on a per-vertex basis and pass each partition V_T to our thread teams. The thread team computes prefix sums P over V_T based on out-degree. P is stored in shared memory. The final prefix sum in P is the sum of edges for V_T, and therefore proportional to the total work that the team needs to do. We can then equally distribute this work among all the threads in the team. To get a specific edge based on a per-thread work assignment j, the source vertex is determined by examining the prefix sums array, and finding the index k that corresponds to a value in P greater than or equal to j, and less than the value at the next highest index. The specific out-edge u from the source vertex can be found by using the difference between the work assignment and the value at the found index in the prefix sums. With the (u, v) pair, the thread can now perform its assigned work.

Algorithm 8.4 Local Manhattan loop collapse.

```
Initialize A_t and S_1
for i = 1 to niter do
    Initialize S_{i+1}[1..n]
    for all Thread Teams do                          ▷ Team-level parallelism
        Retrieve subset V_T from S_i
        P ← PrefixSums(V_T)
        Max ← max(P)
        for j = 1 to Max do                          ▷ Thread-level parallelism
            k ← HighestLessThan(P, j)
            u ← V_T[k]
            v ← E[u][j − P[k]]
            Read/update A_t[u] and A_t[v]
        Team-level synchronization
        Update S_{i+1}
```

The primary benefit of the Local Manhattan Collapse is that it leads to full warp and thread utilization of processor resources. When the cost of looking up a work assignment is low compared to the work that needs to be done, this approach is highly beneficial. As with Hierarchical Exploration, a major drawback to doing the Local Manhattan Collapse is that a vertex is still assigned to a single team,

which might lead to work imbalances for highly skewed graphs.

8.4.3.2 Global Manhattan Collapse

To alleviate any potential work imbalance issues, we implemented a fully-partitioned approach, where the prefix sums for the current iteration are computed on the previous iteration as updates were pushed to the next-iteration work set S_{i+1}. By doing this, we can statically distribute an equal number of *edges* to each team instead of vertices. As can be seen in Algorithm 8.5, the approach closely follows our local method. The primary difference lie in the prefix sum arrays, P_i and P_{i+1}, which must be globally stored and synchronously updated. To minimize data transfer requirements, each thread team can determine its start and end offsets in P_i and do a single transfer of the needed portion to shared memory. Additionally, pushes to P_{i+1} and S_{i+1} can also be coalesced, with only a single atomic update required per team. Because each team needs to determine the offset to start writing to P_{i+1}, as well as the current running sum, we package both these values into a single atomically-updated 64-bit `long int` and perform an atomic fetch-and-add on a current global value.

Algorithm 8.5 Global Manhattan loop collapse.

```
Initialize A_t and S_1
Initialize P_1
for i = 1 to niter do
    Initialize S_{i+1}[1..n]
    for all Thread Teams do
        Retrieve subset j_T to j_{T+1} of max(P_i)
        for j = j_T to j_{T+1} do                    ▷ Thread-level parallelism
            k ← HighestLessThan(P_i, j)
            u ← V_T[k]
            v ← E[u][j − P_i[k]]
            Read/update A_t[u] and A_t[v]
        Team-level synchronization
        Update S_{i+1} and P_{i+1}
```

Ideally, the Global Manhattan Collapse should offer the best work partitioning among thread teams and fastest execution times for a given algorithm. However, as we will show in our results, there are other key factors that hurt the performance of the Global Manhattan Collapse relative to the Local method. For simple graph

algorithms with minimal work per edge, the cost of reading and writing to an additional global array is relatively high. Amortizing this startup and end cost by increasing work per team is not necessarily a good solution, as we would ideally like to have an as-large-as-is-practical number of teams to hide the memory access latencies inherent to the rest of the implemented algorithm. Further, on graphs with a relatively consistent degree, or a modest number of outliers, this method offers no additional benefit in terms of equal per-team work distribution relative to the local collapse. Finally, the maximal degree of many real-world graphs is bounded by $O(\sqrt{n})$. As long as the maximal degree is less than $O(n)$ and there are relatively few outliers, the level of fine-grained global work distribution offered by the global collapse is likely not necessary.

8.5 Performance Analysis and Discussion

8.5.1 Experimental Setup

We evaluate our algorithms on single nodes of three clusters, the *Shannon* and *Compton* systems at Sandia and the NSF *Blue Waters* system at the NCSA. A Shannon node has two Intel Xeon E5-2670 Sandy Bridge-EP processors with 128 GB main memory and an NVIDIA Tesla K40M GPU. The K40M GPU has 12 GB DDR5 memory, 2880 cores, and a peak memory bandwidth of 288 GB/s. Each GPU-enabled compute node of Blue Waters has one AMD 6276 Interlagos processor with 32 GB main memory and an NVIDIA Tesla K20X GPU. The K20X GPU has 6 GB DDR5 memory, 2688 cores, and a peak memory bandwidth of 250 GB/s. For GPU parallelism, Kokkos uses the CUDA programming model. We use Compton nodes for running our Kokkos and OpenMP implementations on its Intel Xeon processors and Intel Xeon Phi MIC coprocessors (Kokkos also utilizes OpenMP for multicore parallelism). The MICs on Compton contain 57 cores at 1.1 GHz with 6 GB memory. In all cases, the version of Kokkos used in our evaluation came from release 11.10.1 of Trilinos, we used `icc` and `nvcc` with the `-O3` optimization option for compilation along with `-arch=sm_35` when compiling for GPU.

We used several real small-world directed graphs that range in size from 5.1 million to 936 million edges for testing. These are listed in Table 8.1. The graphs are from the SNAP database [69], the Koblenz Network Collection [80], and the

Network	n	m	Degree		(S)CCs		
			avg	max	Count	nontriv.	max
Google	875 K	5.1 M	5.8	5 K	370 K	12 K	410 K
Flickr	820 K	9.8 M	12	10 K	277 K	7.3 K	530 K
XyceTest	1.9 M	8.2 M	4.2	250	400 K	2.0 K	1.5 M
LiveJournal	4.8 M	69 M	14	20 K	970 K	23 K	3.8 M
RMAT2M	2.0 M	128 M	64	8.7 K	1 M	1	1.0 M
GNP2M	2.0 M	128 M	64	95	1	1	2.0 M
Indochina	7.4 M	194 M	26	180 K	1.6 M	40 K	3.8 M
DBpedia	67 M	258 M	3.9	650 K	55 M	2.9 M	8.9 M
HV15R	2.0 M	283 M	140	170 K	24 K	15	120 K
uk-2002	18 M	398 M	16	4 K	3.7 M	70 K	12 M
WikiLinks	26 M	600 M	23	400 K	6.6 M	60 K	19 M
uk-2005	39 M	936 M	24	130 K	5.8 M	223 K	26 M

Table 8.1. Information about test networks. Columns are # vertices, # edges, average and max. degree, # of SCCs, # number of nontrivial SCCs, and size of the largest SCC.

University of Florida Sparse Matrix Collection [75]. We selected these graphs to represent a wide mix of graph sizes and topologies. Graph topology also has a strong influence on the performance of BFS and color propagation, while the number of total and nontrivial SCCs, as well as the size of the largest SCC, play an important role in determining performance of the SCC algorithm.

We report BFS and color propagation performance in terms of the Giga Traversed Edges per Second (GTEPS) metric, which normalizes running time to the total number of edges accessed (in billions). Note that our input graphs are directed and most of them have a large SCC. For each BFS execution, we track the total number of edges visited. Similarly, we count the number of vertex color and edge updates to determine overall performance for color propagation. We also run multiple iterations of both algorithms on all the target systems to reduce any variation in running time. In order to be consistent with BFS and color propagation results, we normalize SCC performance also by the number of edges and report an overall GEPS (Giga Edges per second) rate for each graph.

For the Kokkos GPU approach, we fix thread queue sizes at 16, work chunks at 256 vertices per thread team for Hierarchical and Local Manhattan, and work chunks at 2048 per thread team for Baseline (vertex chunks) and Global Manhattan (edge chunks). For Xeon Phi and CPU, we used larger queues of size 1024 and work

chunks of 2048. These values were selected for exhibiting the fastest performance across a range of values on our test suite. We will fully explore the performance impact of these algorithmic parameters in future work.

8.5.2 BFS Performance

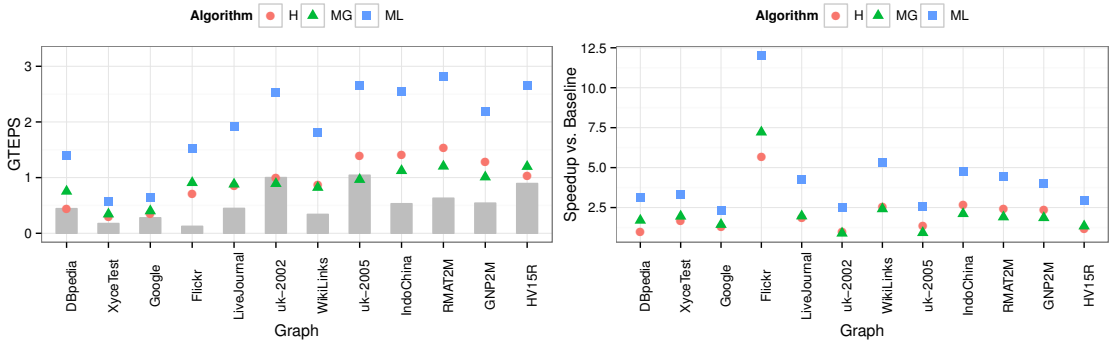

Figure 8.1. BFS performance in terms of GTEPS (left) and speedup vs. Baseline (right) on a Tesla K40M using Manhattan-Local (ML), Manhattan-Global (MG), and Hierarchical (H) loop collapse strategies.

Figure 8.1 gives the performance rates of the Kokkos-based BFS implementations on the Tesla K40M GPU. The baseline rate in the figure corresponds to performance with a vertex-based partitioning of the frontier array among thread teams. It is not a trivial implementation and our speedup numbers are conservative in that sense. We see consistent and significant speedups with three loop collapse strategies (H: hierarchical, MG: Manhattan Collapse using global memory, ML: Manhattan Collapse using GPU shared memory). Using H, MG, and ML, the speedups (geometric mean) over baseline are 1.82×, 1.82×, 3.85×, respectively, for the twelve graphs considered. The graphs are ordered in the figure in increasing order of average vertex degree, from DBpedia (3.9) to HV15R (140). Apart from a couple of anomalies, there is a reasonable correlation between average vertex degree in the graph and the BFS performance of the best variant (ML). Prior GPU graph algorithms work [26] has also made similar observations. However, one striking aspect is that the tuned variant can be more than an order-of-magnitude faster than the baseline, as we note for the Flickr graph. This is likely due to the large skewed degree distribution of Flickr, which severely limits the performance of the baseline approach. Other work has also noted the importance of parallel work

balance with this particular graph [141].

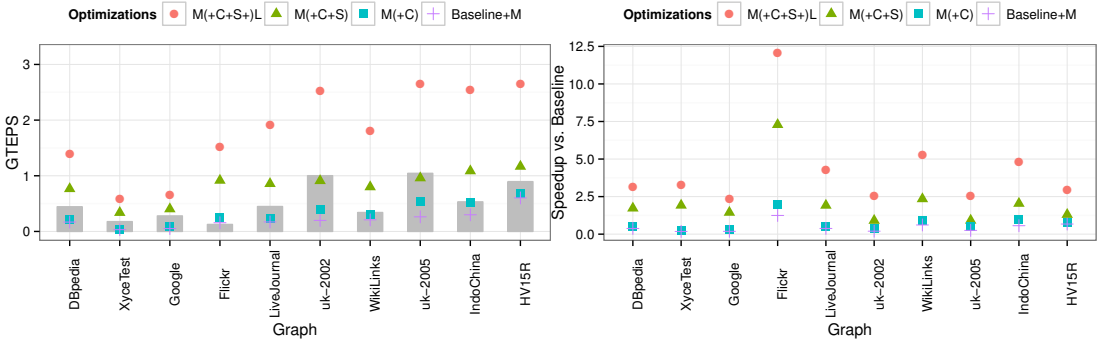

Figure 8.2. Impact in terms of GTEPS (left) and speedup vs. Baseline (right) of various optimization strategies (Manhattan Collapse (M), coalescing (C), team-scan (S), and local primitives (L)) on a Tesla K40M BFS performance.

Next, we summarize the impact of other optimization strategies discussed in Section 8.4. Figure 8.2 gives BFS performance of the baseline and the ML variant again. In addition, we add optimizations in a structured manner to the code, starting with Baseline and finally getting to tuned ML (indicated by M+C+S+L in the figure). The intermediate steps are untuned Manhattan Collapse (indicated by Baseline+M), Manhattan Collapse with memory coalescing (M+C), Manhattan Collapse with memory coalescing and utilizing team-based scan procedures (M+C+S). The final step is the usage of temporary shared memory arrays for each thread team. It is interesting to note that Manhattan Collapse by itself does not provide much performance improvement. It is only after a methodical restructuring of the code, including optimizations such as coalescing and use of optimized scan primitives, that we are able to get the full benefit of the loop collapse optimization.

8.5.3 Color Propagation Performance

Figure 8.3 shows the performance of the loop collapse strategies on color propagation. Unlike BFS, the global Manhattan Collapse strategy does not consistently improve performance over the baseline. For a majority of the graphs, it is actually slower than baseline. Using H and M, though, the speedups (geometric mean) over baseline are 1.72× and 3.10×, respectively, for the twelve graphs considered. Performance of the best variant (ML) seems to well-correlated with average graph degree, with the exception of the synthetic RMAT2M and GNP2M graphs. We see the highest

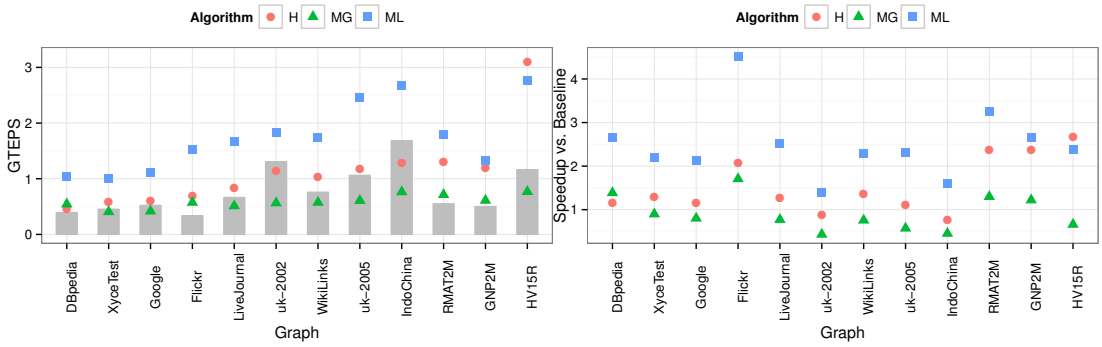

Figure 8.3. Color propagation performance in terms of GTEPS (left) and speedup vs. Baseline (right) on a Tesla K40M using Manhattan-Local (ML), Manhattan-Global (MG), and Hierarchical (H) loop collapse strategies.

overall speedup over baseline (nearly 4.5×) with the Flickr graph. MG performs poorly on several instances due to the nature of iterative color propagation, which tends to have a long tail containing lots of low degree vertices. This effect is especially pronounced on the web graphs (uk, IndoChina), which tend to have long strings of singly-connected vertices. This hurts the performance of MG relative to ML in two ways. Firstly, the low average degree increases the amount of total transfer per team to and from the global prefix sum array. Secondly, the *consistently* low vertex degrees offer no benefit with regards to work partitioning among teams relative to the other approaches.

8.5.4 SCC Evaluation and Performance Portability

We finally evaluate performance of various SCC implementations. Recall that SCC algorithms use both BFS-like and color propagation-like loop nests, in addition to other graph topology-related work reduction heuristics [21, 114]. Our baseline Kokkos implementation for SCC is based on our prior Multistep [21] multicore parallel algorithm. Note that the BFS subroutine in Multistep is direction-optimizing, similar to [112, 157]. Thus we have also been able to express a fast heuristic work-reduction strategy in our Kokkos framework. We further improved the baseline approach using the ML and MG loop collapse strategies, and other GPU-specific optimizations. Figure 8.4 provides a cross-platform comparison of the various approaches on our test suite. Performance rates are indicated in terms of billions of edges per second. Our prior CPU Multistep implementation uses OpenMP, and

can be compiled and run on x86 systems as well as Intel's Xeon Phi coprocessors. We thus report these results on the Sandy Bridge-EP host processor and the Xeon Phi coprocessor, indicated as SNB and KNC in the figure. The OpenMP Multistep implementation is labeled OMP. The Kokkos baseline approach runs on all four platforms, and it is labeled as previous (B). Because Kokkos uses a single thread per team for the Xeon Phis and CPU, we only report performance for the MG variant of loop collapse on these systems, as the ML variant would default into an inefficient (B). We don't consider hierarchical exploration due to the consistently superior performance of MG in previous experiments. We only include ML as a comparator for systems that can't utilize MG. Any missing data points in both Figure 8.4 and Table 8.2 are due to memory limitations on the GPUs and Xeon Phi.

Consider the SNB column of the figure first. We observe that the OMP multistep performance varies between 0.1 to 5 GEPS, a nearly 50× variation. The anomalously-high performance on RMAT2M and GNP2M is due to the fact these synthetic graphs are relatively easy instances for the Multistep algorithm (there is only a single non-trivial SCC, so color propagation is never run). The graphs are ordered from top to bottom by average vertex degree. While OMP tends to do better than the Kokkos baseline on smaller graphs, for four of the twelve graphs, including the larger uk web crawls, the Kokkos baseline is in fact faster than the state-of-the-art OpenMP-based Multistep. MG, the algorithmic variant designed for GPUs, did not do as well as the baseline in SNB.

The Xeon Phi performance results are quite interesting. The Kokkos baseline variant now consistently outperforms Multistep OMP. In comparison to parallel SNB performance, the absolute performance results on KNC are lower. However, note that these results were obtained with little or no parameter tuning for KNC. Besides three instances, MG again lags behind baseline. Thus we can conclude that the loop collapse strategies designed specifically for GPUs may not really lead to portable performance on KNC, without additional tuning.

The GPU SCC performance results are as expected. Notably, we could easily combine the Kokkos BFS and coloring implementations to create this SCC algorithm, and overall performance is quite favorable in comparison to the best parallel CPU implementation.

The original Multistep algorithm compiled with OpenMP and running on

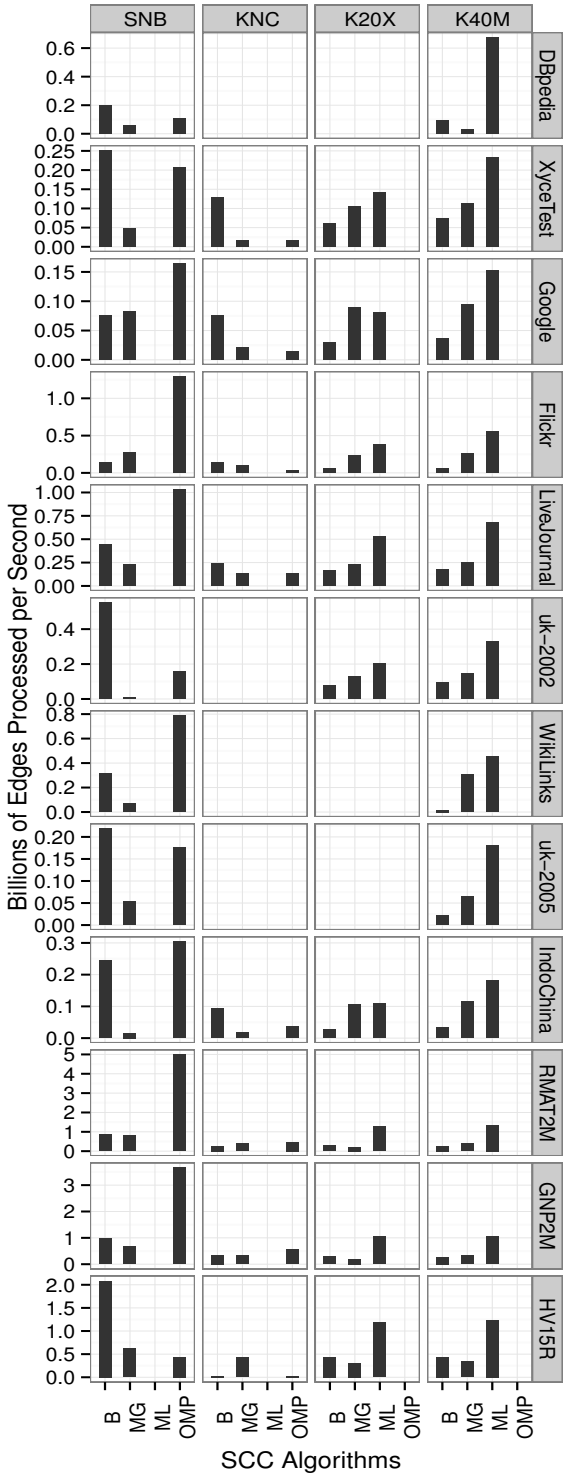

Figure 8.4. Cross-platform performance comparison of SCC implementations.

Network	SCC				BFS	Coloring
	SNB	KNC	K20X	K40M	K40M vs K20X	
	GEPS				GTEPS ratio	
Google	0.16	0.08	0.09	0.15	1.66	1.19
Flickr	1.29	0.14	0.38	0.56	1.37	1.00
XyceTest	0.25	0.13	0.14	0.23	1.08	1.16
LiveJournal	1.03	0.24	0.53	0.68	1.11	1.04
RMAT2M	4.99	0.47	1.28	1.35	1.13	0.95
GNP2M	3.66	0.56	1.06	1.05	1.04	1.04
Indochina	0.30	0.09	0.11	0.18	1.08	1.22
DBpedia	0.20		0.33	0.34	0.82	1.16
HV15R	2.09	0.43	1.18	1.23	2.20	0.99
uk-2002	0.55		0.20	0.33	1.26	1.15
WikiLinks	0.79			0.45	1.27	1.12
uk-2005	0.22			0.18		
Geo. Mean	0.69	0.21	0.35	0.43	1.23×	1.09×

Table 8.2. Cross-architectural performance comparison of best variants.

CPU shows the most consistent performance, followed by the GPU Kokkos ML algorithm running on GPU. Exploring cross-architectural and cross-implementation performance on each graph instance, we note different reasons for why a particular implementation is faster or slower. Multistep was designed to run on CPU with low diameter graphs and, as such, tends to dominate performance-wise on the smaller graphs, where there is lower available work parallelism, the graphs are less skewed, and the problems are generally easier to solve. This is apparent on the two simplest instances, the GNP and RMAT graphs. The GPU ML code arguably shows increasing relative performance with increasing problem difficulty, which is exemplified by DBpedia, the most skewed graph with the largest number of nontrivial SCCs. The additional parallelism for GPU ML across the adjacencies of the largest outliers in DBpedia makes a large relative impact. MG for both CPU and GPU does not show as good performance for DBpedia because, while there is improved parallelism across the largest adjacencies, DBpedia also has a very long tail of low degree vertices. This makes color propagation run very slow with MG due to all of the additional read and writes to the global prefix sums array.

In Table 8.2, we list the SCC GEPS rate of the best-performing variant on each platform. The geometric mean of GEPS rates on each platform for SCC are also

listed. Overall, K20X is 1.67× faster than KNC for SCC. For BFS and Coloring, we compare performance of the best variant on K40M to the best-performing one on K20X. We observe that K40M is overall 1.23× faster for BFS and 1.09× faster for color propagation.

Although we note the present performance benefit of running on the multicore system relative to our manycore implementations, we make a few final points. Current trends in HPC indicate increased parallelism to become more prevalent, with an explicit distinction between a host CPU and coprocessor/GPU becoming blurred. This will make any present performance gaps between multicore and manycore codes less relevant. Additionally, more complex memory hierarchies and wider variation in architecture designs will make it a challenge to write algorithms that are efficient on future architectures. Hence, as we motivated previously, developing manycore techniques that exploit wide parallelism and varied memory hierarchies through reliance on a general framework and back-end for architecture-optimized parallelization will ease the burden on algorithm designers to design portable code for future systems.

8.5.5 Comparisons to Prior Work

To the best of our knowledge, this is the first work on Kokkos-based graph computations targeting GPUs and Xeon Phi accelerators. For SCC, we performed direct comparisons with our prior OpenMP based Multistep method, as discussed in the previous subsection. In terms of mean performance rates, we believe that 0.43 GEPS for SCC, using a high-level framework such as Kokkos, is significant. Our mean BFS performance rate on the K40M is 1.74 GTEPS across all the test networks considered, and the best rate is 2.82 GTEPS for the RMAT2M network. Recently, Nguyen et al. [158] compare performance of several parallel graph analysis frameworks (Ligra, Galois, PowerGraph, GraphChi, and variants) for various graph analytics routines on a 40-core Intel Westmere-EX system. The best BFS performance reported, with Galois on the twitter40 graph, corresponds to a rate of 2.1 GTEPS. Merrill et al. [25] report up to 3.3 GTEPS on the RMAT2M network for an optimized CUDA BFS implementation. For tuned CUDA-based SSSP approaches, Davidson et al. [141] report a peak performance rate of 0.35 GTEPS on an RMAT network and an NVIDIA GTX 680 (GK104). Thus, we

believe that our approaches are competitive with the current state-of-the-art on multicore and manycore platforms.

8.6 Conclusions

We used an algorithmic template that is common to lot of graph algorithms to express algorithms for strongly connected components, breadth first search and color propagation. This algorithmic template was used for portable manycore implementations using the Kokkos library and then optimized for architecture specific features like teams of threads and algorithmic features like loop-collapsing. We gave credence to the the efficacy of our approach by demonstrating the performance of a strongly connected components algorithm that is up to 3.25× faster than a parallel CPU implementation.

We conclude with some commentary on questions posed in Section 8.1. To answer questions (a) and (b), we advocate using simple array-based data structures and an iterative loop nest to perform graph computations, as shown in Algorithm 8.1. This simplifies transitioning from serial to multicore to manycore algorithms. The *Local Manhattan Collapse* optimization proved to be the biggest contributor to performance improvement over a baseline version. Given that most current and emerging real-world networks have skewed degree distributions, this would be the primary optimization strategy for graph analytics. The algorithms we studied in this chapter use the abstraction: "given a large unordered set of vertices, how do we efficiently read and update attributes of the vertices and their adjacencies?" Using Kokkos, we see promising results for performance portability. The performance of our Baseline SCC algorithm on Xeon Phi is 1.97× faster than an OpenMP-based implementation. Further, the multicore CPU algorithm based on Kokkos is only 30% slower than a hand-tuned OpenMP code. So we conclude as an answer to question (c), *yes*, performance-portable graph algorithms are possible using the previously identified libraries and optimization techniques.

While this chapter and the bulk of this prior thesis has focused mostly on single node shared-memory optimizations and algorithmic implementations, the rest of this thesis will instead shift focus to general techniques and considerations when processing in the distributed memory space.

Chapter 9
Complex Small-world Graph Partitioning

9.1 Introduction

This chapter denotes the shift in this thesis from a focus on shared-memory to distributed-memory graph processing. In this chapter, we will introduce our graph partitioner, PuLP, which can be used to accelerate graph algorithms running in distributed memory. The main benefits of using our PuLP partitioner are its low memory overheads, high efficiency, and comparable partition quality compared to other state-of-the-art partitioners. This chapter will introduce the general PuLP algorithm, describe its implementation, and give a performance comparison to prior art.

9.2 Graph Partitioning

Within the past few years, several online repositories that host representative real-world graphs with up to billions of vertices and edges (e.g., [69, 75, 80]) have emerged along with several open-source and commercial distributed graph processing frameworks (e.g., PowerGraph [33], Giraph [36], Trinity [39], PEGASUS [38]). The primary goal of these frameworks is to permit in-memory or parallel analysis of massive web crawls and online social networking data such as what is hosted by the online repositories. These networks are characterized by a low diameter and skewed vertex degree distributions, and are informally referred to as *small-world* or

power law graphs. The graph processing frameworks use different I/O formats and programming models [159, 160], but all of them require an initial vertex and edge partitioning for scalability in a distributed-memory setting.

As such, a key motivation for this chapter is how to efficiently partition large scale graphs to ensure high performance of in-memory graph computations. Two common topology-aware approaches to generate load-balanced partitions are (i) randomly permuting vertex and edge identifiers, and (ii) using a specialized graph partitioning tool. Random permutations ensure load balance, but hurt locality and inter-task communication. Graph partitioning methods attempt to maximize both locality and load balance, and optimize for aggregate measures after partitioning, such as edge cut, communication volume, and imbalance in partitions. There is a large collection of partitioning methods [27, 28] that perform extremely well in practice for regular, structured networks. However, there are three issues that hinder use of existing graph partitioners for small-world network partitioning:

1. Traditional graph partitioners are heavyweight tools that are designed for improving performance of linear solvers. Most graph partitioning methods use multilevel approaches, and these are memory-intensive. Partitioning time is not a major consideration, as it is easy to amortize the cost of partitioning over multiple linear system solves.

2. The collection of complex network analysis routines is diverse and constantly evolving. There is no consensus on partitioning objective measures. Partitioning with multiple constraints and multiple objectives is not widely supported in current partitioners.

3. Small-world graphs lack good vertex and edge separators [70]. This results in problems that are hard to partition the traditional way, resulting in even high-performing traditional partitioners taking hours to partition large small-world graphs.

This chapter takes a fresh look at the problem of distributed graph layout and partitioning. We introduce a new partitioning method called PuLP (Partitioning using Label Propagation), and explore trade-offs in quality and partitioning overhead for a collection of real and synthetic small-world graphs. As the name suggests, PuLP is based on the label propagation community identification algorithm [145]. This algorithm generates reasonably good quality results for the

community identification problem [27, 28], is simple to implement and parallelize, and is extremely fast. One of the goals of any graph partitioning scheme is to reduce the number of inter-partition edges (or the edge cut), as it loosely correlates to inter-node communication costs. Since communities are tightly connected vertices, co-locating vertices of a community in a partition will increase the proportion of intra-partition edges. Other related work has used label propagation as part of a multi-level scheme for graph coarsening [161–163] or, similar to PuLP, as part of a single-level method [164–166]. However, to our knowledge, PuLP is the first label propagation-based partitioner that considers both multiple constraints and multiple objectives. It is also the first single-level partitioner that can produce cut quality comparable to state-of-the-art multilevel partitioners with multiple constraints.

In typical graph analytic algorithms, the number of vertices/edges in each partition represent the local work and the memory usage. In parallel graph analytics utilizing a bulk synchronous parallel (BSP) model, we also want to minimize the maximum communication (cut edges) incurred by any single partition. As a consequence, our approach also tries to impose vertex and edge balance constraints, with the goal to minimize both total edge cut and maximal per-part edge cut.

To demonstrate the efficacy of our approach, we compare the quality of results obtained using PuLP to the multilevel k-way partitioning method in METIS [29, 167] and ParMETIS [168]. We use the multiple constraint version of both the codes [169, 170]. We also compare our code against the KaHIP [171] library, which uses label propagation within a multilevel framework. Our contributions in the chapter are:

1. A fast, scalable, partitioner that is practical for partitioning small-world graphs.

2. A partitioner that handles the multiple objective and multiple constraints that are important for small-world graph analytics.

3. A performance study for a collection of 15 large-scale small-world graphs (number of edges range from 253 thousands to 1.8 billion).

For the large networks and commonly-used quality measures (edge cut), our partitioning scheme is comparable to METIS and better than it in additional objectives (maximum per-part edge cut) for a wide range of partition counts (2-1024) and with fixed edge and vertex balance constraints. The main advantage of

our approach is the relative efficiency improvement: for instance, to partition the 1.8 billion edge Slovakian domain (.sk) crawl [74], PuLP takes less than a minute on a single compute node to generate 32 way partitions of this graph while satisfying given vertex and edge balance constraints. Due to higher memory requirements than the 64 GB available on the test node, neither METIS, ParMETIS, nor KaFFPa were able to successfully partition this graph on our test system.

Note that graph partitioning is frequently used as a preprocessing routine in distributed graph processing frameworks, so PuLP might be used to accelerate the execution time of graph algorithms in software such as Giraph, PowerGraph, or GraphX.

9.3 Preliminaries: The Graph Partitioning Problem

We consider parallelizing analytics over large and sparse graphs: the numbers of vertices (n) are on the order of at least tens of millions, and the numbers of edges (m) are much closer to $O(n)$ instead of $O(n^2)$. The graph organization/layout in a distributed-memory system is characterized by the 'distribution, partitioning, ordering' triple. The current state-of-the-art in distributed-memory implementations is to adopt a graph distribution scheme, a specific partitioning method, and then organize inter-node communication around these choices. In this chapter, we focus on the partitioning aspect of the aforementioned triple and use 1D distribution and natural ordering.

Given p processes or tasks, the most common distribution strategy, called 1D distribution, is to assign each task a p-way disjoint subset of vertices and their incident edges. The advantages of the 1D scheme are its simplicity, memory efficiency, ease of parallel implementation of most graph computations using an 'owner computes' model, and the fact that the interaction of graph partitioning methods and 1D distributions is well understood [27, 28]. A disadvantage of 1D methods is that some collective communication routines could potentially require exchange of messages between all pairs of tasks (p^2), and this may become a bottleneck for large p. However, we focus on 1D instead of 2D or hybrid partitioning methods for this work, due to the increased complexity of algorithm design, primarily the communication steps, and increased memory usage for storing the graph (since both the row and column dimensions may be sparse) that 2D and

hybrid partitioning methods require. Potential future work might be to utilize the 1D partitions that we compute in a 2D distribution [172] or with other distribution schemes such as degree-based ones [173].

The specific partitioning problem we are interested in for graph analytic applications and which is solved in PuLP can be formally described as below. Given an undirected graph $G = (V, E)$, partition V into p disjoint partitions. Let $\Pi = \{\pi_1, \ldots, \pi_p\}$ be a nearly balanced partition such that $\forall i = 1 \ldots p$,

$$(1 - \epsilon_l)\frac{|V|}{p} \leq |V(\pi_i)| \leq (1 + \epsilon_u)\frac{|V|}{p} \tag{9.1}$$

$$|E(\pi_i)| \leq (1 + \eta_u)\frac{|E|}{p} \tag{9.2}$$

where ϵ_l and ϵ_u are the lower and upper vertex imbalance ratios, η_u is the upper edge imbalance ratio, $V(\pi_i)$ is the set of vertices in part π_i and $E(\pi_i)$ is the set of edges such that both its endpoints are in part π_i. We define the set of cut edges as

$$C(G, \Pi) = \{\{(u, v) \in E\} \mid \Pi(u) \neq \Pi(v)\} \tag{9.3}$$

$$C(G, \pi_k) = \{\{(u, v) \in C(G, \Pi)\} \mid (u \in \pi_k \vee v \in \pi_k)\} \tag{9.4}$$

Our partitioning problem is then to minimize the two metrics

$$EC(G, \Pi) = |C(G, \Pi)| \tag{9.5}$$

$$EC_{max}(G, \Pi) = \max_k |C(G, \pi_k)| \tag{9.6}$$

This can also be generalized for graphs with edge weights and vertex weights. PuLP can be extended to handle other metrics like the total communication volume and the maximum per-part communication volume. The communication volume for a given part i can be simply defined as the total number of vertices within one hop of all vertices in part i that are assigned to a different part. The total communication volume would be the sum communication volumes of all parts produced by the partitioning.

In the past, multi-constraint graph partitioning with the EC objective has been implemented in METIS and ParMETIS [169, 170]. We will compare against both these methods in Section 9.5. Pinar and Hendrickson [174] suggested a framework for partitioning with complex objectives (but with a single constraint)

that is similar to our iterative approach. More recently, there are multi-objective partitionings [175] and multi-constraint and multi-objective partitionings [176] for hypergraph partitioning. However, hypergraph methods are often much more compute-intensive than graph partitioning methods.

9.4 PuLP: Methodology and Algorithms

This section introduces PuLP, which is our methodology for utilizing label propagation to partition large-scale small-world graphs in a scalable manner. We will further detail how it is possible to create and vary label propagation weighting functions to create balanced partitions that minimize total edge cut (EC) and/or maximal per-partition edge cut (EC_{max}). This overall strategy can partition graphs under both single and multiple constraints as well as under single and multiple objectives. It is possible to extend this approach even further to include other objectives, e.g. communication volume, beyond those described below.

9.4.1 Label Propagation

Label propagation was originally proposed as a fast community detection algorithm [145]. An overview of the baseline algorithm is given in Algorithm 9.1. We begin by randomly initializing the labels L for all vertices in the graph out of a possible l distinct labels. l is usually the number of vertices in the graph (each vertex v gets a distinct label, usually its numeric vertex identifier, $Vid(v)$), but it can also be chosen experimentally or based on some heuristic to maximize a community evaluation metric, such as modularity or conductance. Then, for a given vertex v in the set of all vertices V in a graph G, we examine for all of its neighbors u each of their labels $L(u)$. We keep track of the counts for each distinct label in $Counts$. After examining all neighbors, v updates its current label to whichever label has the maximal count in $Counts$ with ties broken randomly.

The algorithm proceeds to iterate over all V until some stopping criterion is met. This stopping criterion is usually some fixed number of iterations $Iter$, as we show, or until convergence is reached and no new updates and performed during a single iteration (number of *updates* is zero). For large graphs, there is no guarantee that convergence will be reached quickly, so a fixed iteration count is usually

Algorithm 9.1 Baseline label propagation algorithm.

```
procedure LABEL-PROP(G(V, E), l, Iter)
    for all v ∈ V do
        L(v) ←Vid(v) or Rand(1 ··· l)
    i ← 0, updates ← 1
    while i < Iter and updates ≠ 0 do
        updates ← 0
        for all v ∈ V do
            Counts(1 ··· l) ← 0
            for all ⟨v, u⟩ ∈ E do
                Counts(L(u)) ← Counts(L(u)) + 1
            x ← GetMax(Counts(1 ··· l))
            if x ≠ L(v) then
                L(v) ← x
                updates ← updates + 1
        i ← i + 1
    return L
```

preferred in practice. As with l, the maximal iteration count is usually determined experimentally. Since each iteration performs linear work with regards to the size of the input graph, this results in an overall linear and efficient algorithm.

9.4.2 PuLP Overview

Algorithm 9.2 PuLP multi-constraint multi-objective algorithm.

```
procedure PuLP-MM(G(V, E), p, Iter_loop, Iter_bal, Iter_ref)
    P ← PuLP-Init(G(V, E), p, P)
    N_vert(1 ··· p) ← vertex counts in P(1 ··· p)
    for i = 1 ··· Iter_loop do
        P ← PuLP-VertBal(G(V, E), p, P, N_vert, Iter_bal)
        P ← PuLP-VertRef(G(V, E), p, P, N_vert, Iter_ref)
    N_edge(1 ··· p) ← edge counts in P(1 ··· p)
    N_cut(1 ··· p) ← edge cuts in P(1 ··· p)
    Cut ← current edge cut
    for i = 1 ··· Iter_loop do
        P ← PuLP-CutBal(G(V, E), p, P, N_vert, N_edge, N_cut, Cut, Iter_bal)
        P ← PuLP-CutRef(G(V, E), p, P, N_vert, N_edge, N_cut, Cut, Iter_ref)
    return P
```

In general, label propagation methods are attractive for community detection due to their low computational overhead, low memory utilization, as well as the relative ease of parallelization. In PuLP, we utilize weighted label propagation in several separate stages to partition an input graph. Algorithm 9.2 gives the overview of the three stages to create a vertex and edge-constrained partitioning that minimizes both edge cut and maximal per-part edge cut. We refer to this algorithm as PuLP Multi-Constraint Multi-Objective partitioning, or **PuLP-MM**. We first initialize the partition using a multi-source BFS with p randomly selected initial roots (PuLP-Init in Algorithm 9.3). The initial (unbalanced) partitioning is then passed to an iterative stage that first balances the number of vertices in each part through weighted label propagation (PuLP-VertBal listed in Algorithm 9.4) while minimizing the edge cut and then improves the edge cut on the balanced partition through a refinement stage motivated by FM-refinement [177] (PuLP-VertRef). The next iterative stage further balances the number of edges per part while minimizing and balancing the per-part edge cut through weighted label propagation (PuLP-CutBal listed in Algorithm 9.6) and then refines the achieved partitions through constrained refinement (PuLP-CutRef as shown in Algorithm 9.7). More details of these stages are in the following subsections. We demonstrate a more quantitative analysis on the sensitivity of results to varying iteration counts and algorithmic changes in our results.

$G(V, E)$	Input graph (undirected, unweighted)
$n = \|V\|$	Number of vertices in graph
$m = \|E\|$	Number of edges in graph
$P(1 \cdots n)$	Per-vertex partition mappings
p	Number of parts
ϵ_u	Vertex upper balance constraint {0.1}
η_u	Edge upper balance constraint {0.1}
$Iter_{loop}$	# of iterations in outer loop {3}
$Iter_{bal}$	# of iterations in balanced propagation stage {5}
$Iter_{ref}$	# of iterations in constrained refinement stage {10}
PuLP-X	PuLP subroutine for various stages

Table 9.1. PuLP inputs, parameters, and subroutines.

The input parameters to PuLP are listed in Table 9.1. Listed in the braces are the default values we used for partitioning the graphs during our experiments. The

vertex and edge balance constraints (ϵ_u and η_u) are selected based on what might be reasonable in practice for a typical graph analytic code running on a small-world graph. The iteration counts we use ($Iter_{loop}$, $Iter_{bal}$,, $Iter_{ref}$) were determined experimentally, as they demonstrated the best trade-off between computation time and partitioning quality across our suite of test graphs for a wide range of tested values.

9.4.3 PuLP Initialization

To first initialize our partitions, we use a randomly-sourced breadth-first search approach similar to graph growing as implemented in prior work [29, 178]. Our breadth-first search initialization approach is demonstrated in Algorithm 9.3. To perform our initialization, we randomly select p initial roots (where p is the number of desired parts) and perform a parallel multi-source level-synchronous BFS from these roots. We use a queue-based approach where Q contains the vertices to be explored on the current level, and Q_n is the queue where vertices are placed as they are discovered for exploration on the next level. The queues are swapped at the end of each level. Vertices discovered as descendants from one of these roots are marked as in the same initial part as that root. We impose no explicit constraints during this stage. We note that an initial (vertex/edge) imbalance is expected to occur, the severity of which is dependent on the randomly selected roots; however, in practice, this problem is observed to be minimal, as the subsequent iterative stages are capable of rebalancing the parts. This approach has resulted in the most consistent and highest quality end partitions across a wide range of tested variants, including ones that enforced loose balance constraints up to 100% imbalance.

Some of the tested variants have included the original random label propagation-based approach [47], doing a multi-source BFS with a variety of loose balance constraints, iteratively performing single-source BFSs through a variety of methods to select the roots, and performing multi-source BFSs with a variety of non-random methods to select the roots. Each of these tested variants had several additional sub-variants; however, the approach given by Algorithm 9.3 resulted in the highest average partition quality in terms of edge cut and max per-part cut across the DIMACS and LAW collections of regular and irregular test graphs with part counts as presented later in our results. This unconstrained BFS method performed

Algorithm 9.3 PuLP BFS-based partition initialization procedure.

```
procedure PuLP-Init(G(V, E), p, P)
    Q ← ∅,Q_n ← ∅
    P(1 ··· |V|) = none
    for i = 1 to p do
        v ←UniqueRand(V)
        P(v) ← i
        Add v to Q
    while Q ≠ ∅ do
        for all v ∈ Q do
            for all (v, u) ∈ E do
                if P(u) = none then
                    P(u) ← P(v)
                    Add u to Q_n
        Q ← ∅
        Swap(Q, Q_n)
    return P
```

approximately 2× better on average than the other implemented BFS methods, and approximately 20× better than the original label propagation-based approach [47] on the regular DIMACs problems. This large improvement against the original approach was the result of the label propagation initialization performing very poorly on the regular graphs and meshes that do not contain any intrinsic community structure, where the resulting partitions ended up being comprised of multiple small and disconnected components with high relative edge cuts. The selected method also benefits in terms of ease and scalability of parallelization.

9.4.4 PuLP Vertex Balancing and Total Edge Cut Minimization

With the initial partitioning, the PuLP-VertBal (Algorithm 9.4) balances the vertex counts between parts to satisfy our original balance constraint. Here, we use degree-weighted label propagation. In lieu of doing the standard label propagation approach of assigning to a given vertex v a label based on the maximal label count, $\text{Max}(C(1 \cdots p))$, of all of its neighbors $\langle v, u \rangle \in E$, we utilize an additional degree weighting by considering the size of the neighborhood of u ($|E(u)|$ in Algorithm 9.4). A vertex v is therefore more likely to take u's label if u has a very large degree. This approach enables creation of dense clusters around the high degree vertices

that are common in small world graphs. Doing as such ends up minimizing edge cut in practice by making it preferential for boundary vertices to be of smaller degree, as larger degree vertices will propagate their label to all of their neighbors in the subsequent iterations. We can generalize this to alternative objectives by considering all $C(1 \cdots p)$ values as a benefit function for moving v to some partition $(1 \cdots p)$. Considering, for example, communication volume, then $C(1 \cdots p)$ would be the reduction in communication volume for moving v to each of $(1 \cdots p)$.

Algorithm 9.4 PuLP single objective vertex-constrained label propagation stage.

```
procedure PuLP-VertBal(G(V,E), P, p, N_vert, I_b)
    i ← 0, updates ← 1
    Max_v ← (n/p) × (1 + ε_u)
    W_v(1 ··· p) ← GetMax(Max_v/N_vert(1 ··· p) − 1,0)
    while i < Iter_bal and updates ≠ 0 do
        updates ← 0
        for all v ∈ V do
            C(1 ··· p) ← 0
            for all ⟨v,u⟩ ∈ E do
                C(P(u)) ← C(P(u)) + |E(u)|
            for j = 1 ··· p do
                if Moving v to P_j violates Max_v then
                    C(j) ← 0
                else
                    C(j) ← C(j) × W_v(j)
            x ← GetMax(C(1 ··· p))
            if x ≠ P(v) then
                Update(N_vert(P(v)), N_vert(x))
                Update(W_v(P(v)), W_v(x))
                P(v) ← x
                updates ← updates + 1
        i ← i + 1
    return P
```

There are two additional changes from baseline label propagation to make note of. First, for any part that is overweight, i.e. the number of vertices in that current part π_q ($N_{vert}(q)$ in the algorithm) is greater than our desired maximal Max_v, we do not allow that part to accept new vertices. Max_v is the maximum number of vertices allowed in a given part, depending on the balance constraint ϵ_u. Second, there is an additional weighting parameter $W_v(1 \cdots p)$ that is based on

how underweight any part currently is. For a given part q, $W_v(q)$ will approach infinity as the size of that part approaches zero and will approach zero as the size of the part approaches Max_v. For part sizes above Max_v, we will consider the weight to be zero. As shown in Algorithm 9.4, we calculate W_v for any given part q as $W_v(q) = Max_v/N_{vert}(q)$, where $N_{vert}(q)$ is the current size of part q in terms of vertices. When $N_{vert}(q) > Max_v$, then $W_v(q) = 0$.

This weighting forces larger parts to give vertices away with a preference towards the current most underweight parts. This stage is still degree-weighted and therefore minimizes the edge cut in the aforementioned indirect way, preferring small degree vertices on the boundary. When none of the parts are overweight and there is little difference in W_v values, this scheme will default to basic degree-weighted label propagation. This weighting-based approach is similar to that employed in related work [166]. A possible drawback we will note is that the only new part assignments any vertex is able to assume are the current part assignments of its neighbors, which might limit the ability of the algorithm to achieve proper balance. However, due to the low diameter and general small-world structural characteristics of the graphs for which PuLP is designed and optimized, subdomain connectivity of partitions is very high (i.e. there usually exists multiple edges between any given pair of q_i,q_j parts). This allows vertices to move fluidly among all parts and vertex balance to be quickly achieved. Each outer loop of Algorithm 9.4 runs in $O(np + m)$, so it is linear in the size of the network for a fixed p. Here, p is the number of parts/labels and n and m and the numbers of vertices and edges in the graph, respectively, .

We further explicitly minimize edge cut with a greedy refinement stage, as given in Algorithm 9.5. The refinement stage iteratively examines boundary vertices and passes them to a new part if it results in a lower edge cut without violating the vertex balance constraint (Max_v). We calculate possible refinements by examining the part assignments of all neighbors of a given vertex v, and if the number of neighbors in an adjacent part is greater than the number of neighbors in v's current part, moving v to the adjacent part will result in an overall lower edge cut. Similarly with an alternative objective such as communication volume, we would greedily move vertices to a new part if it improves upon the objective without violating a constraint. As there is no work dependence on the number of parts during refinement, each iteration of the refinement stages run in $O(n + m)$ time.

We perform $Iter_{loop}$ iterations of balancing (Algorithm 9.4 and refining (Algo-

Algorithm 9.5 PuLP single objective vertex constrained refinement stage.

```
procedure PuLP-VertRef(G(V, E), P, p, N_vert, Iter_ref)
    i ← 0, updates ← 1
    Max_v ← (n/p) × (1 + ε_u)
    while i < Iter_ref and updates ≠ 0 do
        updates ← 0
        for all v ∈ V do
            C(1 ··· p) ← 0
            x ← P(v)
            for all ⟨v, u⟩ ∈ E do
                C(P(u)) ← C(P(u)) + 1
                if C(P(u)) > C(x) then
                    x ← P(u)
            if Moving v to P_x does not violate Max_v then
                P(v) ← x
                Update(N_vert(P(v)), N_vert(x))
                updates ← updates + 1
        i ← i + 1
    return P
```

rithm 9.5) before moving on to the stages with other partitioning objectives and constraints. However, in order to only create a vertex-constrained partitioning with the total edge cut minimized, the algorithm can stop after this stage. We call this PuLP Single-Constraint Single-Objective, or simply **PuLP**. We note that very simple changes to Algorithms 9.4 and 9.5 would allow us to constrain edge balance instead of vertex balance.

9.4.5 PuLP Edge Balancing and EC_{max} Minimization

Once we have a vertex balanced partitioning that minimizes edge cut, PuLP balances edges per part and minimizes per-part edge cut (Algorithm 9.6). The total edge cut might increase because of the new objective, hence the algorithm uses a dynamic weighting scheme to achieve a balance between optimizing for the two different objectives while also refining just for the total edge cut objective. The algorithm also ensures the vertex and edge balance constraints will become satisfied if possible. And while the approach uses weighted label propagation under given constraints (similarly to Algorithm 9.4), there are a number of nuances to make note of.

Algorithm 9.6 PuLP multi-objective vertex and edge-constrained label propagation stage.

procedure PuLP-CutBal($G(V,E), P, p, N_{vert}, N_{edge}, N_{cut}, Cut, Iter_{bal}$)
  $i \leftarrow 0$, $r \leftarrow 1$
  $Max_v \leftarrow (n/p) \times (1 + \epsilon_u)$
  $Max_e \leftarrow (m/p) \times (1 + \eta_u)$
  $Cur_{Max_e} \leftarrow \text{GetMax}(N_{edge}(1 \cdots p))$
  $Cur_{Max_c} \leftarrow \text{GetMax}(N_{cut}(1 \cdots p))$
  $W_e(1 \cdots p) \leftarrow Cur_{Max_e}/N_{edge}(1 \cdots p) - 1$
  $W_c(1 \cdots p) \leftarrow Cur_{Max_c}/N_{cut}(1 \cdots p) - 1$
  $d_{edge} \leftarrow 1$, $d_{cut} \leftarrow 1$
  while $i < Iter_{bal}$ **and** $updates \neq 0$ **do**
    $updates \leftarrow 0$
    for all $v \in V$ **do**
      $C(1 \cdots p) \leftarrow 0$
      for all $\langle v, u \rangle \in E$ **do**
        $C(P(u)) \leftarrow C(P(u)) + 1$
      for $j = 1 \cdots p$ **do**
        if Moving v to P_j violates Max_v, Cur_{Max_e}, Cur_{Max_c} **then**
          $C(j) \leftarrow 0$
        else
          $C(j) \leftarrow C(j) \times (W_e(j) \times d_{edge} + W_v(j) \times d_{cut})$
      $x \leftarrow \text{GetMax}(C(1 \cdots p))$
      if $x \neq P(v)$ **then**
        $P(v) \leftarrow x$
        Update($N_{vert}(P(v)), N_{vert}(x)$)
        Update($N_{edge}(P(v)), N_{edge}(x)$)
        Update($N_{cut}(P(v)), N_{cut}(x)$)
        Update(Cut)
        Update($W_e(P(v)), W_e(x)$)
        Update($W_c(P(v)), W_c(x)$)
        Update(Cur_{Max_e},Cur_{Max_c})
        $updates \leftarrow updates + 1$
    if $Cur_{Max_e} < Max_e$ **then**
      $Cur_{Max_e} \leftarrow Max_e$
      $d_{cut} \leftarrow d_{cut} \times Cur_{Max_c}$
      $d_{edge} \leftarrow 1$
    else
      $d_{edge} \leftarrow d_{edge} \times (Cur_{Max_e}/Max_e)$
      $d_{cut} \leftarrow 1$
    $i \leftarrow i + 1$
  return P

Initially, we do not use the given edge balance constraint explicitly. Instead, a relaxed constraint based on the current maximal edge count across all parts Cur_{Max_e} is used to compute the edge balance weights ($W_e(1 \cdots p)$). The edge balance weights are similar to the vertex balance weights ($W_v(1 \cdots q)$), in that these per-

part weighting values increase inversely proportionally with how underweight each part is relative to the current maximum (calculated as $W_e(q) = Cur_{Max_e}/N_{edge}(q)$). This results in the possibility of all parts receiving more edges with the exception of which part is currently the largest, but no part will receive enough edges to become greater than Cur_{Max_e}. As the largest part can only give away vertices and edges, Cur_{Max_e} is iteratively tightened until the given edge balance constraint is met. Once we pass the threshold given by our input constraint, we fix Cur_{Max_e} to be equal to Max_e. To minimize the maximum edges cut per-part, we employ a similar procedure with Cur_{Max_c} and the weightings for maximum cut edges ($W_c(1 \cdots p)$). We iteratively tighten this bound so that, although we have no assurance that the global edge cut will decrease, we will always be decreasing the maximal edges cut per-part.

We also introduce two additional dynamic weighting terms d_{edge} and d_{cut} that serve to shift the focus of the algorithm between hitting the Max_e constraint and minimizing Cur_{Max_c}. For every iteration of the algorithm that the Max_e constraint is not satisfied, d_{edge} is increased by the ratio of which Cur_{Max_e} is greater than Max_e. This shifts the weighting function to give higher preference towards moving vertices to parts with low edge counts instead of attempting to minimize the edge cut balance. Likewise, when the edge balance constraint is satisfied, we reset d_{edge} to one and iteratively increase d_{cut} to now focus the algorithm on minimizing maximal per-part edge cut.

This iterative approach with different stages works much better in practice for multiple constraints, as employing two explicit constraints at the beginning is a very tough problem. The label propagation will often get stuck, unable to find any vertices that can be moved without violating either constraint. Note that we can very easily turn the problem in a multi-constraint single-objective problem by not including Cur_{Max_c} and W_c in our weighting function or constraint checks. We demonstrate this later in Section 9.5 by running PuLP Multi-Constraint Single-Objective, or **PuLP-M**. Additionally, we can instead turn the problem into a single-constraint three-objective problem by ignoring Max_e altogether and instead just attempt to further minimize both Cur_{Max_e} and Cur_{Max_c} along with total edge cut. Finally, we would be able to completely generalize this algorithm into an arbitrary number of q constraints. We would do this by calculating $Max_{1 \cdots q}$, $Cur_{Max_{1 \cdots q}}$, and $W_{1 \cdots q}(1 \cdots p)$ for each of the q constraints, progressively increasing

one of $d_{1\cdots q}$ while setting the others to 1, and then using these to calculate the additional weightings to $C(1\cdots p)$ as $C(1\cdots p) \leftarrow C(1\cdots p) \times \sum_{k=1}^{q} W_k(1\cdots p) * d_k$. We can also again adjust the objective by altering how we initialize the $C(1\cdots p)$ array for each v. As with Algorithm 9.4, each iteration of Algorithm 9.6 also runs in $O(np+m)$.

Algorithm 9.7 PuLP multi-objective vertex and edge-constrained refinement stage.

```
procedure PuLP-CutRef(G(V,E), P, p, N_vert, N_edge, N_cut, Cut, Iter_ref)
    i ← 0, updates ← 1
    Max_v ← (n/p) × (1 + ε_u)
    Max_e ← (m/p) × (1 + η_u)
    Cur_Max_e ← GetMax(GetMax(N_edge(1···p)),Max_e)
    Cur_Max_c ← GetMax(N_cut(1···p))
    while i < Iter_ref and updates ≠ 0 do
        updates ← 0
        for all v ∈ V do
            C(1···p) ← 0
            x ← P(v))
            for all ⟨v,u⟩ ∈ E do
                C(P(u)) ← C(P(u)) + 1
                if C(P(u)) > C(x) then
                    x ← P(u)
            if Moving v to P_x does not violate Max_v, Cur_Max_e, Cur_Max_c then
                P(v) ← x
                Update(N_vert(P(v)), N_vert(x))
                Update(N_edge(P(v)), N_edge(x))
                Update(N_cut(P(v)), N_cut(x))
                Update(Cut)
                updates ← updates + 1
        i ← i + 1
    return P
```

After the completion of Algorithm 9.6, we again perform a constrained refinement, given by Algorithm 9.7. This algorithm uses the current maximal balance sizes of Max_v, Cur_{Max_e}, and Cur_{Max_c}, and we again attempt to find a local minimum for the total edge cut without violating any of these current balances. Although we have a hard cutoff for the number of outer loop iterations $Iter_{loop}$ shown in Algorithm 9.2, in practice, we continue to iterate between Algorithm 9.6

and Algorithm 9.7 if our secondary constraint has not been achieved and progress is still being made towards reaching it.

9.4.6 Algorithm Parallelization and Optimization

One of the strengths of using label propagation for partitioning is that its vertex-centric nature lends itself towards very straightforward and efficient parallelization. For all of our listed label propagation-based and refinement algorithms, we implement shared-memory parallelization over the primary outer loop of all $v \in V$. Max_v, Cur_{Max_e}, Cur_{Max_c}, d_{edge}, and d_{cut} as well as N_{vert}, N_{edge}, and N_{cut} are all global values and arrays and are updated in a thread-safe manner. Each thread creates and updates its own C, W_v, W_e, and W_c counting and weighting arrays.

The algorithm also uses global and thread-owned queues as well as boolean *in queue* arrays to speed up label propagation through employing an approach similar to what can be used for *color propagation* [21]. This technique avoids having to examine all $v \in V$ in every iteration. Although it is possible, because of the dynamic weighting functions, that a vertex doesn't end up enqueued when it is desirable for it to change parts on a subsequent iteration, the effects of this are observed to be minimal in practice. We observe near identical quality between both our queue and non-queue implementations as well as our serial and parallel code.

9.5 Results and Discussion

9.5.1 Experimental Setup

We evaluate performance of our new PuLP partitioning strategies on the small-world networks comprising the Laboratory for Web Algorithmics (LAW) group from the University of Florida Sparse Matrix Collection [74,75,179,180]. Additional tests were performed on the graphs comprising the test suite from the 10th DIMACS Implementation Challenge [27]. The LAW graphs and their global properties are listed in Table 9.2. We omit listing the DIMACS graphs for brevity. We preprocessed the graphs before partitioning by removing directivity in edges, deleting all degree-0 vertices and multi-edges, and extracted the largest connected component. Ignoring I/O, this preprocessing required minimal computational time, only on the order

seconds in serial for each of the datasets. Table 9.2 lists the sizes and properties of the LAW graphs after preprocessing. To determine approximate diameter, we perform iterative breadth-first searches rooted at a randomly-selected vertex from the farthest level of the previous search. We return the maximum number of levels as determined by the BFSs after it doesn't increase in ten consecutive iterations.

Network	n	m	$davg$	$dmax$	$\widetilde{D}$
enron	68 K	253 K	7.5	1.6 K	11
dblp	226 K	716 K	6.3	238	17
amazon	735 K	3.5 M	9.6	1.1 K	23
ljournal	5.4 M	50 M	18	19 K	28
hollywood	1.1 M	56 M	105	11 K	10
cnr	326 K	2.7 M	17	18 K	35
in	1.4 M	131 M	19	22 K	44
indochina	7.3 M	149 M	41	256 K	27
eu	862 K	161 M	37	69 K	22
uk-2002	18 M	261 M	28	195 K	28
arabic	23 M	552 M	49	576 K	48
uk-2005	39 M	781 M	40	1.8 M	21
webbase	113 M	845 M	15	816 K	375
it	41 M	1.0 B	50	1.3 M	26
sk	51 M	1.8 B	72	8.6 M	308

Table 9.2. Test graph characteristics *after* preprocessing. # Vertices (n), # Edges (m), average ($davg$) and max ($dmax$) vertex degrees, and approximate diameter ($\widetilde{D}$) are listed. The bottom ten graphs are all web crawls, while the top five are of various types. $B = \times 10^9$, $M = \times 10^6$, $K = \times 10^3$.

Scalability and performance studies were done on the *Compton* testbed cluster at Sandia National Labs. The executables were built with the Intel C++ compiler (version 13) using OpenMP for multithreading and the `-O3` option.

9.5.2 Performance Evaluation

We evaluate our PuLP partitioning methodology against both single and multi-constraint METIS (v5.1.0) and ParMETIS (v4.0.3) as well as KaFFPa from KaHIP (v0.71b). METIS runs used k-way partitioning with sorted heavy-edge matching and minimized edge cut. KaFFPa results use the *fastsocial* option (KaFFPa-FS),

which does constrained label propagation during the initial graph contraction phase. KaFFPa allows for constraints on either vertices or edges; however, using multiple constraints concurrently is not possible with the current version. METIS allows for multi-constraint partitioning (METIS-M in the results). KaFFPa was unable to process the sk graph due to a 32-bit `int` limitation. A number of experiments were unable to complete for METIS and ParMETIS due to memory limitations.

We use the three aforementioned variants of PuLP for comparison, which are single-constraint single-objective (PuLP), multi-constraint single-objective (PuLP-M), and multi-constraint multi-objective (PuLP-MM). We do comparisons on the basis of edge cut, maximal per-part edge cut, execution time, and memory utilization. For all experiments on the LAW networks, the vertex imbalance ratio is set to 10%. For multi-constraint experiments, the maximal number of edges per-part for each graph is set to the maximum of either 10% imbalance or 4× the number of edges of the highest degree vertex of the graph. The rationale for loosening the edge balance constraints is due to the existence of very high degree vertices in certain graphs, which make a fixed constraint impossible to achieve for higher part counts. We set a relatively high vertex imbalance due to how the work requirements for a lot of graph computations are partially dependent on per-part edges, so achieving a better edge balance at a small cost to vertex imbalance can result in better overall work balance in practice.

9.5.3 Execution Time and Memory Utilization

We first compare PuLP to METIS-M, ParMETIS, and KaFFPa-FS in terms of partitioning times and memory utilization. Table 9.3 gives the serial execution times for computing 32 parts with PuLP-MM, METIS-M, and KaFFPa-FS, as well as the parallel execution times for PuLP-MM and ParMETIS. We also give the speedup of serial and parallel PuLP-MM relative to the fastest serial code as well as the fastest code overall. Note that the execution time given for ParMETIS for each graph was selected as the fastest from 1 to 16 nodes on Compton with 1 to 16 tasks per node (up to 256 total cores). In effect, the speedups we give to PuLP relative to ParMETIS are extremely conservative. Also note that we are comparing against one of the "fast" variants in KaFFPa.

Overall, we observe a geometric mean speedup of 1.07× relative to the next

Network	Execution time (s)					PuLP-MM Speedup	
	Serial			Parallel		vs Best	All
	PuLP-MM	METIS-M	KaFFPa-FS	PuLP-MM	ParMETIS	Serial	Parallel
enron	0.38	0.72	1.18	0.22	0.32	1.89×	1.45×
dblp	0.67	1.23	3.18	0.23	0.34	1.84×	1.48×
amazon	2.70	3.97	11.7	0.54	0.77	1.47×	1.43×
hollywood	41.5	47.9	189	3.88	22.0	1.15×	5.67×
ljournal	119	116	206	12.2	47.6	0.97×	3.90×
cnr	1.56	1.63	2.15	0.44	3.11	1.04×	3.73×
in	6.17	2.9	7.03	0.89	2.6	0.47×	2.92×
indochina	75.7	60.2	52.3	9.74	120	0.69×	5.37×
eu	8.22	6.82	12.0	1.23	9.93	0.83×	5.54×
uk-2002	82.9	67.6	128	7.65	77.1	0.82×	8.84×
arabic	147	129	185	13.4		0.88×	9.63×
uk-2005	336		439	34.9		1.31×	12.6 ×
webbase	521		831	45.2		1.60×	18.4 ×
it	364		409	28.6		1.12×	14.3 ×
sk	644			54.6			

Table 9.3. Comparison of execution time of serial and parallel (16 cores) PuLP-MM algorithm with serial METIS-M, KaFFPa-FS, ParMETIS (best of 1 to 256 cores), computing 32 parts. The "All" speedup compares parallel PuLP-MM to the best of the rest.

fastest of METIS-M and KaFFPa-FS for our serial code across the entire set of LAW graphs. This relatively modest speedup in serial comes from the fact that, although all PuLP variants run in $O(np + m)$, that order of work is performed on a per-iteration basis for up to the 90 iterations of PuLP-MM. This gives a large initial constant factor for execution time. The primary benefits for a single level label propagation-based approach comes from the ease of and efficiency of parallelization as well as the low memory overhead. For our parallel code, we note a mean speedup of 5.0×. Parallel speedups are also observed to increase with increasing graph size, likely a result of the improved scalability of label propagation versus the multilevel approaches.

Table 9.4 compares the maximal memory utilization of PuLP-MM, METIS-M and KaFFPa-FS for computing 32 parts. Memory savings for PuLP versus the best of either METIS-M or KaFFPa-FS are significant (3.0× geometric mean). We note an increase in memory savings with increasing graph size. These memory savings are primarily due to avoiding a multilevel approach. The only structures PuLP needs (in addition to graph storage) are the global array of length n to store the partition mappings; the vertex, edge count, and cut count arrays each of length p; and the thread-owned weight arrays also each of length p. The storage cost

Network	Memory Utilization				**Improv.**
	METIS-M	KaFFPa-FS	PuLP-MM	Graph Size	
enron	50 MB	33 MB	66 MB	2.5 MB	0.5×
dblp	91 MB	83 MB	67 MB	7.4 MB	1.2×
amazon	482 MB	385 MB	106 MB	33 MB	3.6×
ljournal	10 GB	4.8 GB	616 MB	429 MB	7.8×
hollywood	7.5 GB	3.8 GB	534 MB	448 MB	7.0×
cnr	285 MB	143 MB	105 MB	24 MB	1.4×
in	1.2 GB	537 MB	208 MB	113 MB	2.6×
indochina	20 GB	5.3 GB	1.4 GB	1.2 GB	3.8×
eu	1.6 GB	786 MB	217 MB	133 MB	3.6×
uk-2002	23 GB	9.8 GB	2.6 GB	2.2 GB	3.8×
arabic	33 GB	19 GB	5.1 GB	4.5 GB	3.7×
uk-2005	-	30 GB	7.4 GB	6.4 GB	4.1×
webbase	-	38 GB	9.8 GB	7.5 GB	3.9×
it	-	35 GB	9.4 GB	8.3 GB	3.7×
sk	-	-	16 GB	15 GB	-

Table 9.4. PuLP efficiency: Maximum memory utilization comparisons for generating 32 parts.

for all p length arrays is insignificant with a modest thread and part count. We additionally utilize a few more n length integer and boolean arrays as well as smaller thread-owned queues and arrays to speed up label propagation, as mentioned in Section 9.4.

We also plot the scalability of our PuLP codes versus METIS-M, ParMETIS, and KaFFPa-FS relative to increasing the parallelization and increasing the number of parts being computed. We analyze the effects of each on execution time and memory consumption on four selected test graphs (enron, hollywood, ljournal, and uk-2002), as is given by Figure 9.1. The top two plots give the effect on execution time and memory consumption when going from one to sixteen-way parallelism. For this test, ParMETIS was run with sixteen tasks on either one node (when possible) or sixteen nodes, with the lower execution time being reported. The times and memory for serial METIS-M and KaFFPa-FS are plotted as flat lines for comparison. The bottom two plots of Figure 9.1 show the effect on execution time and memory consumption when increasing the number of parts being computed from 2 to 1024 when running in serial for METIS-M and KaFFPa-FS and with

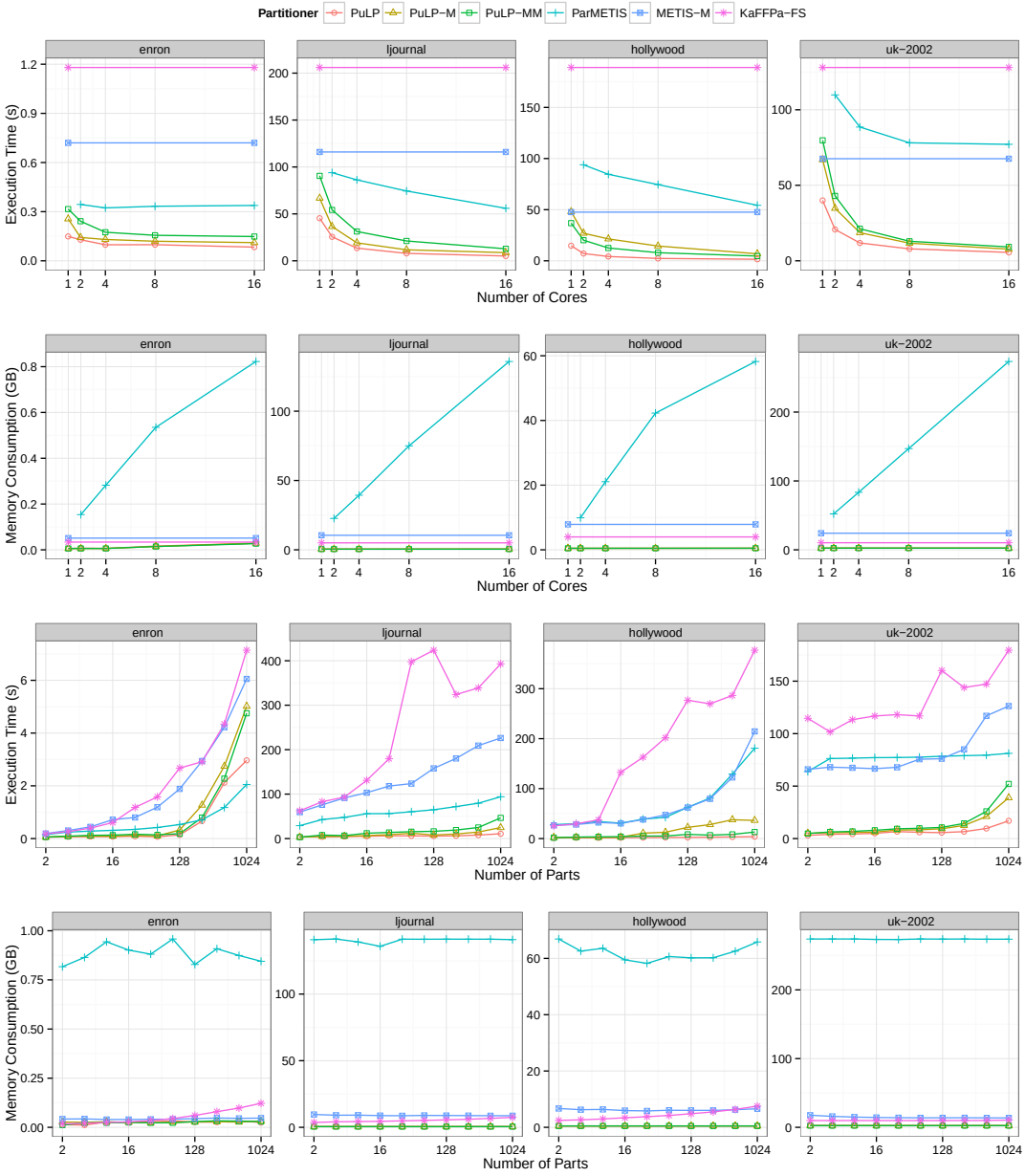

Figure 9.1. Scaling for each partitioner in terms of execution time versus number of cores (top), total memory utilization versus number of cores (2nd from top), execution time versus number of computed parts (3rd from top), and memory utilization versus number of computer parts (bottom).

sixteen-way parallelism with the PuLP codes and ParMETIS.

We observe consistent strong scaling across all of our implementations in Figure 9.1, with a geometric mean speedup of 5.7× for sixteen cores across all graphs

and the three PuLP algorithms. We note our code strong scales better on these instances than ParMETIS, which has a maximum speedup of less than 2×. We also note that our memory requirements increase minimally with an increasing numbers of threads/cores, while the memory load of ParMETIS increases almost linearly with number of tasks. Throughout these tests, we also noted that there is no edge cut cost for increasing parallelism, at least up to 16 cores. This is due to all per-vertex update decisions being made by each thread use globally synchronized information.

When computing an increasing number of parts, we note that our execution time plots are generally flat from 2 parts up to about 128 parts, where times begin to increase. We note a rather large increase for all codes when partitioning enron. Analyzing the computation of the PuLP algorithms, it appears the likely cause is the increasing difficulty to create balanced partitions with larger part counts, as enron is one of the smallest test cases. The bottom plot of Figure 9.1 gives the memory requirements versus part count for the various partitioners. For the PuLP and METIS variants, the curves are close to flat. For KaFFPa, we note a slight dependence of memory requirement on part count.

9.5.4 Edge Cut and Maximal Per-Part Edge Cut

Figure 9.2 (top) compares the quality of the computed partitions from PuLP and METIS with the LAW test graphs for 2 to 1024 parts using multiple constraints for both programs and minimizing the total edge cut. We report the median value obtained over 5 experiments for each part count and method. We omit comparison to KaFFPa-FS as it is unable to generate partitions that satisfy the multiple concurrent constraints.

The top plots of Figure 9.2 show the edge cuts (EC) obtained for multi-constraint METIS (METIS-M) as well as both multi-constraint (PuLP-M) and multi-constraint multi-objective PuLP (PuLP-MM) with varying numbers of parts. Figure 9.2 (bottom) gives the maximal per-part edge cut (EC_{max}) as a ratio of total edges scaled by the number of parts (scaling was done for visualization purposes). For all plots, a lower value indicates a higher quality partitioning. We additionally show Table 9.5, where we quantify a performance metric in terms of edge cut (and max per-part cut, in the three right columns) divided by the best edge cut

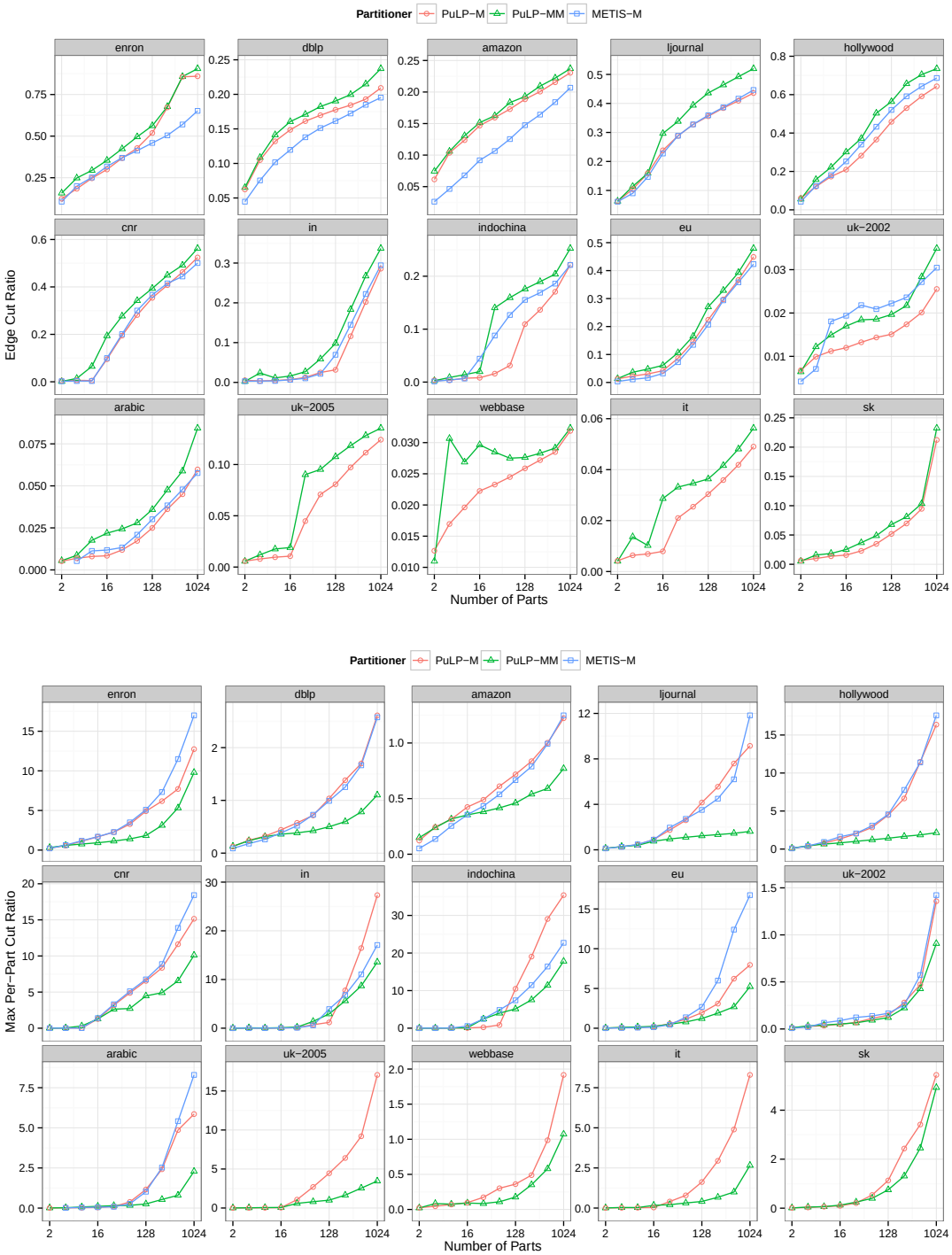

Figure 9.2. Quality metrics of total cut edge ratio (top) and scaled maximum per-part edge cut ratio (bottom) for PuLP-M, PuLP-MM and METIS-M.

Network	Edge Cut			Max Per-Part Cut		
	PuLP-M	PuLP-MM	METIS-M	PuLP-M	PuLP-MM	METIS-M
enron	1.08	1.26	**1.02**	1.72	**1.06**	1.59
dblp	1.18	1.27	**1.00**	1.68	**1.09**	1.48
amazon	1.52	1.59	**1.00**	1.54	**1.20**	1.23
ljournal	1.03	1.17	**1.00**	2.05	**1.01**	2.15
hollywood	**1.04**	1.28	1.11	2.44	**1.05**	2.56
cnr	1.28	1.86	**1.07**	1.63	1.55	**1.46**
in	1.17	2.22	**1.15**	1.46	2.06	**1.23**
indochina	**1.22**	2.29	1.76	**1.55**	2.05	2.26
eu	1.39	1.72	**1.00**	1.73	**1.47**	1.68
uk-2002	**1.08**	1.40	1.35	1.21	**1.12**	1.41
arabic	**1.03**	1.62	1.16	1.97	**1.35**	2.31
uk-2005	**1.00**	1.39	-	2.12	**1.15**	-
webbase	**1.02**	1.20	-	1.48	**1.08**	-
it	**1.00**	1.46	-	2.05	**1.31**	-
sk	**1.02**	1.30	-	1.22	**1.09**	-
Small	1.17	1.57	**1.13**	1.71	**1.32**	1.71
All	**1.12**	1.50	-	1.70	**1.27**	-

Table 9.5. Performance for each partitioner and graph as the geometric mean of the ratio of produced edge cut (EC) and max per-part cut (EC_{max}) relative to the best for across each network and number of generated parts.

(and max per-part cut) out of all partitioners for each "network-number of parts" combination. For each partitioner, we report the geometric mean for each graph across all numbers of parts. We also report an overall mean across all graph and number of parts combinations (All), as well as the overall mean for only the smaller graphs that METIS-M was able to successfully partition (Small). Again for this metric, lower is better, with a minimum score of 1.0 indicating that the partitioner with that score produced the best partitions for all test cases. Taken together, the top and bottom plots of Figure 9.2 and the left and right three columns of Table 9.5 demonstrate the tradeoff offered by PuLP-M and PuLP-MM to minimize either the total edge cut at a cost of maximal per-part edge cut or to minimize the maximal per-part edge cut at a cost of total edge cut.

Overall, we observe in Figure 9.2 (top) and Table 9.5 that PuLP-M does better than METIS-M for hollywood, indochina, uk-2002, and arabic; almost as good

Network	PuLP-MM		METIS-M		% Improvement	
	EC	EC_{max}	EC	EC_{max}	EC	EC_{max}
enron	217 K	2.6 K	144 K	5.7 K	-34%	116%
dblp	154 K	1.1 K	132 K	2.3 K	-14%	113%
amazon	783 K	4.1 K	648 K	6.8 K	-17%	68%
ljournal	24 M	140 K	21 M	600 M	-15%	328%
hollywood	40 M	205 K	36 M	1.3 M	-9%	513%
cnr	1.3 M	35 K	1.2 M	74 K	-10%	112%
in	3.5 M	223 K	2.9 M	283 K	-17%	27%
indochina	30 M	3.3 M	28 M	4.8 M	-9%	44%
eu	6.3 M	85 K	5.8 M	391 K	-9%	361%
uk-2002	7.4 M	218 K	7.1 M	293 M	-4%	34%
arabic	32 M	860 K	26 M	5.8 M	-19%	579%
uk-2005	100 M	3.9 M	-	-	-	-
webbase	25 M	957 K	-	-	-	-
it	49 M	2.0 M	-	-	-	-
sk	187 M	8.7 M	-	-	-	-

Table 9.6. Comparison of the two quality metrics, EC and EC_{max} for PuLP-MM and METIS-M when computing 512 parts. The % improvement shows relative improvement in quality for PuLP-MM with respect to METIS-M quality.

as METIS-M for enron, ljournal, in; and worse on dblp, amazon, eu, and cnr in terms of total edge cuts. Over all graphs that METIS-M was able to partition, PuLP-M has a slightly worse edge cut on average. PuLP-MM does worse than METIS-M and PuLP-M in most instances for the edge cut metric but results in much better partitions in terms of the maximal per-part edge cut on all but three test graphs (cnr, in, and indochina), as shown in Figure 9.2 (bottom) and the right three columns of Table 9.5. When the single objective partitioners (METIS-M and PuLP-M) out-perform multi-objective PuLP-MM, it can be explained by the fact that the lower total edge cut for the single objective partitioners effectively results in a lower max cut, even if the cuts aren't as relatively balanced as they are with PuLP-MM. We also observe that the benefit of max per-part minimization increases with an increasing number of parts. A consistent improvement over METIS-M and PuLP-M is noted with part counts greater than about 16.

As mentioned, Figure 9.2 and Table 9.5 demonstrate that multi-objective PuLP can be relatively effective at minimizing the maximal per-part edge cut on partitions derived from these graphs at a nominal cost to total edge cut. Table 9.6 shows

this tradeoff explicitly between PuLP-MM and METIS-M when partitioning each graph into 512 parts. We compare the quality of both the metrics, EC and EC_{max}, and observe that PuLP-MM improves EC_{max} substantially (up to almost 600% improvement) when compared with METIS-M. This is at a cost of only 4-34% increase in total edge cut. Note that while we have been using METIS-M for comparison, it does not explicitly attempt to minimize edge cut balance.

Additionally, we present Figure 9.3, which shows for the ljournal graph the differences in execution time (left), edge cut (middle), and max per-part cut (right) for computing 2 to 1024 parts with PuLP, PuLP-M, and PuLP-MM. These plots demonstrate the effects on performance and quality with the increasing complexity of objectives and constraints on a representative test instance. In general, the multi-constraint PuLP-M and PuLP-MM run at least 2× slower than PuLP, as they perform twice as many total label propagation iterations due to the secondary constraint balancing and refinement stages. Although PuLP-MM performs the same number of total iterations as PuLP-M, we see another relative doubling of execution time. This is a result of the increasing difficulty involved with the secondary (max per-part cut) objective, which consistently results in more work as a result of more active boundary vertices in the queue on each iteration.

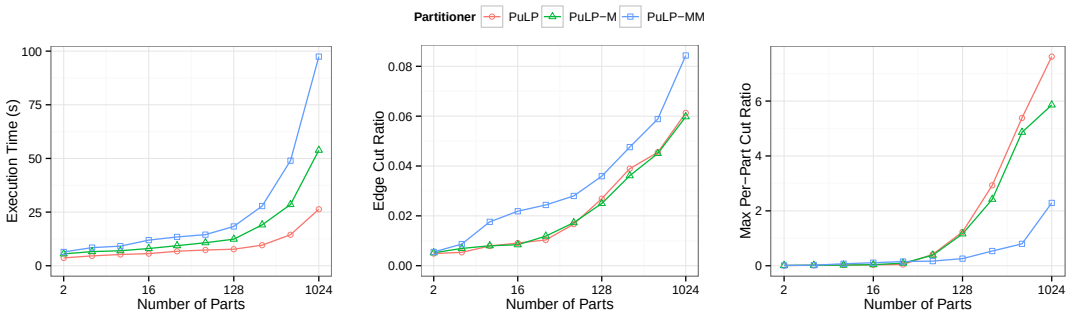

Figure 9.3. Comparison of PuLP, PuLP-M, and PuLP-MM with regards to execution time, edge cut, and max per-part cut to demonstrate the effects of more complex constraints and objectives on execution.

For the edge cut objective, we note a similar difference between PuLP-M and PuLP-MM as observed before. PuLP-MM has a slightly worse global edge cut as a result of the tradeoff involved with additionally optimizing the max per-part cut. PuLP and PuLP-M generally show equivalent edge cuts, as they're both only optimizing for this metric, and therefore both asymptotically approach what can

be considered a relative "lower bound" for the general approach. As we'll show in the next subsection, performing considerably more iterations can improve the objective, but only by a few percent at most. In terms of the max per-part cut as given by the right plot, PuLP-MM shows considerably better quality, as expected, with the relative performance improving with increasing part count. PuLP-M shows moderately better quality for PuLP with this objective, most likely due to a slight increase in cut balance incidentally resulting from balancing for the secondary (edges per part) constraint.

9.5.5 Justification for Algorithmic Choices

There were several small algorithmic details noted in the earlier description of the PuLP algorithms. We ran a set of tests examining the impact of these details and variations on partition quality. Table 9.7 gives the percentage improvement in quality relative to PuLP-MM with these variations. Tested variants include using the original BigData 2014 [47] random label propagation-based initialization procedure (*BigData*), not running any refinement (*NoRefine*), not using degree weighted label propagation during the vertex balance stage (*NoDegWgt*), only running a single iteration for the outer loops instead of three (*LessIter*), and running ten iterations of the outer loops (*MoreIter*). We look at the global averages across all LAW networks and part counts as well as a few select representative instances. A negative value indicates lower quality was produced with the given variant. We ran each "number of parts-network" test case five times and report the geometric mean.

From Table 9.7, we note that running refinement has the largest and most consistent impact on partition quality out of the tested variants. On average, not performing refinement resulted in 64% higher edge cuts and 37% higher max per-part cuts. Degree-weighted label propagation for the vertex balancing stage is especially important for the irregular web crawls (uk-2002 and it), but shows little quality benefit on the other network types (enron communication network and amazon co-purchasing network). As mentioned, this is because the benefits of weighted label propagation are partially dependent on the existence of a skewed degree distribution and an intrinsic community structure.

Table 9.7 also gives insight into our selected number of outer loop iterations.

Percent Improvement for Edge Cut						
Network	# Parts	BigData	NoRefine	NoDegWgt	LessIter	MoreIter
enron	16	-1%	-33%	-2%	-7%	3%
amazon	32	-38%	-37%	-3%	-5%	6%
uk-2002	8	-19%	-214%	-46%	-50%	5%
it	256	-1%	-51%	-15%	-3%	3%
Global	2-1024	-6%	-64%	-17%	-8%	3%
Percent Improvement for Max Per-Part Cut						
Network	# Parts	BigData	NoRefine	NoDegWgt	LessIter	MoreIter
enron	16	3%	-7%	8%	-8%	6%
amazon	32	-20%	-21%	-3%	-6%	4%
uk-2002	8	-5%	-180%	-43%	-104%	7%
it	256	0%	-1%	-1%	-7%	4%
Global	2-1024	-1%	-37%	-13%	-15%	5%

Table 9.7. Comparison of the multiple variants of algorithmic choices on quality in terms of edge cut (top) and max per-part cut (bottom) relative to PuLP-MM.

We observed that three iterations is approximately the point at which diminishing returns on partition quality becomes realized for most test instances. This can be noted by comparing the edge cut and max per-part cut differences to baseline in the *LessIter* and *MoreIter* columns. While the difference between a single iteration (*LessIter*) and the baseline three iterations can result in a quality improvement of over 100% on certain test instances, running up to ten iterations (*MoreIter*) only improves quality by a few additional percent at most. Although running more total outer loop iterations consistently improves partition quality, the considerably higher computation cost doesn't necessarily justify such a minor improvement.

To further demonstrate the effect of iterations on various metrics, we also plot how the edge cut, max per-part cut, vertex imbalance, and edge imbalance changes on a per-iteration basis for a run of PuLP-MM computing 64 parts on the amazon network. We give these plots in Figure 9.4. The thin vertical lines indicate a switch from the balancing to refinement stages, and the thicker vertical lines indicate the beginning of both the vertex balancing/refinement outer loop and the edge and max per-part cut balancing/refinement outer loop as given by Algorithm 9.2. The horizontal lines on the bottom two plots indicate the given 10% imbalance constraints for vertices and edges. As we had shown in Table 9.1, there's 3 iterations

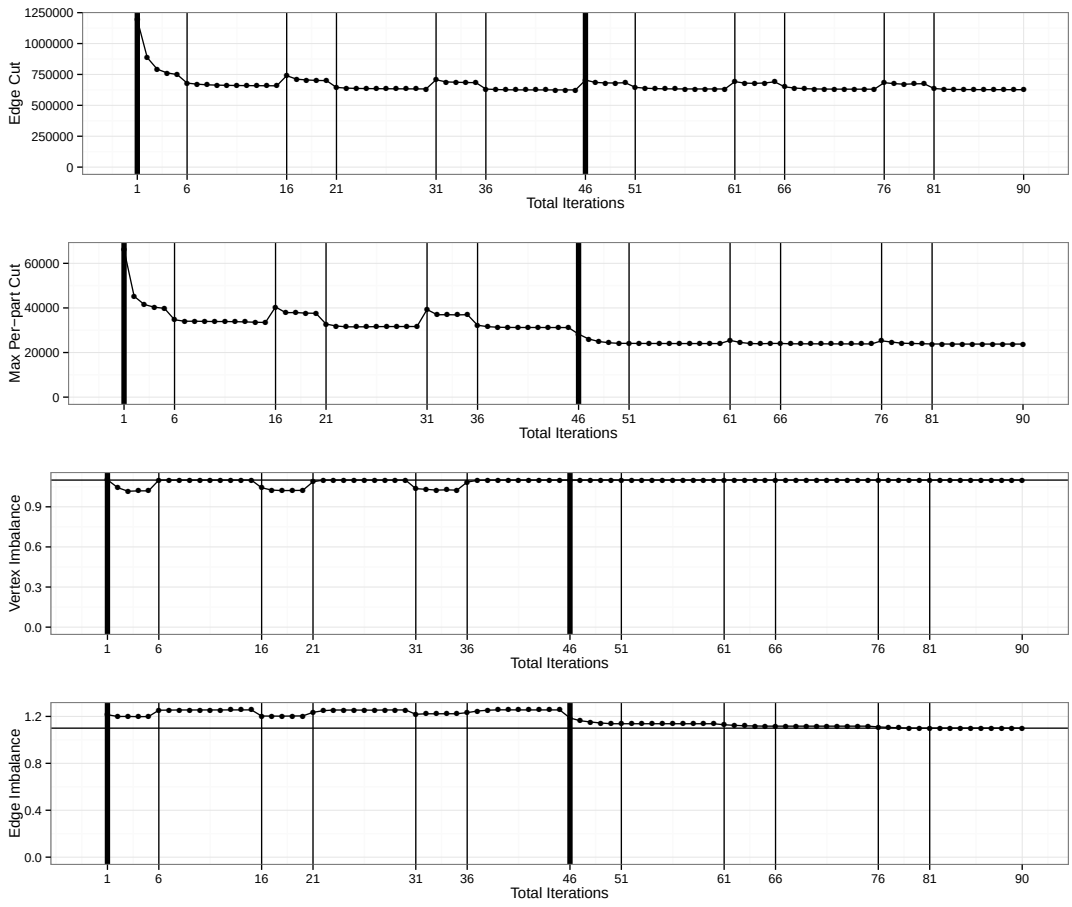

Figure 9.4. Per-iteration performance of PuLP-MM in terms of total edge cut, max per-part edge cut, vertex imbalance, and edge imbalance when computing 64 parts of amazon.

for each outer loop with each balancing stage running for 5 iterations and each refinement stage running for 10 iterations, resulting in 90 total iterations. We consider the plots in Figure 9.4 to be a representative instance for our test networks. In general, we observe that the vertex balance is achieved in few iterations on most networks. Edge balance is slower to achieve, although this is more highly-dependent on the given imbalance ratio and the topology of the network. We note a modest tradeoff in total edge cut and max per-part cut as the edge balance constraint is reached. As was observed by examining the sum affects of greater and fewer iterations with the Global row in Table 9.7, we observe the edge cut and max per-part cut are within a modest fraction of their minimal values after a single balancing and refinement stage. We also note consistent improvements in edge cut

through the refinement iterations, although these improvements are slight enough to not be easily visible with the given scale of the plots. With a looser constraint, we observe correspondingly lower edge cuts and max per-part cuts.

9.5.6 Re-balancing Single Constraint Single Objective Partitions

Another benefit to using a single level label propagation-based partitioner is the ease with which it can be used on a given input partition to re-balance it for additional objectives and constraints. To demonstrate, we used KaFFPa-FS to compute high quality single-objective and single-constraint partitions for all of the LAW graphs across 2-1024 parts. We then used the computed partitions as inputs to PuLP-MM's second loop, the edge balancing and max per-part cut minimization stage. Figure 9.5 gives plots of amazon (top) and webbase (bottom) for edge cut (left) and max per-part cut (right) versus number of parts.

From Figure 9.5, we can observe considerable improvement in terms of the edge cut metric upon our baseline PuLP-MM partition when first using a high quality single-objective and single-constraint partitioning. Additional improvements are noted with the max per-part cut objective as well. Note that while the edge cut is higher than just KaFFPa-FS, the KaFFPa-FS partitions violate the given balance constraints. Overall, we calculated a geometric mean improvement relative to just running PuLP-MM of 22% for edge cut and 17% for max per-part cut across all tested graphs and numbers of parts. Although we don't explicitly perform such an analysis, this approach is also directly applicable to using PuLP to re-partition/re-balance a dynamically changing graph.

9.5.7 DIMACS 10th Implementation Challenge Comparison

While our PuLP approaches were specifically designed for irregular small-world graphs and the multi-objective multi-constraint scenario, we also observe relatively good performance for single-objective single-constraint partitioning of regular networks. To examine the performance of PuLP relative to state-of-the-art, we used PuLP to partition all 90 instances for the DIMACS10 test suite and compared results to both KaFFPa, with both *fast* (KaFFPa-F) and *fastsocial* (KaFFPa-FS) variants, and METIS. We only run the initialization and vertex balancing/refinement stages for PuLP. We again calculated a performance ratio as reported for

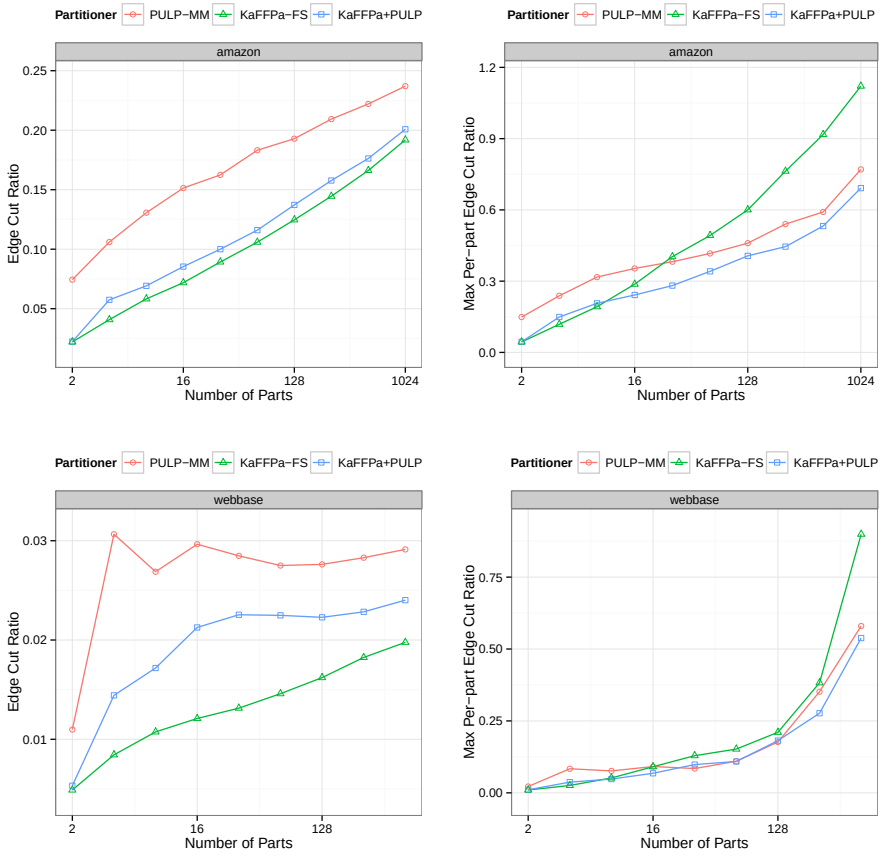

Figure 9.5. Using PuLP-MM to re-partition single objective single constraint partitions computed with KaFFPa-FS. Shown are graphs amazon (top) and webbase (bottom) plotted with edge cut (left) and max per-part cut (right) versus number of parts.

the LAW graphs in Table 9.5, and we report the overall geometric mean across the 90 DIMACS10 test instances. Our tests showed METIS to be the highest scoring partitioner with a ratio of 1.005. KaFFPa-FS had a ratio of 1.053, KaFFPa-F had a ratio of 1.14, and PuLP had a ratio of 1.43. This indicates that, although PuLP was designed for very different partitioning circumstances, the general approach is adaptable enough to still perform within a small factor of the state-of-the-art for other scenarios.

Figure 9.6 gives the edge cut ratio versus number of parts for PuLP, METIS, KaFFPa-F, and KaFFPa-FS on a few test instances from the DIMACS Implementation Challenge. The number of parts chosen for each graph instance were those used in the DIMACS challenge, and are select multiples of 2 from 2 to 1024 as can

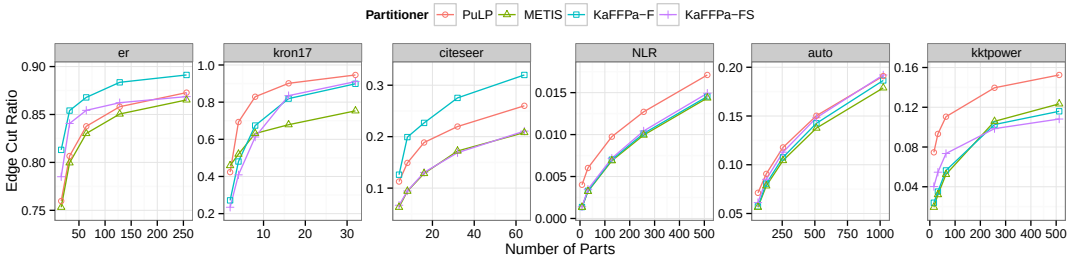

Figure 9.6. Edge cut versus number of parts for a few select representative instances from the 10th DIMACS Challenge for PuLP, METIS, KaFFPa-fast, and KaFFPa-fastsocial.

be discerned in the plotted results. We observe that PuLP is usually within a small factor of the best partitioner and is sometimes better than at least one of the other partitioners. Note that the test graphs here are mostly regular or semi-regular and lack inherent community structure. So despite the fact that PuLP was designed for a drastically different graph structure, the surprising quality of the results shown here demonstrate again just how versatile the general PuLP algorithmic approach can be for graph partitioning.

9.6 Related Work

There are a few other works known to us that use label propagation for the task of partitioning large-scale graphs. We compare our results with their published results, as the codes are not publicly available. Most recently, Meyerhenke et al. [163] utilized their sized-constrained label propagation technique for coarsening [162] in a distributed setting combined with an evolutionary partitioning algorithm to produce fast and high quality single objective and single constraint partitions. On a 32 core machine with 512 GB memory, they report execution times and edge cuts for bipartitioning several graphs from the DIMACS10 and LAW collections. We directly compare to the largest instances from each, sk-2005 and uk-2007. On sk-2005, [163] reports execution times and edge cuts of 471 seconds with 3.2 M cut edges and 1688 seconds with 2.9 M cut edges for their *Fast* and *Eco* variants, respectively. On uk-2007, they report 170 seconds with 1.0 M cut edges and 723 seconds with 981 K cut edges, respectively. Running single objective single constraint PuLP on these graphs with the same balance constraint (3%), we return

a cut of 6.3 M in 20 seconds on sk-2005 and a cut of 1.5 M in 30 seconds on uk-2007 on a 16 core and 64 GB node of *Compton*. Note, however, that PuLP generally demonstrates better relative cut quality with increasing part counts and in the multiple constraint scenario.

Vaquero et al. [165] implement vertex-balanced label propagation to partition dynamically changing graphs. Martella et al. improve upon this general approach with Spinner [166]. They report that Spinner produces a 0.45 edge cut ratio for an edge-balanced 64-way partitioning of the SNAP LiveJournal graph [69, 70]. By comparison, PuLP produces an edge cut ratio of about 0.35 under an equivalent constraint.

Ugander and Backstrom [164] implement label propagation for vertex-balanced partitioning as an optimization problem. They report performance on the LiveJournal graph for generating 100 parts, with a serial running time of 88 minutes and resultant edge cut ratio of 0.49. By comparison, our multi-constraint and multi-objective serial code creates 128 parts of the LiveJournal graph in about two minutes and produces an edge cut ratio of about 0.41.

Wang et al. [161] utilize label propagation in a manner similar to KaFFPa-FS, which is a multilevel approach with label propagation during graph coarsening. At the coarsest level, METIS is used to partition the graph. They also implement non-multilevel partitioning via a label propagation step followed by a greedy balancing phase. Their multilevel single-constraint and single-objective approach to partition LiveJournal has a serial running time of about 75 seconds, consumes about 1.5 GB memory, and has an edge cut about 25% greater than that produced by METIS alone. By comparison, our code consumes only 440 MB memory and produces cut quality comparable to METIS on the same graph. Their non-multilevel approach runs in about half the time, but at a considerable cost to cut quality.

9.7 Conclusions

In this chapter, we presented PuLP, a fast multi-objective multi-constraint partitioner for scalable partitioning of irregular small-world networks. The partitioning method in PuLP is based on the label propagation community detection algorithm. In a fraction of the execution time, while consuming almost an order of magnitude less memory, PuLP can produce partitions comparable or better in terms of total

edge cut to the k-way multilevel partitioning scheme in METIS. In addition, PuLP produces partitions that are better in terms of the maximal number of cut edges per-part. In the next chapter, we will actually demonstrate how one might actually use PuLP to accelerate distributed-memory graph analytics.

Chapter 10
Distributed Graph Layout

10.1 Introduction

This chapter introduces DGL: Distributed Graph Layout, which is our scheme for storing a large graph in distributed memory. We show how the in-memory layout of a graph can have considerable impact on the performance of parallel graph analytics executing on that graph. By using our PuLP partitioner to distribute vertices among tasks and a novel ordering strategy to reorder vertices in-tasks, we demonstrate improved performance relative to other techniques.

10.2 Distributed Graph Processing and Layout

Layouts of graphs and sparse matrices in distributed memory and shared memory have been well-studied for regular graphs that arise in the scientific computing domain. "Layout" in this instance refers to how vertices and edges are partitioned among distributed-memory MPI ranks and how the vertex identifiers are ordered in shared-memory. As has been observed, the impact of partitioning and ordering on irregular graph computations can be considerable [172, 181, 182]. However, using traditional layout strategies based on graph/hypergraph partitioners and orderings for data layout of highly irregular small-world graphs may not be appropriate for the following reasons:

1. Traditional partitioners and even some ordering methods, for example nested dissection, are heavyweight tools that are expensive both in terms of memory usage and time. They are appropriate when followed by even more expensive

linear solvers or when they can be computed once and used for multiple solves. In contrast, graph analytic workloads are constantly evolving and each analytic operation is typically much cheaper than a linear solver.

2. Previous ordering algorithms are designed for metrics appropriate for linear solvers such as minimizing a bandwidth [183] or minimizing the fill-in in a LU factorization [167, 184]. In contrast, ordering methods that improve the layouts in a shared memory context for small-world graphs are needed.

3. The performance of distributed-memory graph algorithms can be dependent on both local and global graph topology. Global topology affects the number of parallel phases and synchronization overhead, while local topological changes impact per-phase load balance. Optimizing for aggregate measures such as conductance or edge cut would ignore local topology changes and may not account for dynamic variations in per-phase execution.

Graph computations on highly irregular graphs require a layout that depends on parallel partitioners and ordering methods that are highly scalable for very large graphs. As such, we utilize the PuLP partitioner introduced in the prior chapter. In addition, we also introduce a breadth-first search-based ordering that is more scalable than other ordering schemes and suitable for small-world graphs in the shared-memory layout. In case of distributed graph processing, we consider various partitioning-ordering possibilities, a simultaneous global partitioning and ordering of all vertices, and a local ordering of vertices after the partitioning phase.

In short, we propose a "distributed-memory graph layout" based on vertex partitioning using label propagation and a BFS-based parallel ordering strategy. The proposed DGL (Distributed Graph Layout) is a fast, memory-efficient, and scalable graph layout strategy. We demonstrate the new DGL layout scheme is about 10-12× times faster to compute than METIS partitioning [167], and about 2.3× faster to compute than RCM-based orderings.

We demonstrate the impact of DGL and present detailed analysis on the end-to-end performance of distinct graph analytic workloads. The graph analysis routines include subgraph counting, breadth-first search (BFS), single-source shortest paths (SSSP), resource description framework (RDF) queries, and PageRank. The five algorithms were chosen to be representative of the diversity in modern graph analytics. We chose the Fascia algorithm for subgraph counting [10, 42, 43],

which is a randomized parallel algorithm to generate approximate counts of tree-structured subgraphs. Although recent related work [144, 157] primarily looks at strong scaling of BFS and related computations on massive synthetic Graph 500 networks, our work examines the subgraph counting algorithm, an analytic that is computationally very different from BFS. However, we also do an in-depth evaluation of BFS and SSSP performance. The fourth benchmark evaluates a distributed-memory implementation of the popular RDF store RDF-3X [185]. Our final included algorithm is a highly scalable implementation of PageRank [48], which is a popular and more computationally-intensive implementation than BFS for benchmarking performance of frameworks and systems.

We use the *end-to-end* graph analysis times for partitioning-ordering-workload in both single-threaded (MPI) and multi-threaded (MPI+OpenMP) distributed programming models. We also consider computation and communication times of the analytic separately, in order to better isolate the effects of partitioning and ordering on performance. We primarily consider *real-world* rather than synthetic graphs in our study. We use tuned implementations, all developed by us, in order to ensure consistency. We also analyze trade-offs between partitioning quality on computational load balance and communication overhead for several large real-world networks. The following are the key contributions of this workload analysis.

1. A comprehensive study of the performance of the five analytics with several partitioning-ordering combinations.

2. Our DGL ordering strategy is about 2× faster than RCM, and our PuLP partitioning strategy is about 10× faster than METIS.

3. We show that DGL layout improves subgraph counting performance by 1.28× in comparison to random partitioning. Partitioning with PuLP would enable end-to-end processing (partitioning & computation) of the counts of ten vertex subgraphs on the 2 billion edge Twitter graph to complete in under fifteen minutes on 16 nodes of *Blue Waters*.

4. DGL layout improves the communication time of BFS and SSSP by 1.48× and 1.43× in comparison to random partitioning.

5. An informed topology-aware graph layout benefits external memory compu-

tations as well, improving the performance of RDF3X-MPI, our distributed-memory implementation of the popular RDF store RDF-3X [185].

6. The total computation time of PageRank can be accelerated by about 5× with a locality-optimizing ordering such as DGL.

7. A cross-analytics comparison reveals new and interesting trade-offs of communication time, load balance, and memory utilization for various graphs.

We finally mention that DGL is not inherent to the MPI processing models considered in this work, and can therefore be utilized as a preprocessing step while running under other graph engines and parallel execution environments.

10.3 DGL: Distributed Graph Layout

In this section, we discuss the distributed graph layout using label propagation-based partitioning and BFS-based ordering methods. We define a distributed graph layout as the pair of *partitioning*×*ordering*. The partitioning part of the layout affects the number of parallel phases and synchronization overhead in a graph computation. It is important to balance the computation in different parallel phases as well as minimize the communication overhead. We explore trade-offs in work and memory balance and communication minimization between tasks with different partitioning strategies. Work performed and memory utilization per-task roughly correlates with the number of vertices and adjacent edges stored on each task. The communication requirements roughly correlates with the number of inter-task edges, or edge cut resulting from partitioning. The ordering part of the layout affects the per-phase computation time in graph computations. We ideally want to increase intra-node memory access locality to reduce cache misses and improve execution times. In order to be practical the *partitioning*×*ordering* pair must be computed in parallel, scalable fashion.

10.3.1 Partitioning

We utilize three partitioners in this work. We use a random partitioning to establish a baseline for benchmarking. We use the well-known METIS [167] partitioner as a representation of the state-of-the-art. We also utilize the PuLP partitioner, which is

specifically optimized to partition the small-world graphs we are considering in this work. We consider both single constraint and multi-constraint partitioning scenarios, where we either balance partitions for vertices or for both vertices and edges. We attempt to minimize total edge cut for both PuLP and METIS. Additionally, for PuLP, we also attempt to balance communication among parts by minimizing the maximal number of cut edges coming out of any single part.

Algorithm 10.1 Label Propagation Algorithm

```
procedure LABEL-PROP(G(V, E))
    for all v ∈ V do                                  ▷ Thread-level parallelism
        L(v) ← V_id(v)
    updates ← 1
    while updates ≠ 0 do
        updates ← 0
        for all v ∈ V do                              ▷ Thread-level parallelism
            Counts(1 ··· |V|) ← 0
            for all ⟨v, u⟩ ∈ E do
                Counts(L(u)) ← Counts(L(u)) + 1
            if Max(Counts(1 ··· |V|) ≠ L(v) then
                L(v) ← Max(Counts(1 ··· |V|)
                updates ← updates + 1
    return L
```

The PuLP partitioner is based off of the community detection label propagation algorithm [145]. Label propagation methods are attractive as they have low computational overhead, low memory utilization, are easy to parallelize, and demonstrate scaling to graphs with billions of vertices. An overview of the basic label propagation algorithm is given in Algorithm 10.1. The algorithm runs as follows. Initially, each vertex v in a graph G with vertex set V and edge set E has its label $L(v)$ initialized to a unique identifier. For simplicity, the initial labels are often just the vertices' numeric vertex identifiers V_{id}. We then consider each vertex in the graph, and update its label to the most common label appearing among all of its neighbors with ties broken randomly (e.g. vertex v has five neighbors, two of which have label a and three of which have label b; v will update its own label to b). This loop can be parallelized without any explicit synchronizations or locking with minimal effect on solution quality [47]. This process continues until no labels are updated, or, more commonly, after some number of iterations of the outermost

While loop (usually 10 or fewer iterations is sufficient).

Algorithm 10.2 PuLP Multi-Constraint Multi-Objective Algorithm Overview

```
Initialize p parts
Execute degree-weighted label propagation.
for k1 iterations do
    Balance parts for vertex constraint.
    Refine parts to minimize edge cut.
for k2 iterations do
    Balance parts to satisfy edge constraint
      and minimize max per-part cut.
    Refine parts to minimize edge cut.
```

PuLP's subroutines essentially use variants of Algorithm 9.1 that limit the number of possible labels to the number of desired parts and impose additional weighting criteria on the *Counts* array to create balanced partitions. This weighted form of label propagation is utilized in two separate stages during execution of PuLP. Algorithm 10.2 gives a very broad overview of the PuLP multi-constraint (vertices and edge per part) multi-objective (minimize total edge cut and maximum edge cut per part) algorithm that demonstrates these two stages. After initialization, we first utilize weighted label propagation in k_1 alternating stages to balance the initial parts for our vertex constraint and then refining to minimize the total edge cut. Next, we perform k_2 alternating stages of balancing for our edge constraint while minimizing the secondary objective of max per-part cut and then again refining to minimize the total edge cut. In prior work [47], we describe the algorithm in considerably greater detail and demonstrate the approach's effectiveness in terms of cut quality and runtime with respect to other traditional partitioners. However, it is critical to show that such label propagation-based partitionings are not only easy to compute, but that they also improve the end-to-end runtimes of graph analytic applications. With DGL, we are able to utilize such a partitioner in the layout strategy and demonstrate its applicability for the first time.

10.3.2 Ordering

For a distributed graph computation, a good graph partitioning will reduce inter-node communication cost. The goal of on-node vertex ordering is to increase locality of intra-node memory references, and thereby reduce intra-node computation time.

RCM is a commonly-used vertex ordering strategy in sparse matrix and graph applications. We propose a BFS-based ordering (see Algorithm 10.3) which can be considered an approximation to RCM. It avoids the costly sorting step used in RCM where it tries to order the nodes with the same parent in terms of the degree. Recently, a similar ordering was proposed for improving the matrix-vector multiply time and bandwidth reduction [186]. The primary focus of that approach was to arrive at parallel orderings to improve the linear solver time. Our focus is to improve the graph computations' end-to-end time.

We randomly choose a minimal-degree vertex as the *root* and perform a standard BFS routine, tracking visitation status with *visited* and the current level with *level*. We add vertices to level sets L when they are visited, as with RCM. We avoid explicit sorting by assuming that each $L_{0 \cdots Max_{level}}$, where Max_{level} is the maximum BFS level, is mostly sorted in the order of decreasing vertex degree, as there is a higher likelihood of encountering high-degree vertices sooner in any given level for most real world graphs. We assign new labels using an incrementing value of n by starting with the vertices in the highest level and working backwards to those in the lowest level. As we will show in the next section, this approach performs better than both random and RCM orderings in applications that have a high number of irregular memory accesses. As with RCM in [186], Algorithm 10.3 can be straightforwardly parallelized.

With the five partitioning methods (random, METIS {single constraint, single objective and multiple constraint, single objective} and PuLP {multiple constraint, single objective and multiple constraint, multiple objective} and three ordering methods (random, RCM and DGL) we evaluate all the combinations of *partitioning*×*ordering* pairs and demonstrate that the DGL layout with PuLP partitioner and DGL-based ordering performs the best in irregular graph computations.

10.4 Parallel Graph Computations

In this section, we will give an overview of the five distributed graph analytics used during our experimental analysis of the impact of partitioning and ordering on analytic performance. In an attempt to best understand the general effects of varying partitioning and ordering on the performance, the graph analytics were

Algorithm 10.3 DGL BFS-based vertex ordering algorithm.

$V_{id} \leftarrow$ DGL-order$(G(V, E))$
for all $v \in V$ **do**
  $V_{id}(v) \leftarrow v$
$level \leftarrow 0$
$root \leftarrow$ SelectRoot()
$Q \leftarrow root$
$Visited(1 \cdots |V|) \leftarrow$ **false**
while $Q \neq \varnothing$ **do**
  for all $v \in Q$ **do**
    Insert v into L_{level}
    for all $\langle v, u \rangle \in E$ **do**
      if $Visited(u) =$ **false then**
        $Visited(u) \leftarrow$ **true**
        Insert u into Q
  $level \leftarrow level + 1$
$Max_{level} \leftarrow level$
$n \leftarrow 0$
for $i = Max_{level} \cdots 1$ **do**
  for $j = 1 \cdots |L_i|$ **do**
    $V_{id}(L_i(j)) \leftarrow n$
    $n \leftarrow n + 1$

selected as to represent a wide range of execution characteristics. The test suite includes an implementation which is relatively computation-heavy, PageRank, algorithms which are relatively more communication-heavy, breadth-first search and single source shortest paths, an algorithm which is both very computation and communication intensive, color-coding subgraph counting, as well as an algorithm whose performance is dependent on the sizes of the n-hop neighborhoods of each partition, distributed query processing of Resource Description Framework stores.

10.4.1 Distributed PageRank

Our distributed PageRank algorithm is given by Algorithm 10.4. We use an MPI+OpenMP approach and an $\frac{|V|}{p}$ partitioning, with each of p MPI tasks calculating the counts for an equivalent portion of the $|V|$ vertices in the graph G. With one MPI task per node, we then use thread parallelism while updating the counts of owned vertices. With the exception of the single MPI communication call on

Algorithm 10.4 Distributed PageRank

```
1: procedure PageRank-dist(G(V, E), δ)
2:     for all v ∈ V do                                  ▷ Thread-level parallelism
3:         PageRanks(v) ← 1/|V|
4:     for i = 1 to iter do
5:         All-to-all exchange of updates in PageRanks
6:         for all v ∈ V do                              ▷ Thread-level parallelism
7:             PageRanks(v) ← 0
8:             for all ⟨v, u⟩ ∈ E do
9:                 PageRanks(v) ← PageRanks(v) + PageRanks(u)/|E(u)|
10:            PageRanks(v) ← PageRanks(v) × δ
11:            PageRanks(v) ← PageRanks(v) + (1−δ)/|V|
12:    return PageRanks
```

each iteration, all per-task work can be done in parallel. Updates are passed among neighbors using an MPI all-to-all exchange. In practice, this specific implementation has been observed to be very efficient and scalable, giving per-iteration costs of less than a few seconds for networks of over 100 billion edges while running on 256 compute nodes. The specific technical details of the implementation are omitted, but please see [48] for a more in-depth discussion.

10.4.2 Subgraph Counting

Subgraph counting is a computationally challenging task, with the naïve approach scaling as $O(n^k)$, where n is the number of vertices in a graph and k the number of vertices in the subgraph being counted. The best known exact algorithm [187] improves the exponent by a factor of $\frac{\alpha}{3}$, where α is the exponent for fast matrix multiplication. Because of these extremely high execution time bounds, recent work has focused on approximation algorithms. One such approach for counting *tree-structured* subgraphs utilizes the color-coding technique of Alon et al. [49].

In prior work, we developed a fast parallel implementations of color-coding subgraph counting in both shared-memory and distributed-memory environments [10, 42,43]. The distributed version of our approach uses several optimizations, including fully partitioning and compressing the memory-intensive dynamic programming table ($Count$) to decreases memory requirement across all tasks, further compress-

Algorithm 10.5 Subgraph counting Fully Partitioned Counting Approach.

Partition subgraph S using *single edge cuts*
for $it = 1$ to $Niter$ **do**
  Randomly color $G(V, E)$ with k colors
  for all S_i in reverse order of partitioning **do**
    Init $Count_{S_i}$ for V_r (vertex partition on task r)
    for all $v \in V_r$ **do** ▷ Thread-level parallelism
      for all $c \in C_i$ **do**
        Compute all $Count_{S_i,c,v}$
    $\langle N, I, B \rangle \leftarrow$ Compress($Table_{i,r}$)
    All-to-all exchange of $\langle N, I, B \rangle$
    Update $Count_{S_i}$ based on information received
  $Count_r += \sum_{v}^{V_r} \sum_{c}^{C_T} Count_{T,c,v}$
$Count \leftarrow$ Reduce($Count_r$)
Scale $Count$ based on Niter and colorful embed prob.

ing the table during communication to reduce the total transfer volume, and using all-to-all exchanges in lieu of broadcasts to reduce communication times. These optimizations demonstrate good scaling and enable us to count subgraphs of 10 and 11 vertices on billion-edge networks in minutes on a modest number of 16 nodes. An overview of the main subgraph counting algorithm as implemented here is given in Algorithm 10.5. For space consideration, we omit a detailed description of our implementation. Instead, please refer to [43] for an in-depth discussion of the stages and execution of the algorithm.

10.4.3 SSSP and BFS

We also assess the performance impact of layout on tuned implementations for parallel breadth-first search (BFS) and single-source shortest paths (SSSP) computation in this chapter. Our parallel BFS approach can take advantage of both 1D and 2D graph distributions [172,188,189]. We use a 1D distribution in this work, as it is easier to correlate communication time with edge cut after partitioning with a 1D distribution. Recent BFS and SSSP implementations use a 1D partitioning and direction-optimizing search [112] for work-efficient and highly scalable execution on Graph 500 test instances. For an overview of the current state-of-the-art in performance optimizations for these routines, we refer the reader to [144, 157].

We use an optimized parallel implementation [190] of the Δ-stepping algorithm [191] for parallel SSSP in this chapter. Each BFS iteration and Δ-stepping phase is comprised of three main steps: local discovery, all-to-all exchange, and local update. To aid adjacency queries, we use a distributed compressed sparse row representation for a graph. The distance array is also partitioned and distributed along with the distributed vertices (for Δ-stepping). In the local discovery step, both algorithms expand their frontiers by listing all corresponding adjacencies and their proposed distance based on vertices in a queue of recently-visited vertices (for BFS) or in a current bucket (for Δ-stepping). Note that BFS visits each reachable vertex only once while Δ-stepping may visit each reachable vertex multiple times before it is settled.

Once all vertices in the queue are processed or the current bucket is empty (with no more vertex reinsertions), all p tasks exchange vertices in these generated lists to make them local to the owner tasks. This step is the same for both BFS and Δ-stepping, and uses an all-to-all collective communication routine. At the end of each BFS iteration and Δ-stepping phase, each task locally updates the distance of its own vertices using the exchanged information. The update in BFS is only on unvisited vertices, while Δ-stepping updates all vertices whose distances can be decreased. Thus, the Δ-stepping algorithm performs more computation and has a higher communication complexity.

Since our goal is to analyze and evaluate the effect of graph partitioning and vertex reordering, we have not yet implemented all the optimizations in [144, 157]. However, our approach has three new optimizations: (i) A semi-sort of vertex adjacencies based on weights is used prior to execution of the algorithm. (ii) Memory-optimized queues are used to represent the bucket data structure. This decreases the algorithm memory requirement, while slightly increasing the running time. (iii) An array of all local unique adjacencies is created and locally used to track tentative distance of adjacencies. This array improves efficiency by filtering out unnecessary requests to be added in the new frontiers.

10.4.4 Distributed RDF Stores and SPARQL Query Processing

Resource Description Framework (RDF) [192] is a popular data format for storing web data sets. Informally, the RDF format specifies typed relationships between

entities, and the basic record in an RDF data set is a *triple*. There are a growing number of publicly-available RDF data sets that contain billions of triples. Thus, database methodologies for storing these RDF data sets, also called triple stores [193,194], are becoming popular. We have developed a distributed MPI-based implementation of an open-source triple store called RDF-3X [185]. Our distributed RDF store is called RDF3X-MPI [195].

An alternate approach to viewing an RDF data set is as a directed graph with edge types. RDF data sets can be queried using a language called SPARQL. We extend the distributed RDF store methodology of RDF-3X to the SPARQL querying phase as well. Thus our RDF3X-MPI tool has two phases, a load phase and a query phase. In the load phase, the given triple data set is partitioned into several independent files, one per task, and each task then constructs an index for helping answering SPARQL queries. It is possible to parallelize some query evaluation in a purely data-parallel manner (i.e., with no communication between tasks), provided there is sufficient replication of triples among partitions. Formally, if the triple partitions satisfy an *n-hop guarantee*, then SPARQL queries in which all pairs of join variables are at distance of less than n hops from each other can be solved without any inter-task communication [196]. So the role of graph partitioning in this application is to reorder vertices such that the number of triples that are replicated between tasks after applying an n-hop guarantee are minimized. If the number of triples that are replicated is reduced, then the database indexes are smaller, making them potentially faster to query. For this application, we study the impact of partitioning on the number of replicated triples. A smaller value of replication is desired, and further, smaller index sizes should translate to faster query times.

10.5 Experimental Setup

We evaluate performance of our new partitioning and ordering strategy DGL and the graph analytics workload on a collection of nine large-scale low diameter graphs, listed in Table 10.1. LiveJournal, Orkut, and Twitter (follower network) are crawls of online social networks obtained from the SNAP Database and the Max Planck Institute for Software Systems [69,73]. uk-2005 and sk-2005 are crawls of the United Kingdom (.uk) and Slovakian (.sk) domains performed in 2005 using UbiCrawler

and downloaded from the University of Florida Sparse Matrix Collection [74,75,179]. WebBase is similarly a crawl obtained in 2001 by the Stanford WebBase crawler. We created the BSBM and LUBM graphs from RDF data sets generated using the Berlin SPARQL benchmark [197] and Lehigh University Benchmark [198] generators. DBpedia was created from RDF triples extracted from Wikipedia [199].

The Orkut graph is undirected and the remaining graphs are directed. For the web and social graphs, we preprocessed the graphs before executing PageRank, BFS, SSSP, and subgraph counting. Specifically, we removed all degree-0 vertices, multi-edges, and extracted the largest (weakly) connected component. Further, edge directivity was ignored when partitioning the graphs using PuLP and METIS and reordering with RCM and DGL. Table 10.1 lists the sizes of these nine graphs after preprocessing.

Network	Category	n	m	d_{avg}	d_{max}	$\widetilde{D}$	Source
LiveJournal	OSN	4.8 M	42 M	18	39 K	21	[70]
Orkut	OSN	3.1 M	117 M	76	33 K	9	[72]
Twitter	OSN	44 M	2.0 B	37	750 K	36	[73]
uk-2005	WWW	39 M	781 M	40	1.8 M	21	[179]
WebBase	WWW	113 M	844 M	15	816 K	376	[179]
sk-2005	WWW	44 M	1.6 B	73	15 M	308	[179]
BSBM	RDF	16 M	67 M	8.6	3.6 M	7	[197]
LUBM	RDF	33 M	133 M	8.1	11 M	6	[198]
DBpedia	RDF	62 M	190 M	6.1	7.3 M	7	[199]

Table 10.1. Test graph characteristics *after* preprocessing. Graphs belong to three categories, OSN: Online social networks, WWW: Web crawl, RDF: graphs constructed from RDF data. # Vertices (n), # Edges (m), average (d_{avg}) and max (d_{max}) vertex degrees, and approximate diameter ($\widetilde{D}$) are listed. $B = \times 10^9$, $M = \times 10^6$, $K = \times 10^3$.

The scalability studies for subgraph counting, BFS, SSSP, and RDF query processing were done primarily on *Blue Waters*, a large petascale supercomputer at the National Center for Supercomputing Applications (NCSA). Each XE compute node of *Blue Waters* is a dual-socket system with 64 GB main memory and AMD 6276 Interlagos processors at 2.3 GHz. The system uses a Cray Gemini 3D torus interconnect. We built our programs with the GNU C++ compiler (version 4.8.2), using OpenMP for multithreading and the `-O3` optimization parameter during compilation. For the pre-processing phases of DGL (partitioning and reordering)

and some scalability runs, we utilized *Compton*, a testbed cluster. *Compton* has a dual socket setup with Intel Xeon E5-2670 (Sandy Bridge) CPUs at 2.60 GHz and 64 GB main memory. Due to the large memory requirements of partitioning with METIS, we also had to use the large memory nodes on *Carver* at NERSC for partitioning the larger networks (Twitter, uk-2005, Webbase, and sk-2005). Carver's large memory nodes have 1024 GB main memory and four Intel Xeon X7550 ("Nehalem-EX") CPUs at 2.00 GHz.

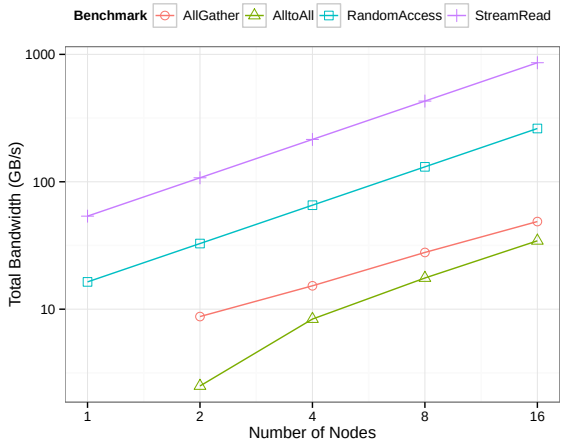

Figure 10.1. Bandwidth of *Blue Waters* for various memory and MPI benchmarks.

To give a relative sense of the intra-node data access and inter-node collective communication performance on *Blue Waters*, we present some memory and collective communication performance results in Figure 10.1 using micro benchmarks These benchmarks include all-gather and all-to-allv MPI bandwidths and intra-node memory bandwidth sustained for regular stride-1 reads and random memory accesses, as a function of processing nodes.

10.6 Results and Discussion

10.6.1 DGL Performance Evaluation

We first evaluate our DGL label propagation-based partitioning methodology, PuLP, against METIS partitioning by examining total running time for generating 16 and 64 partitions. We consider two versions of both PuLP and METIS. For PuLP, we have an implementation that has *both* maximal vertex and edge balance

constraints and minimizes *both* total edge cut and maximal per-part edge cut. We consider this our baseline implementation, and label it in figures as PuLP-MM (PuLP multi-objective multi-constraint). We also have a dual constraint version that only attempts to minimize the total edge cut, which we call PuLP-M. Similarly for METIS, the dual constraint single objective version is termed METIS-M, while the single constraint (vertex balance) and single-objective version is termed simply as METIS. METIS-M and PuLP-M are solving the same problem. For our constraints, we fix the maximal vertex imbalance ratio at 1.10 and the edge imbalance ratio at 1.50. The results will show that the multi-constraint, multi-objective mode of PuLP-MM can be important for irregular graph computations.

Table 10.2 shows the partitioning time of PuLP-MM along with METIS-M running on *Compton*. Due to METIS's large memory requirements (close to 500GB for Twitter), only LiveJournal, Orkut, and the RDF graphs could be partitioned on *Compton*. The larger web graphs and Twitter were all partitioned on a large memory node of Carver. We also report the relative speedup of PuLP to METIS. We omit time comparison to ParMETIS, as the only graphs it was able to successfully partition on any system were LiveJournal and Orkut. Further, ParMETIS's speedups relative to METIS for those two instances were minimal (less than 2× with 16-way parallelism). From Table 10.2 we observe considerable speedup for PuLP, with a geometric mean speedup of 12.4× for 16 parts and 10.1× for 64 parts.

The partitioning quality in terms of both vertex and edge balance constraints and edge cut and maximal per-part edge cut objectives for the different partitioners is shown in Table 10.3. In terms of the total edge cut (EC), the single-constraint, single-objective METIS does the best, but it performs poorly in the maximum per-part edge cut (EC_{max}) and edge balance (E_{max}). PuLP-MM also performs better than all the methods in the EC_{max} metric without sacrificing a lot in EC and still respecting the vertex balance and edge balance constraints. Also note the much larger E_{max} of single constraint METIS. As we will demonstrate, this can have a considerably impact of execution time for the applications in our benchmarks. Note that while METIS does better in $\#CC$, it does not affect the graph analytic applications. Traditional partitioners tend to look for fully connected components. In small-world graphs and the applications that use them this does not necessarily translate into better performance.

Network	16-way partitioning			64-way partitioning		
	METIS-M time (s)	PuLP-MM time (s)	Speedup	METIS-M time (s)	PuLP-MM time (s)	Speedup
LiveJournal	75	7.4	10×	74	7.3	10×
Orkut	156	10	16×	197	13	15×
Twitter	12348	530	23×	12484	565	22×
uk-2005	255	15	17×	353	80	4.4×
WebBase	539	39	14×	551	42	13×
sk-2005	465	39	12×	514	65	7.9×
BSBM	348	28	12×	395	32	12×
LUBM	707	88	8.0×	966	123	7.9×
DBpedia	898	133	6.8×	1001	133	7.5×

Table 10.2. PuLP-MM and METIS-M partitioning time with 16-way and 64-way partitioning. PuLP-MM uses multi-constraint multi-objective partitioning. METIS-M uses multi-constraint single-objective partitioning.

Partitioning	V_{max}	E_{max}	$EC(imp)$	$EC_{max}(imp)$	$\#CC(imp)$
Random	1.15	1.70	1.00	1.00	1.00
METIS	1.10	3.88	**7.71**	2.39	**202**
METIS-M	1.10	1.50	4.40	2.16	62.1
PuLP-M	1.10	1.50	5.50	2.10	72.0
PuLP-MM	1.10	1.50	5.00	**3.18**	22.9

Table 10.3. Average partitioning characteristics across all graphs. Geometric mean of vertex balance V_{max}, edge balance E_{max}, improvement over random partitioning for edge cut ratio EC and max per-part edge cut EC_{max}, and the mean improvement (decrease) in the average total number of connected components for all parts (#CCs) are shown. The best values for each of the last three columns are in bold font.

We additionally compare our DGL vertex ordering strategy to RCM. Table 10.4 gives the average running times of both DGL and RCM in serial across all three partitioning strategies for reordering the vertices within each partition. DGL reordering results in a 2.3× average speedup compared to RCM for reordering both 16 and 64 parts. This reduction is due to the avoidance of explicit sorting required by RCM. There does not seem to be a large dependence of running times on the number of partitions, although with a greatly increased partition count for a fixed graph, it would be expected that running time decreases due to a lower diameter

Network	16-way partitioning			64-way partitioning		
	RCM time (s)	DGL time (s)	Speedup	RCM time (s)	DGL time (s)	Speedup
LiveJournal	2.3	1.0	2.3×	2.3	1.0	2.3×
Orkut	3.9	1.9	2.1×	3.9	1.9	2.1×
Twitter	50	24	2.1×	61	29	2.1×
uk-2005	16	8.4	1.9×	17	7.6	2.2×
Webbase	33	13	2.5×	35	17	2.1×
sk-2005	24	11	2.2×	23	11	2.1×
BSBM	5.1	2.3	2.2×	4.7	2.3	2.0×
LUBM	5.7	1.7	3.4×	5.7	1.7	3.4×
DBpedia	16	6.1	2.6×	17	6.9	2.5×

Table 10.4. DGL serial reordering time with 16-way and 64-way partitioning.

BFS search and overall increased cache utilization. Both these methods can be parallelized as DGL can use a parallel BFS and RCM can be implemented using the parallel version [186]. However, their timings are insignificant in the end-to-end performance of complex analytics such as our subgraph counting benchmark.

We include one more table to demonstrate how our DGL ordering strategy might improve cache performance of executing codes. To improve the performance of linear solvers, a common ordering metric to optimize for is graph bandwidth, which is the maximum integer distance between vertex identifiers for vertices that share a single neighbor. RCM is an effective means of bandwidth reduction for regular matrices. However, for small-world graphs, the bandwidth is usually going to be large, on the order of d_{max}, where d_{max} is the maximal degree of any vertex in the graph. Comparing bandwidth measures between different orderings therefore won't show any global improvements in compaction for rows of much lesser degree vertices.

As such, we look at other metrics to give an indication of the possible cache efficiency in practice. Across the entire adjacency array, we measure how often edges listed in order also have identifiers within a single integer value of each other. This indicates that these edges would be neighboring nonzeros in the same row of an adjacency matrix. Co-located edges improve cache utilization of per-vertex information accesses, such as checking visitation status for BFS or PageRank value lookups. To quantify how many co-located vertex identifiers for the edges are in the

adjacency list, we report two values. First, we report a ratio of how co-located all edges are, where a value of zero indicates that no edges are co-located and a value of one indicates that all edges are co-located. Second, we report a running "cost" as the sum of the distances, or gaps, between vertex identifiers in the adjacency list. We scale the distances by their log, as a distance of one or close to it indicates that vertices are closely co-located and would have minimal cache cost for their subsequent accesses, and the cost difference between large and very large distances is minimal, since it's likely a new cache line would need to be loaded in both instances. Finding an ordering that minimizes this sum is referred to in the literature as the *Minimum Logarithmic Gap Arrangement* problem, which is NP-hard [200]. In an attempt to give a graph-independent ratio, we further scale the log sum by a worst-case possible value of $m \log n$. The true dependence of cache utilization on distance would be architecture-specific, but the approximation of this cost gives enough insight for comparative purposes when examining ordering quality.

Network	16-way partitioning					64-way partitioning				
	Co-loc. Ratio		Gap Sum Ratio			Co-loc. Ratio		Gap Sum Ratio		
	DGL	RCM	DGL	RCM	Rand	DGL	RCM	DGL	RCM	Rand
LiveJournal	0.115	0.010	0.036	0.034	0.043	0.104	0.014	0.009	0.006	0.012
Orkut	0.028	0.001	0.046	0.057	0.054	0.021	0.001	0.010	0.020	0.011
Twitter	0.032	0.005	0.037	0.035	0.038	0.026	0.006	0.013	0.010	0.016
uk-2005	0.659	0.176	0.015	0.022	0.046	0.582	0.184	0.005	0.006	0.011
Webbase	0.562	0.162	0.020	0.039	0.050	0.519	0.172	0.005	0.006	0.011
sk-2005	0.613	0.149	0.018	0.026	0.050	0.689	0.167	0.002	0.004	0.013
BSBM	0.146	0.146	0.040	0.040	0.062	0.146	0.153	0.007	0.006	0.009
LUBM	0.105	0.105	0.026	0.026	0.045	0.094	0.105	0.006	0.006	0.009
DBpedia	0.442	0.257	0.028	0.036	0.046	0.398	0.267	0.006	0.008	0.010
Overall	**0.298**	0.112	**0.030**	0.034	0.048	**0.274**	0.119	**0.007**	0.008	0.011

Table 10.5. Ordering performance for DGL, RCM, and Random in terms co-location ratio (Co-loc. Ratio) and log sum of gap distances (Gap Sum Ratio) for 16-way and 64-way partitioning, averaged across the five different partitioning strategies.

Table 10.5 gives both the co-location ratio (Co-loc. Ratio) as well as the log sum of gap distances ratios (Gap Sum Ratio) for all ordering combinations across all graphs for 16 and 64 parts. We report the geometric mean values across all five partitioning strategies (Random, METIS, METIS-M, PuLP-M, PuLP-MM). For co-location ratio, higher indicates better locality, while for the log gap sum ratio, lower indicates better locality. We omit reporting the co-location ratio for Random

ordering in Table 10.5, as all values are close to zero, a few orders of magnitudes less than RCM and DGL. We observe nearly that DGL ordering results in the best co-location ratio and lowest log gap sum ratio ratio across almost all instances. The computational timings results we'll report next in our benchmarks will demonstrate that these measurements translate into real performance benefits across a wide range of graph analytics.

10.6.2 PageRank Performance

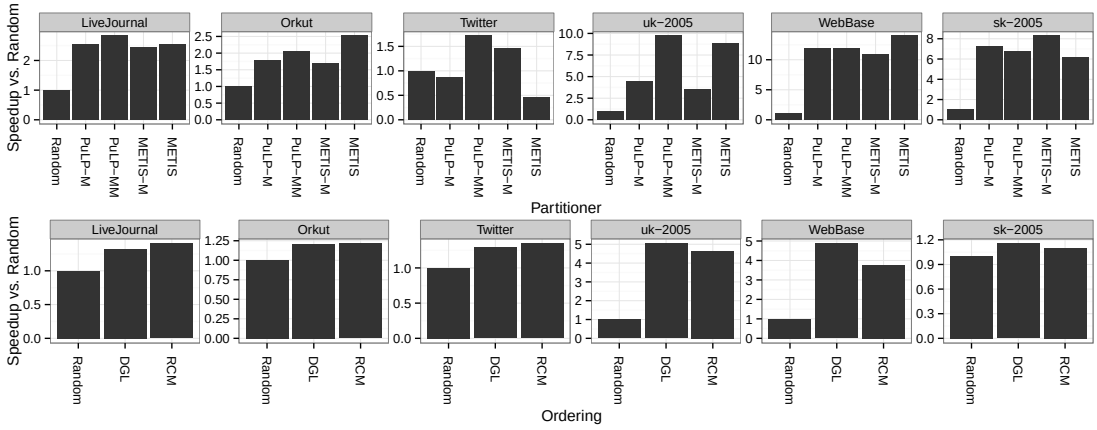

Figure 10.2. Communication speedup of the PageRank implementation on 16 nodes with various partitioning options (top) and computation speedup of PageRank with various ordering strategies (bottom).

Network	Partitioning				Ordering	
	METIS	METIS-M	PuLP-M	PuLP-MM	RCM	DGL
LiveJournal	2.560	2.453	2.561	2.832	1.404	1.325
Orkut	2.519	1.689	1.784	2.068	1.214	1.205
Twitter	1.454	1.459	1.871	1.716	1.346	1.292
uk-2005	8.913	3.518	4.427	9.725	4.641	5.039
WebBase	13.99	10.87	11.92	11.93	3.776	4.870
sk-2005	6.170	8.293	7.287	6.797	1.100	1.155
Overall	3.621	3.525	3.395	**4.465**	1.881	**1.970**

Table 10.6. Speedups of various partitioning and ordering strategies versus random partitioning and random ordering for the PageRank counting benchmark.

For our first set of experimental benchmarks results, we examine the effect of partitioning and ordering on a distributed PageRank implementation. We will first show the effect that different partitionings have on communication times, and then we will show the effect that orderings have on computation times. For these experiments we use the three social network graphs (LiveJournal, Orkut, and Twitter) as well as the three web crawls (uk-2005, WebBase, sk-2005). Figure 10.2 (top) gives the speedups relative to random partitioning for METIS single and multiple constraint partitionings and PuLP multiple constraint with single and multiple objective partitionings. Figure 10.2 (bottom) gives the speedups relative to random ordering for DGL and RCM. Table 10.6 gives the explicit speedup values and overall geometric means across the six test graphs. These value are for 20 iterations of PageRank executing on 16 nodes of *Blue Waters*.

We observe that all partitionings offer considerable speedups relative to random. In general, the web crawls show even greater speedups than the social networks. This is due to the web crawls being greater in diameter and more separable than social networks, resulting in a decrease in the number of cut edges and subsequently greater performance improvements relative to random. Averaged across all six test graphs, PuLP multiple constraint and multiple objective partitioning offers the greatest speedup. The performance benefit is due to the implementation's use of an iterative bulk synchronous model and moderately low required communication, so the improved communication balance resulting from PuLP-MM's decrease in max per-part cut becomes apparent in the timings.

Additionally, we note that both RCM and DGL offer considerable speedups for total computation times relative to random ordering. On the uk-2005 and WebBase graphs, the speedups for DGL are about 5×. Again we observe that the social networks generally show less performance benefit relative to random, and this is again due to their lower diameter and small-world characteristics, which makes effective ordering more difficult. However, we still observe a consistent 20%-40% speedups with the improved orderings. Overall, DGL gives a greater performance speedup over RCM by about 10%, a result we expected based on our measurement of potential locality and cache performance as demonstrated in Table 10.5.

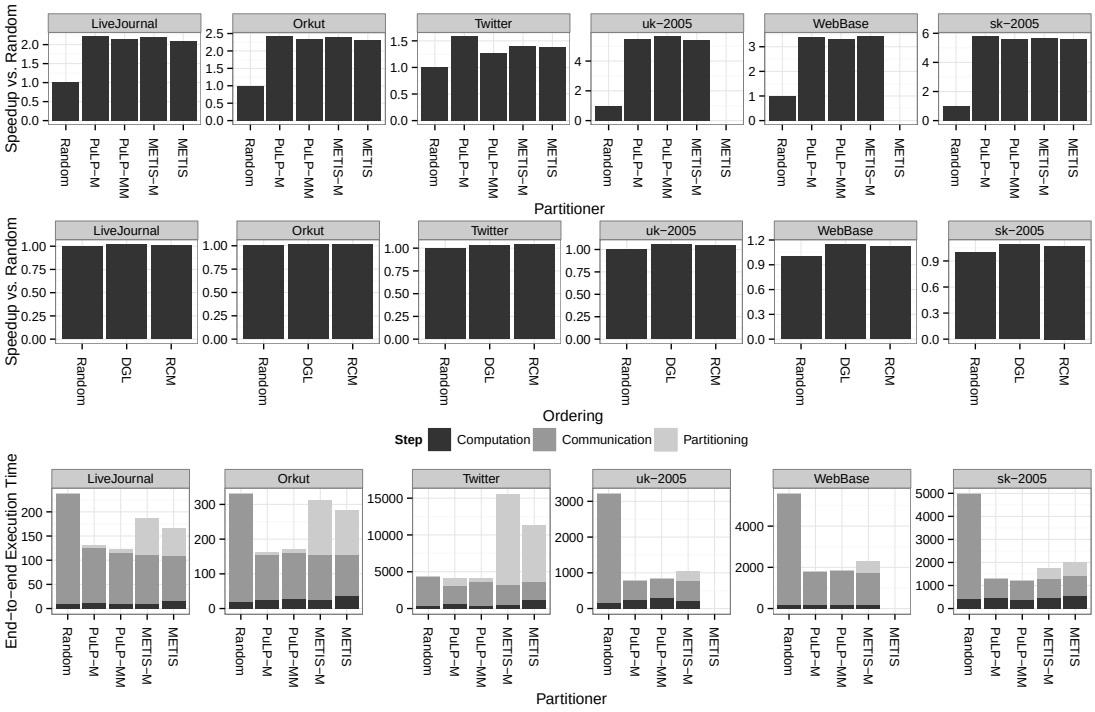

Figure 10.3. Speedups achieved with subgraph counting for total communication time of the various partitioning strategies relative to random partitioning, all with random ordering. Additionally, the speedups for the RCM and DGL orderings relative to random ordering with PuLP multi objective partitioning. The bottom plot gives total end-to-end execution time in terms of the initial partitioning, total computation time, and total communication time.

Network	Partitioning				Ordering	
	METIS	METIS-M	PuLP-M	PuLP-MM	RCM	DGL
LiveJournal	2.099	2.202	2.211	2.150	1.009	1.020
Orkut	2.307	2.400	2.411	2.350	1.014	1.015
Twitter	1.378	1.399	1.580	1.271	1.041	1.029
uk-2005	-	5.433	5.476	5.642	1.049	1.057
WebBase	-	3.412	3.375	3.311	1.125	1.148
sk-2005	5.568	5.675	5.772	5.621	1.072	1.091
Overall	-	3.033	**3.106**	2.961	1.051	**1.059**

Table 10.7. Speedups of various partitioning and ordering strategies versus random partitioning and random ordering for the subgraph counting benchmark.

10.6.3 Subgraph Counting Performance

We next compare the impact of various partitioning and ordering strategies with regards to the running times of our subgraph counting implementation. We run on

16 node of *Blue Waters*. We compare communication times resulting from each of the 5 partitioners with a fixed random ordering. We also compare the computation times resulting from the 3 ordering strategies with fixed PuLP-MM partitioning. The speedups for each strategy on the 6 test graphs are given in Figure 10.3 and Table 10.7. We also look at total end-to-end execution time for the five partitioning strategies with random ordering in terms of total time spent in the communication, computation, and partitioning steps. Note that the results with single constraint METIS for the uk-2005 and WebBase graphs are absent. This is due to execution times taking longer than 24 hours for these instances.

Several trends can be observed in Figure 10.3. The top subfigure gives the speedup of the communication phase of subgraph counting for each of the partitioning strategies relative to random partitioning. We again note considerable speedup for all partitioners. We note that the PuLP methods give the best improvement for five out of the six tested graphs. Overall PuLP-M gives the highest speedup overall. This implementation doesn't benefit as highly from the more communication-balanced PuLP-MM partitioning due to the overall higher communication requirements (the Twitter graph requires compression and transfer of several terabytes of data in total for the *Count* table exchanges between tasks) and lower overall synchronization cost relative to PageRank, so total edge cut is observed to have a greater effect in practice. This emphasizes the fact that a one-size-fits all solution is not optimal in practice, and implementation knowledge is required to extract the best performance for any given running application when utilizing a layout strategy.

The middle subfigure of Figure 10.3 plots the speedup relative to random ordering for the DGL and RCM reordering strategies with PuLP-MM partitioning. We again note that DGL reordering demonstrates the highest speedup for five out of the six test instances. Note that DGL ordering can both be computed faster than RCM and can also result in better application performance. The reordering makes more noticeable impact on the larger graphs, where the importance of cache efficiency is higher, as is expected. Overall, we note about a 6% improvement for DGL and 5% improvement for RCM ordering relative to random. These improvements are much lower than PageRank's improvement due to considerably more information stored per-vertex in the stored counts table, so greater cache locality has less of an effect in preventing re-accesses to main memory; however,

we note even a modest 5%-6% consistent improvement can be noteworthy in this instance. On processors with larger cache, this relative improvement would be expected to increase.

Finally, the bottom subfigure of Figure 10.3 shows the total end-to-end execution times for initial partitioning plus running of the subgraph counting application. We further split subgraph counting into the sum of time spent in each of its computation and communication phases. We observe that our partitioning and ordering strategies result in the fastest end-to-end running times for all test instances. The time spent for partitioning is considerable relative to execution time for METIS, as is the extra communication costs that result with random partitioning. The additional partitioning time cost for METIS might be amortized in practice by re-using the same partitions for subsequent analysis, but we note that **PuLP partitioning shows an immediate decrease in total end-to-end time after a single analytic run**.

10.6.4 Execution Timelines

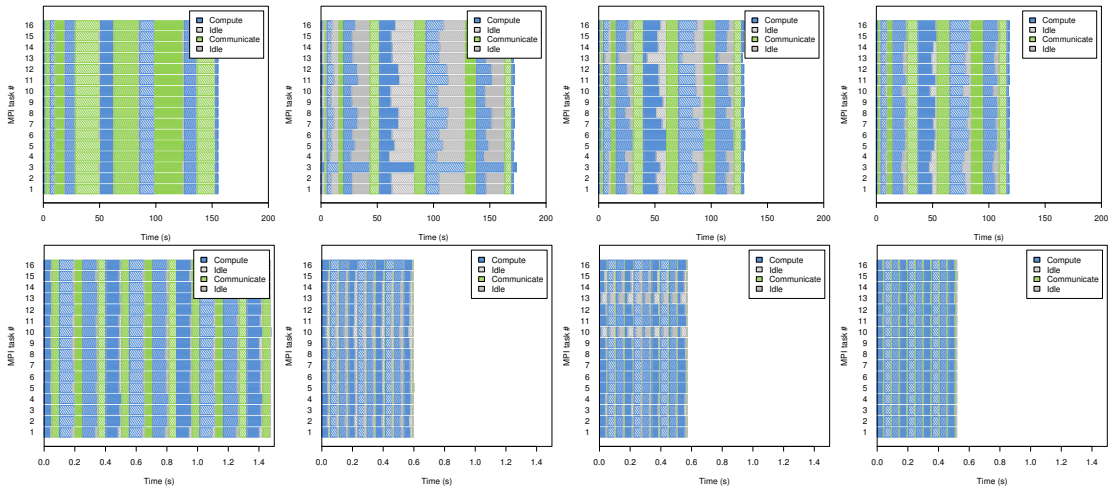

Figure 10.4. Subgraph counting (top, single color-coding iteration with a 10-vertex template) and PageRank (bottom, 10 iterations) execution timelines on 16 tasks and 32 threads with (left to right) random, single and multi-constraint METIS, and PuLP-MM partitioning strategies. Random ordering was used in all cases.

To offer visual explanation of the performance of balanced constraint partitioning on total execution time, we give execution timelines in Figure 10.4 of a single run of counting a 10 vertex template on the LiveJournal graph and 10 iterations of

PageRank on the Webbase graph. We used the *Compton* system for these tests and random, single-constraint METIS, multi-constraint METIS, and PuLP-MM partitioning (from left to right, respectively) with random ordering. On looking at subgraph counting (top), we note first the two extreme cases. Random shows the lowest total computation times at a high cost of communication, while single objective METIS results in low communication times but high total times during the execution stages. This is due to unbalanced work among each task, which is directly proportional to the edge balance among each part. We observe that balanced multi-objective PuLP partitioning gives the best tradeoff in terms of work balance and communication requirements. For PageRank, we note a large performance gap between Random partitioning and the other strategies. This is due to the implementation's computational and communication requirements for each task being dependent on the one hop neighborhood and per-part cut. These values are much higher with Random partitioning. We observe that PuLP gives the best performance, due to the fact that the multiple objectives are explicitly optimizing for these metrics while keeping work balance very consistent. Overall, we notice about a 5-10% total execution time improvement for PuLP versus the METIS variants by using multi-objective partitioning. As noted, considering total end-to-end execution time with partitioning costs, this speedup would be even more dramatic.

10.6.5 SSSP and BFS Performance

In this section, we analyze the performance of our SSSP and BFS implementation when using the different partitioning and ordering layouts. These benchmarks were run on 64 nodes of *Blue Waters*. While the running time of distributed subgraph counting is dominated by large-scale data transfers during the communication phases, SSSP's performance is more dependent on intra-task computation, similar to PageRank, but has considerably less communication. BFS has the overall lowest communication and computation requirements out of all of the benchmarks thus far.

Figure 10.5 and Table 10.8 show the speedups for communication and computation for SSSP performance with 64 MPI tasks. The top subfigure of Figure 10.5 shows the communication speedups relative to random partitioning for the other

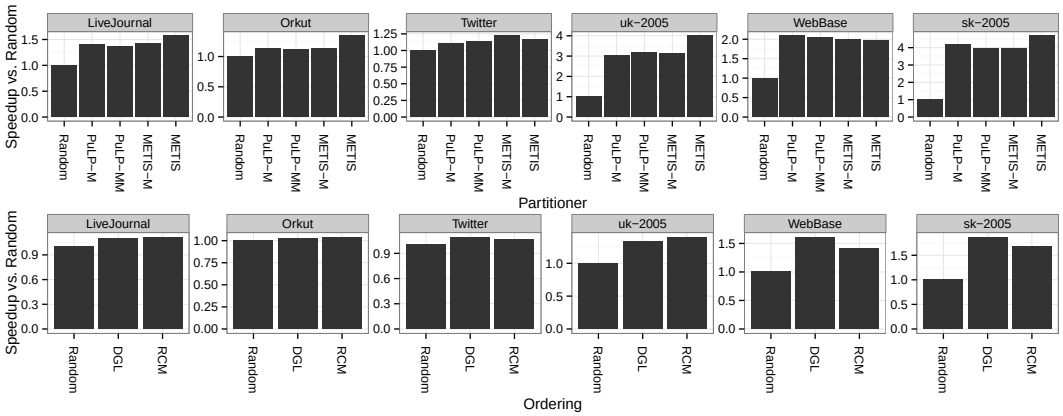

Figure 10.5. Communication time of SSSP implementation on 64 nodes with various partitioning options (top) and computation time of SSSP with various ordering strategies (bottom).

Network	Partitioning				Ordering	
	METIS	METIS-M	PuLP-M	PuLP-MM	RCM	DGL
LiveJournal	1.575	1.422	1.400	1.372	1.117	1.097
Orkut	1.346	1.123	1.131	1.111	1.041	1.026
Twitter	1.172	1.224	1.109	1.141	1.063	1.094
uk-2005	4.008	3.122	3.044	3.187	1.399	1.328
WebBase	1.971	1.998	2.092	2.035	1.407	1.612
sk-2005	4.693	3.934	4.159	3.963	1.689	1.870
Overall	**2.125**	1.907	1.897	1.884	1.266	**1.304**

Table 10.8. Speedups of various partitioning and ordering strategies versus random partitioning and random ordering for the SSSP counting benchmark.

partitioning strategies. Due to the relatively lower communication requirements for this SSSP implementation, we correspondingly observe lower speedups relative to what was observed in Figure 10.3 with subgraph counting. We note that METIS gives the highest communication speedup, due to the lower overall communication load and total synchronization costs which emphasize a lower total workload than explicit balance. We observe speedups for computation times on all graphs, and especially the web crawls, with both ordering strategies. DGL ordering gives around 30% speedup overall.

The two subfigures of Figure 10.6 and Table 10.9 give the speedup in communication time with different partitioners and speedups in computation time with

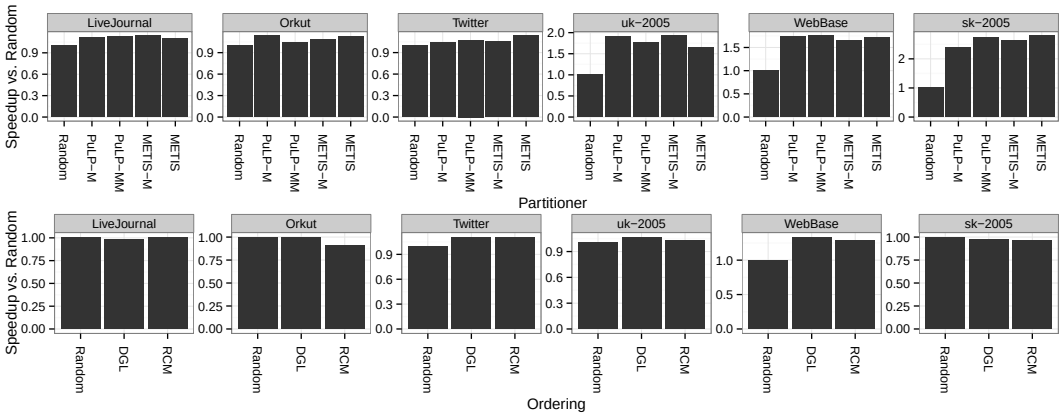

Figure 10.6. Communication time of BFS implementation on 16 nodes with various partitioning options (top) and computation time of BFS with various ordering strategies (bottom).

Network	Partitioning				Ordering	
	METIS	METIS-M	PuLP-M	PuLP-MM	RCM	DGL
LiveJournal	1.100	1.141	1.110	1.124	1.007	0.987
Orkut	1.125	1.083	1.138	1.038	0.913	0.991
Twitter	1.127	1.060	1.047	1.076	1.099	1.109
uk-2005	1.655	1.929	1.920	1.763	1.023	1.068
WebBase	1.709	1.657	1.732	1.756	1.287	1.335
sk-2005	2.798	2.624	2.393	2.737	0.960	0.977
Overall	**1.494**	1.491	1.480	1.483	1.042	**1.071**

Table 10.9. Speedups of various partitioning and ordering strategies versus random partitioning and random ordering for the BFS counting benchmark.

different orderings for the BFS implementation. We notice similar trends to SSSP in these plots. Overall, the lower total computation and communication workload of BFS contributes to lower speedups when using better ordering and partitioning strategies compared to random.

10.6.6 SPARQL Query Processing

In this final section, we study the impact of partitioning and ordering on the performance of RDF stores and SPARQL querying, a benchmark algorithm that is very different than the previous ones. In Table 10.10, we report replication

ratios observed when an undirected 2-hop guarantee is enforced. Our RDF3X-MPI implementation uses a 2-hop guarantee to partition the graph, and a lower replication ratio indicates a smaller index size, which should translate to faster query times in practice. Table 10.10 compares PuLP-MM with METIS-M and random partitioning for 16 and 64 parts on the 3 RDF graphs. Out of the 6 total graph-part count scenarios, the PuLP-MM approach shows the lowest replication ratio for half of them. Note that none of these partitioners are explicitly optimizing for this metric, so the performance of PuLP-MM in this instance is indirect.

Partitioning	16-way			64-way		
	BSBM	LUBM	DBpedia	BSBM	LUBM	DBpedia
Random	7.256	10.58	5.580	22.07	34.84	10.50
METIS	5.566	9.714	**1.552**	19.02	36.56	2.257
METIS-M	5.577	9.146	**1.552**	19.01	38.02	**2.255**
PuLP-M	5.308	**8.944**	1.905	14.73	36.86	2.815
PuLP-MM	**5.112**	9.227	2.448	**13.78**	**29.94**	2.963

Table 10.10. Distributed RDF store replication ratios using various partitioning strategies. An undirected 2-hop guarantee is enforced. Lower values are better and best value for each graph and parts count is in bold.

In Table 10.11, we report sum of query times of RDF3X-MPI averaged over the BSBM, LUBM, and DBpedia data sets. We use a selection of queries from the Berlin SPARQL Benchmark. We use the 16 part partitions for this test and additionally look at the performance affects of the three ordering strategies. PuLP-MM partitioning with random ordering yields the best performance, while PuLP-MM further demonstrates the highest performance when using the other two ordering strategies as well. This corresponds to PuLP-MM having the lowest replication ratios. We note that since PuLP-MM is faster and much more memory-efficient than METIS, this is a promising result, and future work can attempt to optimize PuLP for the one and two hops replication ratio metrics for further improvements. The effect of ordering strategy on query times is interesting, in that the higher-locality orderings demonstrate correspondingly worse performance. To store the RDF data, RDF3X-MPI converts the input RDF graph structure into multiple indexes, which are created by sorting the RDF data, creating B+ trees, and then performing compression. We note that the worsened performance

with locality-optimized ordering is most likely an artifact of this pre-processing stage. This further indicates that knowledge of a graph analytic's algorithmic details is important when determining an optimal graph layout, as unexpected and counter-intuitive performance impacts are a real possibility.

Partitioning	Ordering		
	Random	DGL	RCM
Random	3.41	4.58	4.32
PuLP-M	3.41	3.97	3.94
PuLP-MM	**3.32**	4.01	3.51
METIS	3.71	4.41	3.91
METIS-M	3.87	4.20	4.13

Table 10.11. Total query times in seconds relative for the various partitioning and ordering strategies, summed over all 3 graphs with 16 parts.

10.7 Related Work

We selected METIS and RCM for comparison to the partitioning and ordering aspects of DGL, as they represent the most popular and current state-of-the-art approaches for these problems in terms of both speed of computation and quality produced. There are various other partitioning algorithms and methods, including multi-level partitioners similar to METIS [27, 162, 201], coordinate and geometry-based partitioners [202, 203], and hypergraph partitioners [204]. Hypergraph partitioners can often calculate higher quality partitions than graph partitioners for regular matrices, but at a considerably higher cost to compute. Other graph partitioners have utilized label propagation in single or multilevel approaches [161, 163–166], demonstrating improved algorithm execution times with these partitions versus naïve methods. However, while some of these partitioners produce very high partition quality with good computational efficiency [163], they only consider single constraint partitioning scenarios. As we've demonstrated, using multi-constraint partitioning is important for optimal algorithm performance. Other recent work [205] has correspondingly demonstrated that complex partitioning objectives beyond simply edge cut and/or communication volume are a necessary consideration for optimal performance in distributed computations.

In addition to RCM ordering, Cuthill-McKee (CM) [183], nested dissection [167], and Approximate Minimum Degree (AMD) [184] are a few examples of sparse matrix reordering strategies used in the past. Some techniques, such as space-filling curves [206] or spectral bisection and orderings based on calculated eigenvectors [207] have been utilized for both partitioning and ordering of sparse matrices. These and similar methods and their variants are often very effective for improving cache performance for computations on regular graphs and meshes [208]. Ordering methods on irregular networks such as social and Internet graphs has been studied for the purposes of visualization [209] and compression [180, 200, 210]. Although the authors know of no performance analysis of the effects of applying these ordering techniques to distributed graph computation in literature, promising future work might involve utilizing and optimizes these ordering for such purposes.

10.8 Conclusions

In this chapter, we presented DGL, a methodology for distributed *graph layout* (partitioning, vertex ordering) using the scalable PuLP partitioner and a new vertex ordering strategy. We give a comprehensive performance analysis by examining the effects of graphs layouts on several varied graph analytics. In general, we note that graph analytics which have a high relative computation cost can benefit greatly from our locality-optimizing vertex ordering strategy. Graphs analytics that have a relatively high communication volume but few synchronizations might benefit most from a partitioning that optimizes explicitly for edge cut, while computations which consist of numerous synchronizations would benefit from a more balanced partitioning in terms of per-task communication loads. In general, the higher the computation/communication cost for a given analytic, the more that partitioning and ordering makes an impact. While the layout of a graph in distributed memory can significantly impact the performance of analytics processing on it, the actual implementation methodology of the analytic algorithm can have an even greater impact. In the next chapter, we'll describe in detail a simple and effective methodology for the implementation of distributed-memory graph analytics that is applicable to a broad algorithmic class.

Chapter 11
Distributed Graph Processing

11.1 Introduction

This chapter further builds upon the ideas previously discussed in how to develop generalizable approaches for graph processing. Here, instead of focusing on single-node performance with multicore and manycore processors, we consider system-level performance of a large high performance computing system. We identify disparate algorithm classes based on distributed communication patterns and develop optimized baseline approaches for these classes. We then take the baseline approaches and use it to implement several algorithms fitting each of the classes. We will show that our techniques are highly effective, giving orders-of-magnitude performance improvements over prior art, enabling the distributed analysis of massive scale networks.

11.2 Graph Processing on HPC

The growth of the Internet along with online social networks has motivated considerable interest in the study of large-scale and irregular graphs. Understanding the structure of such graphs, of any scale, has many uses. Being able to efficiently search such irregular data sets to extract useful information is perhaps one of the main applications. Research in the areas of social and web graph analysis has led to the creation of several frameworks designed to make the study of such graphs easier and faster, be it through simplified programming models, abstraction of parallelism, exploiting latent graph structure, or processing large computations with-

out specialized hardware. These frameworks include those both targeted towards large-scale distributed processing, such as Pregel [37], Giraph [36], Trinity [39], GraphLab [32], PowerGraph [33], PowerLyra [34], PEGASUS [38], and others, as well as those designed for highly-efficient shared memory processing, such as Galois [211], Ligra [115], STINGER [212], and FlashGraph [35].

In this chapter, we study well-known computations on such graphs from a High Performance Computing (HPC) perspective. While our main goal is to provide an informed commentary on the efficiency, scalability, and ease of implementation of graph analytics on current high-end computing systems, we also demonstrate how the same techniques we apply to high-end systems can also accelerate analytics at the smaller scale. We consider the largest publicly-available hyperlink graph: the 2012 Web Data Commons graph[1], which was in-turn extracted from the open Common Crawl web corpus[2], and we design clean-slate parallel algorithms and implementations for the Blue Waters supercomputer, one of the world's most powerful computing platforms. After demonstrating scalability on up to 65,536 cores of Blue Waters, we then show how our implementations, without large modifications, can also outperform state-of-the-art frameworks on a small testbed cluster.

As there is a lot of current work on graph algorithms, analytic frameworks, and graph data management systems, the following aspects motivate our present work and differentiates it from other related research efforts:

- Most parallel computing research efforts focus on a single computational routine and study its scalability for a collection of large graphs [157, 188]. Instead, we want to simultaneously analyze performance of multiple analytics, and make decisions about graph data representations and decomposition strategies based on our findings. To help with this, we have picked some of the largest possible real-world graphs that are publicly available.

- Synthetic graphs can never substitute real-world graph instances, as there are always some topological aspects that parsimonious models miss. For real-world graphs, we may be able to *exploit more structure* when designing analysis algorithms. There may also be some intricacies associated with real-

[1] http://webdatacommons.org/hyperlinkgraph/
[2] http://commoncrawl.org

world graphs that do not manifest in synthetic graphs. One of our objectives is to precisely quantify some challenges for a collection of graph analytics on an extremely large graph. We also want to identify known optimizations that work for such graph instances.

- Algorithms that are commonly used for graph analytics have per-iteration operation counts that scale linearly (in the asymptotic case) with the number of vertices and edges. Furthermore, memory requirements and communication costs are also linear. Thus, the constant factors in the implementations are what lead to large performance gaps on real systems. Distributed graph analytic frameworks that provide linear-work algorithms should thus be evaluated on the largest-available graphs.

- I/O costs are often ignored when doing in-memory graph analytics. There is also quite a lot of research on external and semi-external memory graph algorithms and frameworks, where minimizing I/O costs, and not wallclock time, is the primary focus. An end-to-end evaluation of multiple analytics, considering I/O, memory, and network costs, would be more representative of real-world performance.

This chapter presents a methodology for in-memory, end-to-end analysis of large publicly-available data sets. We consider parallel I/O, graph construction, and running multiple useful analytics. In addition, this chapter makes the following new algorithmic and parallel implementation contributions to analyze massive graphs:

- We present a fast and memory-efficient scheme to read and store massive graphs in a distributed setting.

- We present algorithms for analyzing the connectivity, centrality rankings, PageRank, and community structures of such graphs.

- We describe the approaches for our implementations in detail and present optimizations applicable across a broad class of graph algorithms.

We also claim that graph analysis at large-scale concurrencies need not be daunting. All the analyses discussed in this chapter fit into less than 2000 C++ source lines and use only MPI and OpenMP for parallelization, with no other

external library dependencies. Each analytic itself is only around 200 lines of code. If one makes informed algorithmic and data structure choices, end-to-end execution times can be quite fast. Using just 256 compute nodes of Blue Waters, we are currently able to perform all six implemented analytics in under 20 minutes, and this includes graph I/O and preprocessing. Using our analytics, we are additionally able to obtain new insights into the global structure of the web graph. Some of these insights are from, to our best knowledge, the first in-memory global community structure experiment on this massive graph.

11.3 Design Choices and Optimization

We'll now describe our end-to-end methodology for storing, reading, and performing distributed computations on the web crawl. We will describe our distributed graph storage format as well as our implemented algorithms and their optimizations. While our largest example problem is the web crawl, the techniques we describe here are applicable to many large real world graph instances and HPC systems.

11.3.1 I/O and pre-processing

We store the web graph in binary format as a list of edges, where each directed edge is represented by two vertices of 32-bit `unsigned integer` type $\langle v_0, v_1 \rangle$ (64-bit integers would be required for a graph with a number of vertices exceeding 2^{32}). We ingest data by calculating offsets for each MPI task, with each task being given a nearly-identical portion of the file to read. To achieve high read bandwidth from Blue Waters' shared Lustre-based scratch filesystem, we striped the input file across storage units. In the web graph case it is a 1 TB file across 160 storage units.

We chose a memory-efficient *one-dimensional* graph representation in this work (Section 11.3.3), where each MPI task owns $\frac{n}{p}$ vertices and all of the incoming and outgoing edges of these vertices. After each task reads its share of outgoing edges, the edges are exchanged using an MPI `Alltoallv` step. We then proceed to reverse the edges and do another `Alltoallv` exchange. Once each task has all of the outgoing and incoming edges for the vertices owned by this task, they can then convert the edge arrays into a compressed sparse row (CSR)-like representation.

The graph ingestion and creation stage is the most memory-intensive part of

our implementation. To hold the outgoing edge list in memory, we require $8m$ *bytes* of global memory, where m is the number of edges in the graph. This implies approximately 1 TB of aggregate memory. In order to use MPI collectives in the edge exchange step, we also need to create send and receive buffers, which require an additional $16m$ *bytes* of memory. However, the necessary memory will be multiplied by the maximum edge imbalance among all tasks. This is an important consideration for partitioning graphs with a skewed degree distribution and it is dependent on partitioning strategy. Using a one dimensional vertex block partitioning with the web crawl, we observe up to 2× edge imbalance with a task counts between about 100 and 512. This effectively bounds the minimum number of tasks required to be about 188, which is also observed in practice.

11.3.2 Partitioning Strategy

The performance and scalability of distributed algorithms can be highly dependent on the chosen partitioning strategy. For graph algorithms, the effect of partitioning can be even more drastic, as the ratio of computation versus communication can be significantly lower than traditional scientific applications. The most common partitioning strategy for graph analytics is a one dimensional variant, where each task owns some subset of vertices in the graph and all outgoing and incoming edges for those vertices. Ideally, it is desired for each task to own equivalent numbers of vertices and edges, with the ratio of internal edges (edges between owned vertices) to external edges (edges to vertices owned by another task) as high as possible. The number of external edges is commonly called the edge cut.

Current traditional high quality partitioners are unable to process graphs of the scale being considered. As such, we consider three primary partitioning strategies in this work. We implement *vertex block* partitioning, where each task gets $\frac{n}{p}$ vertices distributed in natural (or some computed) ordering, *edge block* partitioning, where each task gets approximately $\frac{m}{p}$ edges again distributed with some ordering, and *random* partitioning, where each vertex is randomly assigned to a task. In general, there can be significant edge imbalance among tasks with vertex block partitioning and vertex imbalance with edge block partitioning. This can make time to complete computational stages highly variable, which increases idle time at synchronization points for all tasks except the one with the highest workload.

For random partitioning, there is generally a reasonable balance among tasks (though this can be dependent on the graph's degree skew and the number of tasks). However, random partitioning suffers from an inherent lack of intra-task and inter-task locality, which increases computation load within a task and communication load among tasks.

11.3.3 Distributed Graph Representation

The design choices for our distributed graph representation have two primary goals: compactness in memory and speed of access for any task-local graph information. Table 11.1 gives an overview of the primary structural information we store.

Table 11.1. Distributed Graph Representation.

Data	Size	Description
`n_global`	1	Num. of global vertices
`m_global`	1	Num. of global edges
`n_loc`	1	Num. of locally owned vertices
`n_gst`	1	Num. of ghost vertices
`m_out`	1	Num. of locally owned out edges
`m_in`	1	Num. of locally owned in edges
`out_edges`	`m_out`	Array of out-edges
`out_indexes`	`n_loc`	Start indices for local out-edges
`in_edges`	`m_in`	Array of in-edges
`in_indexes`	`n_loc`	Start indices for local in-edges
`map`	`n_loc+n_gst`	Hash table for global to local ids
`unmap`	`n_loc+n_gst`	Array for local to global ids
`tasks`	`n_gst`	Array for tasks of ghost vertices

It is common for many graph algorithms to have tasks repeatedly access and update data stored per-vertex for both local and ghost vertices. Having each task store this data in an `n_global` length array is not scalable, so a common strategy is to use a hash map. To avoid accessing a slow hash map and using arrays instead, we relabel all locally owned and ghost vertices. Local vertices are relabeled from [0 to (`n_local`-1)], while ghost vertices assume labels from [`n_local` to (`n_local`+`n_ghost`)]. This relabeling is done to the out- and in-edge arrays. We can then store and access any vertex information in an (`n_local`+`n_ghost`) length array.

To look up the local label for any global vertex id, such as when receiving a message from a neighboring task, we utilize a fast linear-probing hash map (`map`[`global_id`] = `local_id`). To map from local vertex ids to global vertex ids, such as when sending a message about per-vertex information to a neighboring task, we have the `unmap` array (`unmap`[`local_id`] =`global_id`). Finally, we also have an array of length `n_ghost` that stores which task owns each local ghost vertex. Although with simple block partitioning, we can easily calculate this task using the `global_id` on-the-fly, for more complex partitioning or re-ordering scenarios, we are required to hold this information.

The per-task memory storage requirement for this distributed representation is dependent on the partitioning strategy. A vertex block partitioning will have imbalance among tasks with the in- and out-edge arrays, an edge block partitioning will have imbalance with the indexing and `unmap` arrays as well as the hash map, and a random partitioning will have imbalance through having a much higher number of ghost vertices. The actual level of imbalance is highly dependent on graph structure.

11.3.4 Implemented Algorithms

We'll now describe some of the algorithm-specific optimizations we implemented. For these optimizations, we strove for a strong balance between high performance, relative ease of implementation, and applicability across a large number of analytics. For ease of discussion, we consider that our implementation efforts focus on what can be reduced to two distinct but closely related classes of graph algorithms. The first class of algorithms have all vertices propagating per-vertex information to all neighbors. We consider our implementations of PageRank and the Label Propagation community detection algorithm [145] to fall under this class. The second class includes algorithms which begin by propagating information from an original root through it's neighbors with what can be considered a global queue. This second class doesn't necessarily pass per-vertex information, and can instead only pass vertex identifiers themselves, with per-vertex data updated locally. These algorithms use some derivate of breadth-first search (BFS) as a central subroutine. Our implemented analytics that use this approach include routines for extracting the strongly connected component (SCC) containing a given vertex using the

Forward-Backward approach [19], a routine for determining approximate k-cores (a maximal subgraph in which all vertices have degrees of k or greater), as well as a routine for calculating the harmonic centrality [213] for a given vertex. Additionally, we have implemented the Multistep [21] algorithm for weakly connected component (WCC), which has its first stage similar to the algorithms in latter class (BFS-like) and second stage similar to the algorithms in the former class (PageRank-like).

Both of the aforementioned classes can be considered to fall under a broader class of graph algorithms that follow a tri-nested loop structure, with loops over some number of iterations, vertices in a queue or in the entire graph, and a final loop of the edges of the vertices being considered in the middle loop [45]. This structure both fits a very large number of graph algorithms and is amenable to a bulk synchronous parallel (BSP) model of parallel computation due to the dependencies imposed by the outer iterative loop. We admit by no means that all graph algorithms fit the aforementioned classes, but a great many more that have not been explicitly implemented in this current work might benefit from our described optimizations.

11.3.4.1 Algorithm Overviews

As mentioned, we implemented WCC, SCC, harmonic centrality, an approximate k-core decomposition algorithm, PageRank, and the label propagation community detection algorithm [145]. We use a BFS-based approach for finding the largest weakly-connected component (WCC) and strongly connected component (SCC). For SCC, we utilize the Forward-Backward approach [19]. To find the remaining WCCs, we use the Multistep algorithm [21]: after removing the largest WCC, we then perform color propagation to find the remaining smaller components. We calculate harmonic centrality for a given vertex in the standard way, by calculating the distances of all vertices that can reach the given vertex using a BFS and then performing a sum of the inverses of all the distances for all of these vertices. A k-core of a graph is the . To calculate approximate k-core scores, we create a set of seed vertices using the 1000 largest in-degree vertices and use those seeds to iteratively identify maximal components containing vertices of some minimal power-of-two degree. This effectively gives us an upper bound on true k-core values. For our label propagation implementation, we use an approach similar to the constrained label propagation implementation by Meyerhenke et al. [162]. We ignore directivity

of edges and consider them all undirected; labels can propagate in either direction. We calculate PageRank using the standard original algorithm.

11.3.4.2 PageRank-Like Algorithms

We'll now describe in more detail our PageRank, label propagation, and the second phase of our weakly connected components algorithms. As mentioned, the first class of algorithms we consider potentially propagate some stored per-vertex value to all neighbors on every iteration. For PageRank, this would be the vertex's PageRank value to all of its out-edge neighbors. For label propagation and weakly connected components, this would be the vertex's current label or color being passed to both in- and out-edge neighbors. Our implementations of these algorithms all follow a consistent pattern. We demonstrate this pattern by using PageRank as our example in Algorithm 11.1. The algorithm is run on each MPI task, with G being the task-local distributed graph representation and V and E being the local vertex and edge sets, respectively.

We initially pass our per-vertex information in two queues. One queue holds the global vertex ids being passed ($vSend$) and the other queue holds the associated PageRank values for each vertex ($pSend$). Due to the nature of small world graphs, subdomain connectivity is extremely high among different task partitions. As such, we utilize MPI Alltoallv collectives in lieu of explicit sends and receives to simplify the communication process. Alltoallv requires us to calculate the number of items being passed to each task ($NumSend$) as well as the offsets in the send queues for where the items for each task begin ($SendOffs$). To initialize our queues, we therefore need to loop over all local vertices and edges to first count the numbers of items being sent and then to actually place the items in our queues. We utilize a boolean array ($ToSend$) while examining the edges of each vertex ($v \in V$) to track which tasks we've already going to send our vertex information to. We then calculate prefix sums of the per-task counts to create our offset array. During the second loop through, we calculate the initial PageRank for each vertex, and then place that PageRank as well as the global id of the vertex into the actual send queues using a copy of the offsets array ($SendOffsCpy$) where we increment the current offset after each item placement. For label propagation and WCC, we would be initializing and sending labels and colors in lieu of PageRanks in our queue. We perform thread-based parallelization over these loops. It is nontrivial,

Algorithm 11.1 Distributed PageRank

```
1: procedure PageRank-dist(G(V, E), δ)
2:     nprocs ← numTasksMPI()
3:     procid ← localTaskNum()
4:     NumSend(1 ··· nprocs) ← 0
5:     for all v ∈ V thread parallel do
6:         ToSend(1 ··· nprocs) ← false
7:         for all u ∈ E(v) do
8:             t = getTask(u)
9:             if t ≠ procidand ToSend(t) ≠ true then
10:                ToSend(t) ← true
11:                NumSend(t) ← NumSend(t) + 1
12:    SendOffs(1 ··· nprocs) ← prefSums(NumSend)
13:    SendOffsCpy ← SendOffs
14:    for all v ∈ V thread parallel do
15:        PageRanks(v) ← initPR(v)
16:        ToSend(1 ··· nprocs) ← false
17:        for all u ∈ E(v) do
18:            t = getTask(u)
19:            if t ≠ procidand ToSend(t) ≠ true then
20:                ToSend(t) ← true
21:                vSend(SendOffsCpy(t)) ← globalId(v)
22:                pSend(SendOffsCpy(t)) ← PageRanks(v)
23:                SendOffsCpy(t) ← SendOffsCpy(t) + 1
24:    vRecvs ← Alltoallv(vSend, NumSend, SendOffs)
25:    pRecvs ← Alltoallv(pSend, NumSend, SendOffs)
26:    for i ··· |vRecvs| thread parallel do
27:        vIndex = localId(vRecvs(i))
28:        vRecv(i) = vIndex
29:        PageRank(vIndex) = pRecvs(i)
30:    for i = 1toiter do
31:        for all v ∈ V thread parallel do
32:            PageRanks(v) ← CalcPR(v, E′(v))
33:        for i ∈ |vSend| thread parallel do
34:            pSend(i) ← PageRanks(vSend(i))
35:        pRecvs ← Alltoallv(pSend, NumSend, SendOffs)
36:        for i ∈ |vRecv| thread parallel do
37:            PageRanks(vRecv(i)) ← pRecv(i)
38:    return PageRanks
```

so we'll describe how we implement it in detail in a later section.

Once the second loop over all vertices and edges completes, we perform our initial sends and receive into buffers for vertices ($vRecv$) and PageRanks ($pRecv$). We then update all of our local PageRank values based on the vertices and associated values in the queue. When we first examine $vRecv$, the vertices are in the buffer as global ids. We first must convert them back to their local ids using the hash map in our distributed graph representation. Since this implementation assumes all PageRank values will be updated on each iteration, we can utilize a couple of optimizations. We first cut the size of data being sent in half for each iteration by retaining the vertex queue and only updating and sending the PageRank queues. Since we don't update the vertex queue, we simply save in $vRecv$ the local ids in place of the global ids for each vertex and use those when doing the PageRank updates on subsequent iterations. This avoids multiple (relatively) costly hash map accesses. Be retaining queues, we also avoid having to completely rebuild them on each iteration. Experimentally, a certain cutoff can be determined for when it would be better in terms of the computation-communication trade-off to switch from retaining the queues to rebuilding them.

The bulk of time during the run of the algorithm is spent in the main loop, shown in Algorithm 11.1 as $i = 1toiter$ (a stopping criteria other than fixed iterations is also common for this loop). This loop has four main phases. First, we update our PageRank values for all local vertices. Second, we loop over the size of our retained send queues to update the values in $pSend$. We then perform the exchange of the updates value among all tasks. We finally loop to update the PageRanks with those received in $pRecv$ using our retained $vRecv$. All of these three primary loops are trivial to parallelize in shared memory, and we observe consistent and high speedups when doing so. Our WCC and label propagation algorithms follow an identical pattern to PageRank, with the primary difference being how we initialize and calculate updates for the per-vertex values.

11.3.4.3 BFS-like Algorithms

As mentioned, the second class of implemented algorithms we consider don't explicitly pass per-vertex information, instead creating a queue only of vertices, with all per-vertex updated happening locally on each task. Updates might be visitation statuses, levels in a BFS tree, or some other data. We demonstrate

the general outline used for our implementations in Algorithm 11.2 with a BFS that tracks distances from a root vertex. Although a lot of recent work has focused on optimizing distributed BFS for the Graph500, we omit any BFS-specific optimizations in our discussion and focus on those generalizable to all of the algorithms we are considering.

As Algorithm 11.2 demonstrates, we utilize task-local queues (Q) as a basis for the work to be performed during each iteration. The queue is initialized first with the root vertex for the task that owns it, and we use task parallelism when looping over the contents of the queue. We use an array ($Status$) to determine visitation status (as needed for WCC). In this instance, $Status$ is also used to hold the distances from the root for each visited vertex (as needed for harmonic centrality). The $Status$ value is set to -2 initially, and updated to -1 when the vertex has been visited. This first update is done to signify that the vertex has either been added to the local queue for the next level or the send queue for a neighboring task, so the exploration of subsequent edges incident on the vertex don't end up re-queuing that vertex. If the vertex is local, the $Status$ value will be updated to the current $level$ on the next iteration. We track the number of vertices to send to each neighboring task ($NumSend$) during exploration, and then create a send queue and perform an Alltoallv collective at the end of the iteration similar to how we create the original queue with PageRank. We also track the sum size of all queues over all tasks ($globalSize$) and use it as a stopping criteria for the algorithm (i.e. there are no more vertices left to explore on any task). Again, our implemented algorithms that follow this template are extremely parallelizable, with the only serial portions being the global communication operations. We'll describe in a bit more detail how we handle the queues within shared memory next.

11.3.5 MPI+OpenMP

Due to the overheads associated with handling graphs in distributed memory, which tend to increase with an increasing MPI task count, utilizing all available intra-node parallelism is extremely important for maximizing performance. As mentioned, we employ OpenMP shared-memory parallelism to allow us all available cores while keeping MPI task counts as low as possible. The biggest concerns with this model are the addition of synchronization overheads among threads as well as minimizing

Algorithm 11.2 Distributed BFS

```
1: procedure BFS-DIST(G(V, E), root)
2:     nprocs ← numTasksMPI()
3:     procid ← localTaskNum()
4:     Status(1 ··· v) ← −2
5:     level = 0
6:     globalSize = 1
7:     if getTask(root) = procid then
8:         Q ← root
9:     while globalSize ≠ 0 do
10:        NumSend(1 ··· nprocs) ← 0
11:        for all v ∈ Q thread parallel do
12:            if Status(v) ≥ 0 then
13:                continue
14:            else
15:                Status(v) ← level
16:            for u ∈ E(v) do
17:                if Status(u) = −2 then
18:                    Status(u) ← −1
19:                    t = getTask(u)
20:                    if t = procid then
21:                        Q_next ← u
22:                    else
23:                        Q_send ← u
24:                        NumSend(t) ← NumSend(t) + 1
25:        SendOffs(1 ··· nprocs) ← prefSums(NumSend)
26:        SendOffsCpy ← SendOffs
27:        for all v ∈ Q_send thread parallel do
28:            t = getTask(v)
29:            vSend(SendOffsCpy(t)) ← globalId(v)
30:            SendOffsCpy(t) ← SendOffsCpy(t) + 1
31:        vRecv ← Alltoallv(vSend, NumSend, SendOffs)
32:        for i ∈ |vRecv| thread parallel do
33:            Q_next ← localId(i)
34:        swap(Q, Q_next
35:        globalSize ← Allreduce(|Q|, SUM)
36:        level ← level + 1
```

the effects of intra-node load imbalance.

All of our implemented algorithms use some form of a queue for passing updates

between tasks. Having multiple threads concurrently writing to a shared queue is not necessarily scalable in terms of both cache utilization efficiency and the number of atomic operations required. To minimize the impact of these unavoidable issues, we implement thread-local queues. To demonstrate our thread-local queue implementation, we explicitly show how they are used during the initialization of the *vSend* and *pSend* queue for PageRank in Algorithm 11.3. This method is generalizable for updating queues in all of our algorithms.

For Algorithm 11.3, we assume first that $NumSend$, $SendOffs$, and $SendOffsCpy$ have already been initialized. We then initialize thread-owned queues $vSend_t$ and $pSend_t$ that will be used to temporarily hold updates to the *vSend* and *pSend* queues. The size of these thread-owned queues will optimally be dependent on CPU architecture and cache size. We also have another thread-owned array $NumSend_t$ that holds per-task counts for the current contents of the thread's queues. When a thread fills its queue, it pushes updates to the global queues. First, it does the equivalent of an atomic fetch-and-add to get an offset (o_t) for the location in the send queue for each task. We can just overwrite the values in $NumSend_t$ to hold these offsets. It then goes through the contents of its queues, placing the vertices and per-vertex PageRank values into the task-level queues starting at the proper offsets. When a thread finishes all of it's work during the main loop, it empties whatever remains of its queue in the same way. This straightforward strategy can improve cache performance and greatly decrease the total number of atomic operations when using thread queues. We utilize variants of this strategy within each of our implemented algorithms.

11.4 Data and Setup

We primarily use the NCSA Blue Waters supercomputer for our large scale graph analysis. Blue Waters is a hybrid Cray XE6/XK7 system with around 22,500 XE6 compute nodes, and 4200 XK7 compute nodes. Each XE6 node contains two AMD Interlagos 6276 processors. There are four NUMA domains per node, each with four cores and sharing a 8 MB L3 cache. The memory capacity of each node is 64 GB and the peak memory bandwidth is 102.4 GB/s. We do not use the GPU-accelerated XK7 partition of Blue Waters. One of the main reasons we chose Blue Waters was the high-performance file system. The Lustre-based scratch file

Algorithm 11.3 Distributed PageRank initialization demonstrating OpenMP thread queuing.

```
1: procedure PageRank-dist(G(V, E), δ)
2:     nprocs ← numTasksMPI()
3:     procid ← localTaskNum()
4:     ···
5:     begin parallel
6:     vSend_t[Q_SIZE] ← ∅
7:     pSend_t[Q_SIZE] ← ∅
8:     NumSend_t(1 ··· nprocs) ← 0
9:     for all v ∈ V thread parallel do
10:        PageRanks(v) ← initPR(v)
11:        ToSend(1 ··· nprocs) ← false
12:        for all u ∈ E(v) do
13:            t = getTask(u)
14:            if t ≠ procid and ToSend(t) ≠ true then
15:                ToSend(t) ← true
16:                vSend_t ← globalID(v)
17:                pSend_t ← PageRanks(v)
18:                NumSend_t(t) ← NumSend_t(t) + 1
19:                count ← count + 1
20:                if count > Q_SIZE then
21:                    for i = 1 ··· nprocs do
22:                        atomic capture
23:                        o_t ←
24:                            SendOffsCpy(t) += NumSend_t(t)
25:                        NumSend_t(t) ← o_t − NumSend_t(t)
26:                    for i = 1 ··· count do
27:                        tt ← getTask(vSend_t(i))
28:                        vSend(NumSend_t(tt)) ← vSend_t(i)
29:                        pSend(NumSend_t(tt)) ← pSend_t(i)
30:                        NumSend_t(tt) ← NumSend_t(tt) + 1
31:                    NumSend_t(1 ··· nprocs) ← 0
32:    for i = 1 ··· nprocs do
33:        atomic capture
34:            o_t ← SendOffsCpy(t)+ = NumSend_t(t)
35:        NumSend_t(t) ← o_t − NumSend_t(t)
36:    for i = 1 ··· count do
37:        tt ← getTask(vSend_t(i))
38:        vSend(NumSend_t(tt)) ← vSend_t(i)
39:        pSend(NumSend_t(tt)) ← pSend_t(i)
40:        NumSend_t(tt) ← NumSend_t(tt) + 1
41:    barrier
42:    ···
43:    return PageRanks
```

system uses 180 Scalable Storage Units (SSUs) and the rated I/O bandwidth is

a remarkable 960 GB/s. We compile our C++ programs using Intel's compilers (version 15.0.0) and use the Intel Programming Environment (version 5.2.40) and Cray-MPICH version (7.0.3). Some of our experiments were also run on the Sandia National Labs *Compton* testbed. Each node of Compton is a dual-socket system with 64 GB main memory and Intel Xeon E5-2670 (Sandy Bridge) CPUs at 2.60 GHz and 20 MB last-level cache running RHEL 6.1.

Graph	n	m	D_{avg}	Source
Web Crawl	3.5 B	129 B	36	[214]
R-MAT	3.5 B	129 B	36	[87]
$G(n,p)$	3.5 B	129 B	36	
R-MAT	2^{25}-2^{32}	2^{29}-2^{36}	16	[87]
$G(n,p)$	2^{25}-2^{32}	2^{29}-2^{36}	16	
Host	89 M	2.0 B	22	[214]
Pay	39 M	623 M	16	[214]
Twitter	53 M	2.0 B	38	[73]
LiveJournal	4.8 M	69 M	14	[70]
Google	875 K	5.1M	5.8	[70]

Table 11.2. Real world and generated graphs used during experiments.

For our large-scale experiments, we run our analysis codes on the largest publicly available graph known to the authors: the 2012 Web Data Commons hyperlink graph. The 2012 graph is available for download with three levels of aggregation: at page-level, at the granularity of subdomains/hosts, and at the granularity of pay-level-domain (PLD). We primarily work with the page-level graph in this chapter, but use the smaller graphs for comparisons to related work. Also for our comparitive analysis, we run on the LiveJournal and Google graphs from SNAP [69, 70] and the Twitter crawl from the Max Planck Institute [73]. For additional large-scale experiments, we generated an R-MAT and Erdös-Rényi $G(n,p)$ graphs of equivalent size to the web crawl for comparative analysis. We also generated R-MAT and $G(n,p)$ graphs of scale 25 to 32 for a weak scaling analysis. The R-MAT graphs were generated with Graph500 parameters (A=0.57, B=0.19, C=0.19, D=0.05). We do not pre-process or prune the graphs in any way, and use the ordering as given from the original source (or as generated) when determining vertex and edge block partitionings. A summary of all graphs is given in Table 11.2.

11.5 Performance Results

11.5.1 End-to-end Analytic Execution Time

We demonstrate the performance of the data ingestion and graph creation phases. Column 2 of Table 11.3 lists the total web graph I/O read time when varying the number of compute nodes on Blue Waters, with a single MPI task per node. These times correspond to read bandwidths between 20-30 GB/s, which is under a minute total to read the edges into memory. We primarily note that using a larger number of tasks generally corresponds to faster I/O. This is possibly due to the lower read volume requirement for each task, so slowdowns from a single task due to network traffic and concurrent file system accesses end up having a lesser effect on total time. Table 11.3 also shows the running times in seconds for exchanging both outgoing and incoming edges, as well as the time to create the final distributed CSR representation when varying the numbers of tasks. We note a degree of strong scaling with increasing task count. We also include in Table 11.3 a performance rate in billions of edges processed per second (GE/s), corresponding to the total number of edges processed (128 billion in- and 128 billion out-edges).

#	Time (s)				Perf	Speedup
Nodes	*Read*	*Excg*	*LConv*	*Total*	**Rate (GE/s)**	
256	47	109	41	197	1.30	1.00×
512	45	90	33	168	1.52	1.17×
1024	42	61	27	130	1.97	1.51×
2048	34	55	24	113	2.27	1.74×
4096	39	68	23	130	1.97	1.51×

Table 11.3. Parallel performance for various stages of graph construction.

Table 11.4 gives the total execution times of all of our algorithms with the web crawl running on 256 nodes of Blue Waters with the various partitioning strategies (WC-np, WC-mp, WC-rand). We also include times for the equivalently-sized R-MAT and $G(n, p)$ graphs; we just use vertex block partitioning for the synthetic graphs. We note the total end-to-end execution time for the vertex block partitioning strategy (WC-np) to be about 20 minutes when including I/O and preprocessing. The k-core and label propagation algorithms are noted as taking the longest computation time, as k-core requires multiple iterations of breadth-first

searches and label propagation is the most computationally challenging due to the repeated hash map accesses on its inner loop. However, both of these algorithms each still finish in well under 10 minutes under all three partitioning sceneries with the hyperlink graph. The synthetic graphs take longer to perform label propagations due to lack of good locality in either graph, and additionally poor load imbalance for the R-MAT graph.

Analytic	WC-np	WC-mp	WC-rand	R-MAT	$G(n,p)$
PageRank	87	111	227	125	121
Label Propagation	400	435	367	993	992
WCC	19	21	41	59	69
Harmonic Centrality	54	46	101	252	84
K-core	445	363	583	579	481
SCC	184	108	184	89	83

Table 11.4. Exec. Times on 256 Nodes of Blue Waters.

11.5.2 Weak and Strong Scaling

We run two sets of experiments to examine the scalability of our approach. First we'll examine how our algorithms perform under weak scaling on generated graphs. Figure 11.1 gives the execution time of harmonic centrality and PageRank on generated R-MAT and GNP graphs. The R-MAT graphs are generated with Graph500 parameters and all graphs have an average degree of 16. The number of nodes being run on is varied from 8 to 1024 with 2^{23} vertices per node (graphs of scale 25 to 32). We just assume a simple vertex block partitioning for these tests. We observe that our harmonic centrality code scales extremely well on the GNP until 1024 nodes, where communication times for the collective operations begin to increase. We note that the R-MAT graph doesn't scale quite as well, due to communication and work imbalances introduced by the large degree vertices. For PageRank, we note that the code scales moderately well on both graph instances.

Next, we will look at strong scaling at the large scale. Figure 11.2 gives the strong scaling of our label propagation implementation running on 256-4096 nodes of *Blue Waters*. We look at the scaling of the web crawl with the various partitioning strategies as well as generated R-MAT and GNP graphs with equivalent numbers of vertices and edges. We note that our label propagation code scales quite well on the

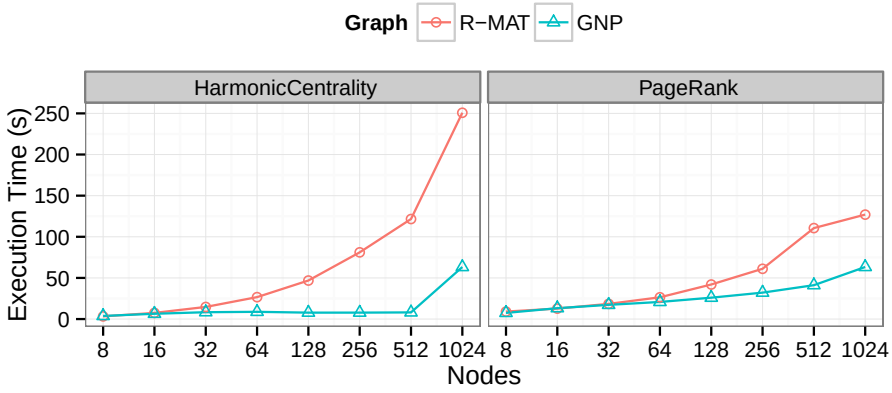

Figure 11.1. Weak scaling of R-MAT and GNP graphs of scale 25 to 32 size running on 8-1024 nodes of *Blue Waters*.

generated graphs and best on the web crawl when using the random partitioning, with a geometric mean speedup of 3.6× across all tests. The scaling drops off on the block partitioning strategies at high node counts due to load imbalance.

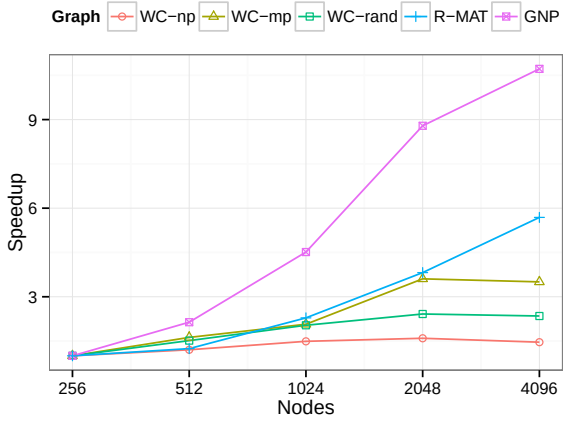

Figure 11.2. Strong scaling of the Web Crawl graph as well as R-MAT and GNP graphs of equivalent size running Label Propagation on 256-4096 nodes of *Blue Waters*.

To more closely examine the effect of partitioning selection on scalability, we break our execution time into three components: time each task spends in computation, time each task spends idle waiting for updates from other tasks, and total time spent in communication. We track the minimum, maximum, and average across all tasks for each of these stages. For the web crawl on 256-4096 these results are reported for PageRank. For consistency, we scale each data point as the execution time ratio for each component relative to the total time.

Looking at Figure 11.3, we note that average relative computational time is

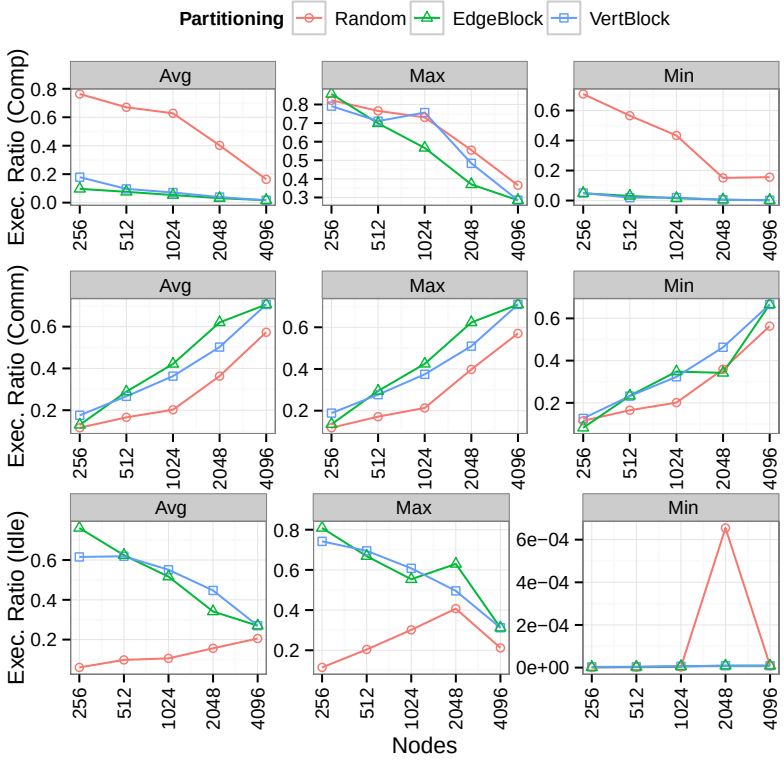

Figure 11.3. Minimum, maximum, and average per-task execution time ratio of computation, communication, and idle times for scaling from 256 to 4096 nodes on Blue Waters.

very high for random partitioning versus the block methods. This is due to two reasons. This first being that we retain the relatively good ordering with the block-based strategies, which gives us good intra-node cache performance. The block strategies also have a lower relative number of ghost nodes and therefore lower number of total global and local id lookups and hash map accesses. The maximal computational ratios are consistent among partitionings due to very high degree vertices that result in long computational times regardless of placement. We note that the communication ratio increases with increasing node count, as is expected when strong scaling. The idle times are mostly related to imbalances in computation, as larger computation imbalance leads to some tasks being delayed at a synchronization or communication point while waiting for the longest running task to complete its portion of work. We in general observe that random partitioning has the lowest average and max idle times as it inherently has the best work balance. Minimum idle times are near zero, as is expected. What appears to be a large

increase in idle time for random at 2048 is just system noise; its unscaled value is still well under one second.

11.6 Comparison to Prior Work

We will now directly compare our code against some of the most popular available frameworks designed for large graph analysis. For direct comparison, we focus on frameworks that utilize in-memory graph storage. We compare to GraphX, PowerLyra, and PowerGraph. GraphX is the Apache Spark API designed for parallel graph computation in memory. PowerGraph and PowerLyra are both GraphLab derivatives designed to more efficiently process graphs with a skewed degree distribution, such as the web crawls and social networks we test on, with PowerLyra offering a more advanced partitioning strategy that differentiates between high and low degree vertices. In addition, we also compare to the external memory framework of FlashGraph, which can utilize arrays of external SDDs, as it is the only other framework known to the authors that has reported performance numbers on the Web Crawl. Although our primary goal was to maximize the balance between ease of implementation and computational efficiency, we admit that there are far more metrics for comparison than what we will show by just comparing execution times (fault tolerance, programmability, etc.).

We perform our comparison on *Compton*. Figure 11.4 gives comparison of our code running on 16 nodes (SRM-16) to GraphX (GX), PowerGraph (PG), and PowerLyra (PL). We also compare our single node performance (SRM-1) to FlashGraph. We compare to FlashGraph running in both in-memory only standalone (FG-SA) and while using external SDD and SAFS (FG) modes. We note that we did not have an SSD array at our disposal for these experiments, as each *Compton* node only has a single SSD attached, so the numbers for FlashGraph running with SAFS are much higher than what is reported by the FlashGraph authors. The numbers might only be considered as what typically would be measured in an average high performance computing setting without access to specialized hardware. We look at the performance of our PageRank and WCC implementations on the Host, Pay, Twitter, LiveJournal, and Google graphs using random partitioning. We compare directly to the supplied implementations of PageRank and (weakly) connected components for each of the frameworks. We

setup each framework to explicitly utilize all available memory and cores on each node when possible.

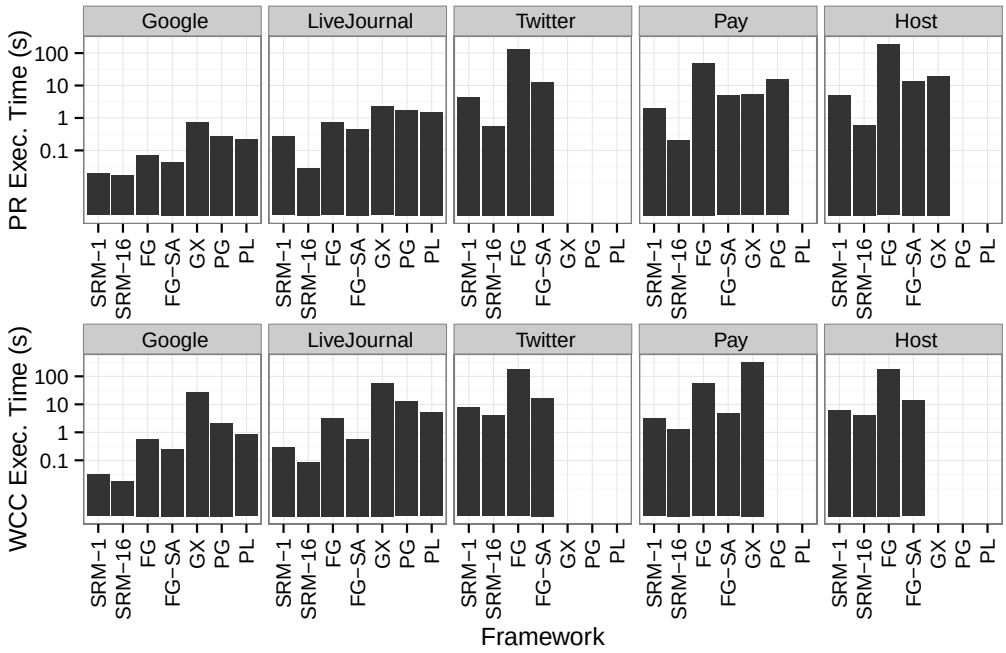

Figure 11.4. Comparison of our code running on *Compton* to various popular graph analytic frameworks.

As can be seen in Figure 11.4, we observe considerably higher performance with our code on both PageRank (top) and WCC (bottom). Running on 16 nodes, GraphX, PowerGraph, and PowerLyra all failed to process some of the larger graphs. None of the three were able to process the Twitter graph for either algorithm. The causes of failure were noted as being out of memory and related errors. Overall, our code offers geometric mean speedups relative to the other distributed codes across all graphs of 38× for PageRank and 201× for WCC (mean of 78× across both algorithms and all frameworks) on 16 nodes. Running on a single node, FlashGraph in standalone mode gave the most comparable results, with our code having a mean speedup of 2.4× and 2.6× relative to FlashGraph-SA and 12× and 19× relative to FlashGraph with SAFS for PageRank and WCC, respectively. We point out that our speedups for WCC are likely much larger than the speedups for PageRank due to our use of the relatively more efficient Multistep algorithm instead of traditional single-stage WCC approaches.

11.6.0.1 Further Comparisons

Other works have recently processed or analyzed graphs of similar scale on similar scale systems, so we will also do some implicit comparisons of our execution times relative to those reported in literature. A recent paper [41] describing Facebook's Giraph processing framework ran label propagation and PageRank on several Facebook snapshots of similar scale to the web crawl. On a 701 M vertex and 48 B edge graph, they report per-iteration times of 9.5 minutes for a label propagation algorithm on 200 nodes. On a 2 B vertex and 400 B edge graph, they report per-iteration times of 5 minutes for PageRank. By comparison, on the 3.5 B vertex and 129 B edge web crawl on 256 nodes, our per-iteration times are 40 seconds and 2.15 seconds for label propagation and PageRank, respectively. Recent related work by Microsoft demonstrated their Trinity graph processing system [39]. They report per-iteration times for PageRank and BFS on 8 nodes for processing an approximately scale 28 R-MAT graph of average degree 13 to be about 15 seconds per iteration for PageRank and 200 seconds for BFS. Re-running the same experiments on 8 nodes of *Compton*, we observe execution times of 1.5 seconds per iteration for PageRank and a total time of about 32 seconds for BFS. Although hardware limitations limited our ability to directly compare against FlashGraph on the web crawl, they report [35] execution times of 461 seconds for WCC and 68 seconds per iteration for PageRank on a test machine with 32 cores and an attached array of 15 SSDs. Our time on 256 nodes for WCC was 19 seconds and for PageRank it was 2.15 seconds per iteration as mentioned.

11.6.0.2 Other Related Work

Orthogonal to the above efforts, there are a number of other graph frameworks for graph analytics in distributed memory systems. Satish et al. [215] compare a subset of these frameworks, hand-optimized code, and other codes. They demonstrate that the performance gap between native code and using a framework is huge: 2-30× in most cases but as high as 300× in some cases. While processing their largest real-world graph of a Twitter crawl with 61 M vertices and 1.4 B edges, their hand optimized version takes ∼3 seconds for an iteration of the PageRank and ∼5 seconds for the BFS using four nodes.

The introduction of the Graph500 benchmark [124] has increased the relevance

of supercomputers for graph analytics. The top ten of the most recent Graph500 list are all supercomputers designed for general problems. The focus of Graph500 is on one benchmark application (BFS) with a synthetic graph (R-MAT). This has lead to renewed interest in the race for achievable GTEPS and algorithms for BFS/R-MAT related problems [157, 173, 188]. All these work focused on scaling the Graph500 benchmark or similar algorithm on various synthetic graphs. They vary on the data layout (vertex vs edge distribution), handling of high degree vertices, communication layer and the BFS algorithm used. These works are some of the largest analysis on synthetic graphs before this work. In contrast, we analyze a real world graph of similar size but use richer analytics. Additionally, some of the optimizations used for the optimized BSF benchmark are not necessarily applicable across a broader algorithmic class.

11.7 Web Graph Analysis

Using our implementations, we were able to explore the structure of the web crawl in detail. We present some of the more interesting results in this section.

11.7.1 Computing Global Statistics

To verify the integrity of the ingested binary web crawl data and get a sense of the global structure, we first computed and now present the out- and in-degree distributions. In Figure 11.5, we plot these distributions as cumulative fractions of total edges versus total vertices, with vertices sorted in decreasing order of degree. We also plot an in-degree distribution of our generated similar-scale $G(n, p)$ random graph.

As was observed by Muesel et al. [216], we see a very skewed degree distribution. A small fraction of vertices account for the vast majority of the total incoming edges. The solid vertical line indicates a 5% cumulative fraction of vertices, which own over 83% of all incoming edges. The highest in-degree vertex in the crawl, `www.youtube.com`, has about 93 million incoming links. The bias for outgoing edges is not as skewed, with 5% of vertices owning 34% of the edges and a maximal degree of about 56 thousand. However, we mention that there were also some settings in the original crawl that prevented exploring pages with a large collection

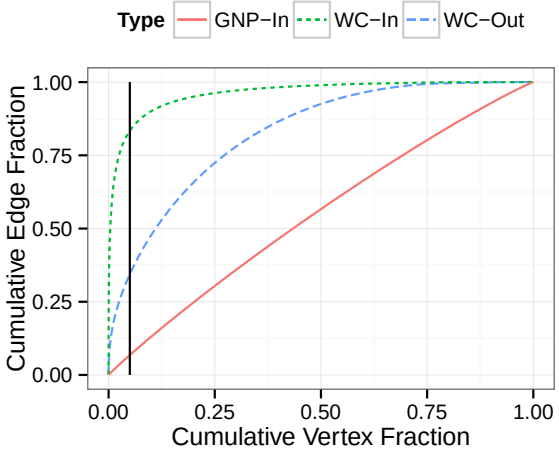

Figure 11.5. Cumulative vertex versus in and out edge fraction for the web crawl and random graph.

of hyperlinks, and so the out-degree is artificially bounded.

We note two additional observations about the web crawl that were not mentioned in the prior Muesel et al. paper. Firstly, there are about 120 million vertices in the crawl having no incoming or outgoing connections. Secondly, there is a large number of repeated edges. For some of the high degree vertices, only about 75% of their incident edges are unique. This knowledge is especially useful for trimming and preprocessing the graph in order to reduce storage requirements and improve execution times of algorithms that ignore repeated edges during execution (e.g. SCC, harmonic centrality). We believe both may be artifacts of the web crawl, especially the first, possibly due to the large number of seeds.

11.7.2 Centrality Measures

We also compute several centrality metrics as part of our analytics test suite. Centrality measures generally indicate the relative importance of a single vertex in graph. We look at the centrality measures of harmonic centrality, PageRank, as well as in-degree and out-degree.

Previously, the Web Data Commons team and Meusel et al. calculated centrality measures on the condensed host graph. In order to gain a different perspective on the highest vertices by centrality, we chose instead to calculate centrality measures on the full page graph. The top-scoring results for centrality values of out-degree, in-degree, PageRank, and harmonic centrality are given in Table 11.5. Note that

Out-degree	In-degree	PageRank*	Harmonic*
photoshare.ru/..	youtube.com	youtube.com	wordpress.org
dvderotik.com/..	wordpress.org	youtube.com/t/..	twitter.com
zoover.be/..	youtube.com/t/..	youtube.com/testtube	twitter.com/privacy
cran.r-project.org/..	youtube.com/..	youtube.com/t/..	twitter.com/about
cran.rakanu.com/..	youtube.com/t/..	youtube.com/t/..	twitter.com/tos
linkagogo.com/..	youtube.com/..	tumblr.com	twitter.com/account/..
cran.r-project.org/..	youtube.com/t/..	google.com/intl/en/..	twitter.com/account/..
fussballdaten.de/..	gmpg.org/xfn/11	wordpress.org	twitter.com/about/resources
fussballdaten.de/..	google.com	google.com/intl/..	twitter.com/login
fussballdaten.de/..	google.com/intl/..	google.com	twitter.com/about/contact

Table 11.5. The top 10 web pages according to different centrality indices (* Harmonic, PageRank centrality rankings are approximate).

we consider the PageRank values approximate, as we report the top scores after 20 iterations. The harmonic centrality values are also considered approximate, because we only calculated the values for the top 100 vertices by in-degree.

There are several conclusions that can be reached based on the results of Table 11.5. The foremost being that out-degree does not have any significance as a centrality measure in this crawl. Looking at the other three columns, we note consistent similarities in the sites that appear between our results, and those calculated on just the host level graph (e.g. `youtube`, `wordpress`, `google`, `gmpg`, `twitter`). However, we also note how pages belonging to the same host consistently appear together in the rankings. As we will show with our next analytic, a number of these vertices are found to additionally appear in similar dense communities. We observe this trend continuing well beyond just the top 10 scoring pages, though these additional results are omitted for brevity. Overall, it is apparent that using the full page-level graph can be quite noisy, indicating the necessity for preprocessing and aggregating the page-level graph. While considering only the host-level graph for analysis and extraction of useful information is one way to do this (e.g. Meusel et al.), other potential approaches require further study.

11.7.3 Community Structure

Exploring dense clusters or communities in networks has been the focus of a lot of recent research [47, 70, 72, 217–221]. Recently, the label propagation algorithm [145] has received considerable attention due to the fact that it gives high-quality and stable communities, is very scalable, and is also easy to implement and parallelize. Table 11.6 gives the largest communities obtained after running our label prop-

n_{in}	m_{in}	m_{cut}	Representative vertex
57	1600	30	www.youtube.com
55	46	440	www.google.com/intl/en/..
17	370	400	www.tumblr.com
13	383	226	www.amazon.com
9	515	84	creativecommons.org/..
7	176	426	wordpress.org/..
5	38	194	www.flickr.com/..
4	120	147	www.google.com
4	281	18	tripadvisor.com
1	19	30	gmpg.org/xfn/
112	2126	32	www.youtube.com
18	548	277	www.tumblr.com
9	516	84	creativecommons.org/..
8	186	85	wordpress.org/..
7	57	83	www.amazon.com
6	41	21	www.flickr.com/..
6	39	58	askville.amazon.com
4	133	142	www.google.com
4	280	18	tripadvisor.com
3	78	13	www.househunt.com

Table 11.6. The top 10 communities ordered by vertex count, as given by our clustering output. The top half shows the list after 10 iterations, and the bottom list is after 30 iterations.

agation algorithm for 10 (top) and 30 (bottom) iterations. We give the number of vertices in the community (n_{in}), the number of intra-community edges (m_{in}), and well as the number of cut edges (m_{out}). These results are produced from separate runs, but demonstrate the previously-observed stability of communities produced from label propagation [222], as we note high similarity between the two lists (this is especially apparent with the Tumblr, WordPress, and TripAdvisor communities). The biggest difference we observe when increasing the number of iterations of label propagation is that the communities become denser and the intra-community versus inter-community edge ratio increases. Additionally, it is possible that large communities end up merging. Notably, it appears the two largest communities from the 10 iteration run would have eventually combined in the subsequent iterations, possibly a result of the high number of outgoing edges from the

www.google.com/intl/en/.. community into the www.youtube.com community.

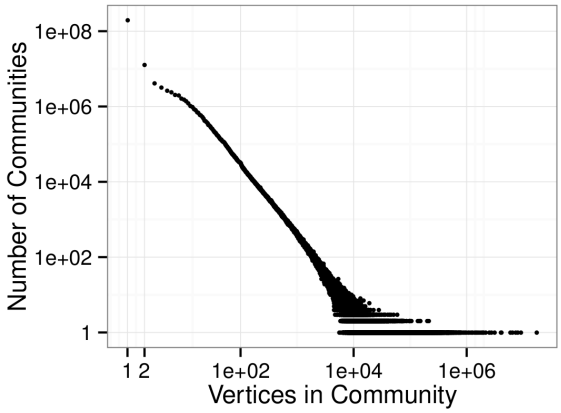

Figure 11.6. Frequency plot of community structure

We plot the frequency of community sizes produced after 30 iterations of label propagation on the web crawl in Figure 11.6. This plot has a striking similarity to the frequency plots of in-degree, out-degree, WCC, and SCC in Meusel et al. [216]. We note that this potentially gives additional credibility to the notion of an intrinsic heavy-tailed structure that has been observed previously in web crawls.

11.7.4 K-core Distribution

We finally use our approximate k-core implementation to explore the k-core decomposition of the web crawl. We capture the maximal approximate k-core sizes versus k-core values we retrieved during our analysis of the web crawl. These are plotted as cumulative fractions in Figure 11.7. We observe the sizes of the approximate k-cores are quite large, dropping below 500 million vertices only after reaching a k-core value of 128. However, by about a k-core value of 1024, only a small fraction of vertices remain (20 million, or about 0.5%). This demonstrates the existance of a very highly connected and dense central component to the web. These approximate k-core results can be refined, if necessary, to compute exact k-core values.

11.8 Conclusion

In this chapter, we focused on the implementation and optimization of graph analytics on HPC systems to process billion vertex scale graphs. We introduce our

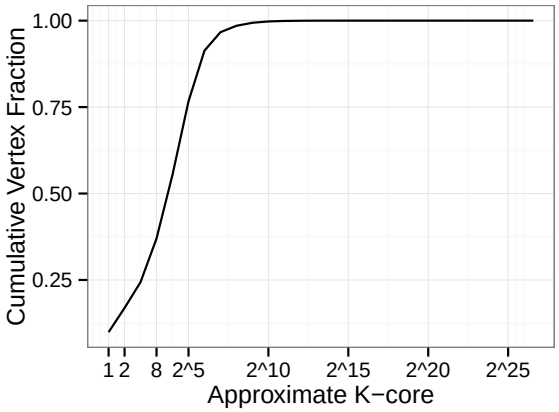

Figure 11.7. Cumulative fraction of vertices versus approximate k-core values.

approach in detail along with our implemented algorithms and describe how our approach might be extended to a broader class of graph analytics. We demonstrate that our code can both scale to over 65 K cores of a large scale supercomputer while also delivering high performance on a single node. Comparing to previous works, we show our code runs up to several orders of magnitude faster and is able to process larger graphs than other frameworks in the limited memory of a small cluster. Our code isn't needlessly complex, either. Each implemented analytics fits in only a couple hundred lines of C code. Using our implementation, we're able to run an end-to-end analysis of a 3.5 billion vertex and 129 billion edge web crawl in minutes, and provide novel insight into the structure of the web.

Chapter 12
Concluding Remarks

Graph-structured data is ubiquitous throughout all of the social and physical sciences. From protein interaction networks to electrical networks to human interaction and epidemiological networks, the ability to study and analyze graphs is challenging in its variance. With the explosive growth of the Internet and on-line communication possible between a majority of humans and their devices, the scale of real-world datasets makes utilizing modern high performance computing hardware a necessity for complex graph analysis. Overall, this thesis sought to find general purpose techniques for implementing a wide variety of complex analytics aimed at graphs from a wide variety of domains utilizing modern computational systems. This thesis further opens the door for considerable future work across all of the topics introduced and explored.

12.1 Summary of Contributions

This thesis introduced a new and highly optimized implementations of the color-coding approach for subgraph counting and enumeration, Fascia, as well as for minimum-weight path finding, FastPath. Utilizing work avoidance strategies, memory-reduction techniques, efficient parallelization, and a low-cost representation of the basic color-coding algorithm, Fascia and FastPath demonstrated orders-of-magnitude improvement versus prior work. Utilizing Fascia, tree-structured subgraphs were demonstrated as being a powerful analytic for cross-domain graph analysis. Additionally, the tree-structured subgraphs found by Fascia were also demonstrated to be a very useful feature for determining high quality network alignments with FastAlign.

This thesis also introduced new algorithms for various graph connectivity problems. Using knowledge from a study of prior decomposition techniques, the MULTISTEP approach for graph connectivity, weak connectivity, and strong connectivity was introduced. Through various shared-memory optimizations, such as multi-level queuing and other cache considerations, minimization of synchronization overheads, and subroutine optimizations such as utilizing a direction-optimizing breadth-first-search, the MULTISTEP algorithms offered at least a 2× average speedup compared to prior art. These various optimizations were leveraged while developing new algorithms for biconnectivity, which again offered considerable speedups when compared to the prior art. These optimizations were used along with new optimizations, such as loop collapse methods, to develop a general approach for implementing these and similarly-structured algorithms on manycore processors such as GPUs and Intel Xeon Phis.

As scalability for graph algorithms in a distributed environment is highly dependent on the methodology used for storing and partitioning a given input graph to the algorithm, another central topic of this thesis was graph layout and partitioning in distributed memory. This thesis introduced PuLP, a graph partitioner designed explicitly to account for the structural characteristics of real-world graphs commonly arising from human and natural interactions. PuLP is able to calculate partitions of a quality equal or better than other state-of-the-art approaches in a fraction of the time while using a fraction of memory. Combing PuLP with a novel ordering algorithm produced a method for distributed graph layout, DGL. While utilizing DGL on a wide variety of distributed graph computations, consistent reductions in execution times of up to an order-of-magnitude resulted when compared to naïve approaches.

A final aspect of this thesis considered an end-to-end approach for efficient distributed graph computations on graphs of very small to very large scales. This approach utilized a number of the shared and distributed memory optimization techniques as determined during prior research efforts. Compared to modern state-of-the-art graph processing frameworks, several orders-of-magnitude speedups were noted. Additionally, scalability to graphs with billions of vertices and hundreds of billions of edges was possible, which enabled the analysis of what is the current largest publicly available web crawl.

12.2 Future Directions

Future work will further extend FASCIA to even larger networks and for scaling to larger compute platforms. The all-to-all exchange step of FASCIA is currently a bottleneck in the distributed-memory implementation, and future work could investigate alternatives communication approaches. For FASTPATH, combing the general color-coding techniques introduced with FASCIA, along with optimizations of prior art to reduce per-iteration and total execution costs [65, 66, 101], would further reduce computational costs for minimum weight pathway search and allow scaling to larger networks and longer pathways. Future work with FASTALIGN might explore a more complex network alignment approach that utilizes network-specific information beyond degrees and subgraph counts. Additionally, combining both graphlet and treelet counts might prove promising, although the development of a similarity metric that combines subgraphs of varying sizes and structure might be challenging. Further work would also strive to analyze node and interaction correctness in addition to edge correctness, as these metrics are much more powerful in providing better insight into the actual effectiveness of the alignment approaches. Further quality and performance comparisons of sampling methods for subgraph counting vs. the color-coding approach and the use of graphlets vs. treelets would also make for interesting future work.

Future work with MULTISTEP might be a more in-depth study of performance optimizations for distributed processing of large-scale graph instances. The biconnectivity algorithms could benefit from a more theoretical analysis to identify potential worst-case instances, with mitigation efforts to minimize the performance effects. Focusing on more general performance tuning and experimental analysis of the biconnectivity algorithms would also be useful future work. Using biconnectivity decompositions and relative component sizes to characterize graph structures, particularly community structure in real-world social networks, will perhaps provide novel insight. With regards to manycore processing, future work might involve applying the developed methodology to a wide range of other graph analytics. Further, there are several research efforts on using both the host and the accelerator for graph analytic workloads [26, 156, 223], and this would correspondingly be another avenue for future work.

Promising future work with PULP would be to implement PULP in a dis-

tributed setting to partition graphs too large to fit in the main memory of a single compute node. The scalability of the label propagation technique makes this a promising approach for partitioning larger graph instances than have ever been demonstrated. Although not investigated with DGL, there are several other promising ordering methods that might be applied to accelerate the computational portions of distributed graph analysis [180, 200, 209, 210]. More generally, a study of the effects of a wider variety of ordering methods and partitioning objectives and constraints might lead to greater insight into how best to represent graphs in distributed memory space.

The work in a general approach for distributed graph analysis lays the foundation for future research in three different directions. An investigation into a performance portable compression method will allow us to run analytics in-memory with an even smaller footprint on a small number of nodes. A smarter partitioning methodology, while introducing considerably more complexity into the code, would allow us to scale up to even larger graphs on system-scale runs of hundreds of thousands of cores. This would be a definite use-case for a future distributed version of PuLP. Additionally, the techniques described could be applied to dozens of other graph analytics that follow the same general computational pattern, allowing more in-depth analyses of graph instances up to massive scales.

Bibliography

[1] Pržulj, N. (2007) "Biological Network Comparison Using Graphlet Degree Distribution," *Bioinformatics*, **23**(2), pp. e177–83.

[2] Conte, D., P. Foggia, C. Sansone, and M. Vento (2004) "Thirty years of graph pattern matching in pattern recognition," *International Journal of Pattern Recognitition and Artificial Intelligence*.

[3] Broecheler, M., A. Pugliese, and V. Subrahmanian (2010) "Cosi: Cloud oriented subgraph identification in massive social networks," in *International Conference on Advances in Social Networks Analysis and Mining*.

[4] Milo, R., S. Shen-Orr, S. Itzokovitz, N. Kashtan, D. Chklovskii, and U. Alon (2002) "Network motifs: simple building blocks of complex networks," *Science*.

[5] Chandola, V., A. Banerjee, and V. Kumar (2009) "Anomaly detection: a survey," *ACM comput. Surv.*

[6] Defense Advanced Research Projects Agency (2008) *DARPA Mathematical Challenges*, *Tech. Rep. BAA-07-68*, DARPA.

[7] Babcock, B., S. Babu, M. Datar, R. Motwani, and J. Widom (2002) "Models and issues in data stream systems," in *Proceedings of the 21st ACM SIGNMOD-SIGACT-SIGART symposium on Priciples of database systems*.

[8] et al., N. J. (1985) "On the complexity of the subgraph problem," *Common Math*.

[9] Zhao, Z., G. Wang, A. R. Butt, M. Khan, V. S. Kumar, and M. V. Marathe (2012) "SAHAD: Subgraph analysis in massive networks using Hadoop," in *IEEE 26th International Parallel and Distributed Processing Symposium*.

[10] Slota, G. M. and K. Madduri (2013) "Fast Approximate Subgraph Counting and Enumeration," in *2013 International Conference on Parallel Processing (ICPP13)*.

[11] Broder, A., R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener (2000) "Graph structure in the Web," *Computer Networks*, **33**, pp. 309–320.

[12] Mislove, A., M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee (2007) "Measurement and analysis of online social networks," in *Proc. 7th ACM SIGCOMM Conf. on Internet measurement (IMC '07)*, pp. 29–42.

[13] Pinheiro, C. A. R. (2011) *Social Network Analysis in Telecommunications*, SAS Institute Inc.

[14] Sariyüce, A. E., K. Kaya, E. Saule, and U. V. Catalyürek (2013) "Incremental Algorithms for Closeness Centrality," in *IEEE International Conference on BigData*.

[15] Orzan, S. (2004) *On Distributed Verification and Verified Distribution*, Ph.D. thesis, Vrije Universiteit.

[16] Pothen, A. and C.-J. Fan (1990) "Computing the block triangular form of a sparse matrix," *ACM Trans. on Mathematical Software (TOMS)*, **16**(4), pp. 303–324.

[17] Hopcroft, J. and R. Tarjan (1973) "Efficient Algorithms for Graph Manipulation," *CACM*, **16**(6), pp. 374–378.

[18] Tarjan, R. E. (1972) "Depth first search and linear graph algorithms," *SIAM Journal of Computing*, **1**, pp. 146–160.

[19] Fleischer, L. K., B. Hendrickson, and A. Pinar (2000) "On Identifying Strongly Connected Components in Parallel," in *Parallel and Distributed Processing*, vol. 1800 of *LNCS*, Springer Berlin Heidelberg, pp. 505–511.

[20] Tarjan, R. E. and U. Vishkin (1985) "An efficient parallel biconnectivity algorithm," *SIAM Journal on Computing*, **14**(4), pp. 862–874.

[21] Slota, G. M., S. Rajamanickam, and K. Madduri (2014) "BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems," in *Proc. Int'l. Parallel and Distributed Processing Symp. (IPDPS)*.

[22] Saule, E. and U. V. Çatalyürek (2012) "An early evaluation of the scalability of graph algorithms on the Intel MIC architecture," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, IEEE, pp. 1629–1639.

[23] Hong, S., S. K. Kim, T. Oguntebi, and K. Olukotun (2011) "Accelerating CUDA graph algorithms at maximum warp," *ACM SIGPLAN Notices*, **46**(8), pp. 267–276.

[24] Davidson, A. A., S. Baxter, M. Garland, and J. D. Owens (2014) "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths," in *International Parallel and Distributed Processing Symposium*, vol. 28.

[25] Merrill, D., M. Garland, and A. Grimshaw (2012) "Scalable GPU graph traversal," in *ACM SIGPLAN Notices*, vol. 47, ACM, pp. 117–128.

[26] Hong, S., T. Oguntebi, and K. Olukotun (2011) "Efficient parallel graph exploration on multi-core CPU and GPU," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, IEEE, pp. 78–88.

[27] Bader, D. A., H. Meyerhenke, P. Sanders, and D. Wagner (2013) "Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop," *Contemporary Mathematics*, **588**.

[28] Bader, D. A., H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner (2014) "Benchmarking for Graph Clustering and Partitioning," in *Encyclopedia of Social Network Analysis and Mining*, pp. 73–82. URL `http://dx.doi.org/10.1007/978-1-4614-6170-8_23`

[29] Karypis, G. and V. Kumar (1998) "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, **20**(1), pp. 359–392.

[30] Slota, G. M., K. Madduri, and S. Rajamanickam, "Distributed Graph Layout for Scalable Small-World Network Analysis," .

[31] Gonzalez, J. E., R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica (2014) "GraphX: Graph Processing in a Distributed Dataflow Framework," in *OSDI'14*.

[32] Low, Y., J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein (2014) "GraphLab: A New Framework For Parallel Machine Learning," *CoRR*, **abs/1408.2041**.

[33] Gonzalez, J. E., Y. Low, H. Gu, D. Bickson, and C. Guestrin (2012) "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *Proc. OSDI*.

[34] Chen, R., J. Shi, Y. Chen, and H. Chen (2015) "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proc. ECCS*, ACM, p. 1.

[35] Zheng, D., D. Mhembere, R. C. Burns, and A. S. Szalay (2015) "FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs," in *Proc. FAST*.

[36] Ching, A. and C. Kunz (2011) "Giraph: Large-scale graph processing infrastructure on Hadoop," *Hadoop Summit*, **6**(29), p. 2011.

[37] Malewicz, G., M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski (2010) "Pregel: a system for large-scale graph processing," in *Proc. SIGMOD*, ACM, pp. 135–146.

[38] Kang, U., C. E. Tsourakakis, and C. Faloutsos (2009) "PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations," in *Proc. ICDM*.

[39] Shao, B., H. Wang, and Y. Li (2013) "Trinity: A distributed graph engine on a memory cloud," in *Proc. SIGMOD*.

[40] McSherry, F., M. Isard, and D. G. Murray, "Scalability! But at what COST," .

[41] Ching, A., S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan (2015) "One trillion edges: graph processing at Facebook-scale," *Proceedings of the VLDB Endowment*, **8**(12), pp. 1804–1815.

[42] Slota, G. M. and K. Madduri (2014) "Complex Network Analysis using Parallel Approximate Motif Counting," in *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS14)*.

[43] ——— (2015) "Parallel Color-Coding," *Parallel Computing, Systems & Applications*, **47**, pp. 51–69.

[44] ——— (2014) "Simple Parallel Biconnectivity Algorithms for Multicore Platforms," in *IEEE International Conference on High Performance Computing*.

[45] Slota, G. M., S. Rajamanickam, and K. Madduri (2015) "High-performance Graph Analytics on Manycore Processors," in *International Parallel & Distributed Processing Symposium (IPDPS)*.

[46] Slota, G. M., K. Madduri, and S. Rajamanickam (under review), "Complex Network Partitioning Using Label Propagation," .

[47] ——— (2014) "PuLP: Scalable Multi-Objective Multi-Constraint Partitioning for Small-World Networks," in *IEEE International Conference on Big Data*.

[48] Slota, G. M., S. Rajamanickam, and K. Madduri (2016) "A Case Study of Complex Graph Analysis in Distributed Memory: Implementation and Optimization," in *To appear in the Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*.

[49] Alon, N., R. Yuster, and U. Zwick (1995) "Color-coding," *J. ACM*, **42**(4), pp. 844–856.

[50] Cook, S. A. (1971) "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, ACM, pp. 151–158.

[51] Pržulj, N., D. G. Corneil, and I. Jurisca (2004) "Modeling Interactome, Scale-Free or Geometric?" *Bioinformatics*, **20**(18), pp. 3508–3515.

[52] Alon, N., P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. Sahinalp (2008) "Biomolecular network motif counting and discovery by color coding," *Bioinformatics*, **24**(13), pp. i241–i249.

[53] Huan, J., W. Wang, and J. Prins (2003) "Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism," in *Proc. 3rd IEEE Int'l. Conf. on Data Mining (ICDM)*, p. 549.

[54] Kuramochi, M. and G. Karypis (2001) "Frequent subgraph discovery." in *In Proc. IEEE Int'l. Conf. on Data Mining (ICDM)*.

[55] Kashtan, N., S. Itzkovitz, R. Milo, and U. Alon (2004) "Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs," *Bioinformatics*, **20**(11), pp. 1746–1758.

[56] Wernicke, S. (2004) "Efficient Detection of Network Motifs," *IEEE/ACM Trans. on Computational Biology and Bioinformatics*, **3**(4), pp. 347–359.

[57] Rahman, M., M. Bhuiyan, M. Al Hasan, et al. (2014) "Graft: An efficient graphlet counting method for large graph analysis," *Knowledge and Data Engineering, IEEE Transactions on*, **26**(10), pp. 2466–2478.

[58] Chen, J., W. Hsu, M. L. Lee, and S.-K. Ng (2006) "NeMoFinder: dissecting genome-wide protein-protein interactions with meso-scale network motifs," in *Proc. 12th ACM SIGKDD Int'l. Conf. on Knowledge Discovery and Data mining (KDD)*, pp. 106–115.

[59] Pržulj, N., D. Corneil, and I. Jurisica (2006) "Efficient estimation of graphlet frequency distributions in protein-protein interaction networks," *Bioinformatics*, **22**(8), pp. 974–980.

[60] Milenković, T. and N. Pržulj (2008) "Uncovering Biological Network Function via Graphlet Degree Signatures," *Cancer Informatics*, **6**, pp. 257–273.

[61] Bordino, I., D. Donata, A. Gionis, and S. Leonardi (2008) "Mining large networks with subgraph counting," in *Proc. 8th IEEE Int'l. Conf. on Data Mining (ICDM)*, pp. 737–742.

[62] Zhao, Z., M. Khan, V. S. A. Kumar, and M. V. Marathe (2010) "Subgraph Enumeration in Large Social Contact Networks using Parallel Color Coding and Streaming," in *Proc. 39th Int'l. Conf. on Parallel Processing (ICPP)*, pp. 594–603.

[63] Gülsoy, G., B. Gandhi, and T. Kahveci (2012) "Topac: alignment of gene regulatory networks using topology-aware coloring," *Journal of bioinformatics and computational biology*, **10**(01).

[64] ——— (2011) "Topology aware coloring of gene regulatory networks," in *Proceedings of the 2nd ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, ACM, pp. 435–440.

[65] Gabr, H., A. Dobra, and T. Kahveci (2012) "From Uncertain Protein Interaction Networks to Signaling Pathways Through Intensive Color Coding," in *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, World Scientific, pp. 111–122.

[66] Hüffner, F., S. Wernicke, and T. Zichner (2008) "Algorithm engineering for color-coding with applications to signaling pathway detection," *Algorithmica*, **52**(2), pp. 114–132.

[67] Beckenbach, E. (1981) *Applied Combinatorial Mathematics*, Krieger Pub Co.

[68] Klimmt, B. and Y. Yang (2004) "Introducing the Enron corpus," in *Proc. 1st Conf. on Email and Anti-Spam (CEAS)*.

[69] "Stanford Large Network Dataset Collection," `http://snap.stanford.edu/data/index.html`, last accessed July 2014.

[70] Leskovec, J., K. Lang, A. Dasgupta, and M. Mahoney (2009) "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters," *Internet Mathematics*, **6**(1), pp. 29–123.

[71] Network Dynamics and Simulation and Science Laboratory (2006) *Synthetic Data Products for Societal Infrastructures and Proto-Populations: Data Set 1.0*, *Tech. Rep. NDSSL-TR-06-006*, Virginia Polytechnic Institute and State University.

[72] Yang, J. and J. Leskovec (2012) "Defining and Evaluating Network Communities based on Ground-truth," in *Proc. IEEE Int'l. Conf. on Data Mining (ICDM)*, pp. 745–754.

[73] Cha, M., H. Haddadi, F. Benevenuto, and K. P. Gummadi (2010) "Measuring User Influence in Twitter: The Million Follower Fallacy," in *Proc. Int'l. Conf. on Weblogs and Social Media (ICWSM)*.

[74] Boldi, P., B. Codenotti, M. Santini, and S. Vigna (2004) "UbiCrawler: A Scalable Fully Distributed Web Crawler," *Software: Practice & Experience*, **34**(8), pp. 711–726.

[75] Davis, T. A. and Y. Hu (2011) "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, **38**(1), pp. 1–25.

[76] Xenarios, I., L. Salwinski, X. J. Duan, P. Higney, S. M. Kim, and D. Eisenberg (2002) "DIP, the Database of Interacting Proteins: a research tool for studying cellular networks of protein interactions," *Nucleic Acids Research*, **30**(1), pp. 303–305.

[77] Radivojac, P., K. Page, W. T. Clark, B. J. Peters, A. Mohan, S. M. Boyle, and S. D. Mooney (2008) "An integrated approach to inferring gene-disaese assicoations in humans," *Proteins*.

[78] Omidi, S., F. Schreiber, and A. Masoudi-Nejad (2009) "MODA: an efficient algorithm for network motif discovery in biological networks," *Genes Genet Syst*, **84**(5), pp. 385–395.

[79] Leskovec, J., A. Singh, and J. Kleinberg (2006) "Patterns of influence in a recommendation network," *Proc. 10th Pacific-Asia Conf. on Advances in Knowledge Discovery and Data Mining (PAKDD)*, pp. 380–389.

[80] Kunegis, J., "KONECT - the Koblenz Network Collection," `konect.uni-koblenz.de`, last accessed July 2014.

[81] Leskovec, J., J. Kleinberg, and C. Faloutsos (2007) "Graph Evolution: Densification and Shrinking Diameters," *ACM Trans. on Knowledge Discovery from Data*, **1**(1).

[82] Bimal, V., A. Mislove, M. Cha, and K. Gummadi (2009) "On the evolution of user interaction in Facebook," in *Proc. 2nd ACM Workshop on Online Social Networks (WOSN)*, pp. 37–42.

[83] Ripeanu, M., A. Iamnitchi, and I. Foster (2002) "Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design," *IEEE Internet Computing*, **6**(1), pp. 50–57.

[84] Csardi, G. and T. Nepusz (2006) "The igraph software package for complex network research," *InterJournal*, **Complex Systems**, p. 1695.

[85] McAuley, J. and J. Leskovec (2012) "Learning to Discover Social Circles in Ego Networks," in *Proc. 26th Annual Conf. on Neural Inf. Proc. Systems (NIPS)*, pp. 548–556.

[86] Richardson, M., R. Agrawal, and P. Domingos (2003) "Trust Management for the Semantic Web," in *Proc. 2nd Int'l. Semantic Web Conf. (ISWC)*, pp. 351–368.

[87] Chakrabarti, D., Y. Zhan, and C. Faloutsos (2004) "R-MAT: A Recursive Model for Graph Mining," in *Proc. Int'l. Conf. on Data Mining (SDM)*.

[88] Ruskey, F., "The (Combinatorial) Object Server," `http://theory.cs.uvic.ca/root.html`, last accessed Feb 2014.

[89] Kuchaiev, O., T. Milenković, V. Memisević, W. Hayes, and N. Pržulj (2010) "Topological network alignment uncovers biological function and phylogeny," *Journal of the Royal Society Interface*.

[90] ——— (2010) *Supplementary Information for: Topological network alignment uncovers biological function and phylogeny*, *Tech. rep.*, University of California, Irvine.

[91] Milenković, T., W. L. Ng, W. Hayes, and N. Pržulj (2010) "Optimal Network Alignment with Graphlet Degree Vectors," *Cancer Informatics*.

[92] Kuchaiev, O. and N. Pržulj (2011) "Integrative Network Alignment Reveals Large Regions of Global Network Similarity in Yeast and Human," *Bioinformatics*.

[93] Memisević, V. and N. Pržulj (2012) "C-GRAAL: common-neighbors-based global GRAph ALignment of biological networks," *Integrative Biology*.

[94] Dost, B., T. Shlomi, N. Gupta, E. Ruppin, V. Bafna, and R. Sharan (2007) "QNet: a tool for querying protein interaction networks," in *Research in Computational Molecular Biology*, Springer, pp. 1–15.

[95] Shlomi, T., D. Segal, E. Ruppin, and R. Sharan (2006) "QPath: a method for querying pathways in a protein-protein interaction network," *BMC bioinformatics*, **7**(1), p. 199.

[96] Http://bio-nets.doc.ic.ac.uk/GRAAL_suppl_inf/.

[97] Collins, S. R., P. Kemmeren, X.-C. Zhao, J. F. Greenblatt, F. Spencer, F. C. Holstege, J. S. Weissman, and N. J. Krogan (2007) "Toward a comprehensive atlas of the physical interactome of Saccharomyces cerevisiae," *Molecular & Cellular Proteomics*, **6**(3), pp. 439–450.

[98] Dyer, M. D., C. Neff, M. Dufford, C. G. Rivera, D. Shattuck, J. Bassaganya-Riera, T. M. Murali, and B. W. Sobral (2010) *PLoS One*.

[99] Shimoda, Y., S. Shinpo, M. Lphara, Y. Nakamura, S. Tabata, and S. Sato (2008) *DNA Res.*

[100] Http://bio-nets.doc.ic.ac.uk/C-GRAAL/.

[101] Scott, J., T. Ideker, R. M. Karp, and R. Sharan (2006) "Efficient algorithms for detecting signaling pathways in protein interaction networks," *Journal of Computational Biology*, **13**(2), pp. 133–144.

[102] Kelley, B. P., R. Sharan, R. M. Karp, T. Sittler, D. E. Root, B. R. Stockwell, and T. Ideker (2003) "Conserved pathways within bacteria and yeast as revealed by global protein network alignment," *Proceedings of the National Academy of Sciences*, **100**(20), pp. 11394–11399.

[103] Steffen, M., A. Petti, J. Aach, P. D'haeseleer, and G. Church (2002) "Automated modelling of signal transduction networks," *BMC bioinformatics*, **3**(1), p. 34.

[104] Chatr-Aryamontri, A., A. Ceol, L. M. Palazzi, G. Nardelli, M. V. Schneider, L. Castagnoli, and G. Cesareni (2007) "MINT: the Molecular INTeraction database," *Nucleic acids research*, **35**(suppl 1), pp. D572–D574.

[105] Hüffner, F., S. Wernicke, and T. Zichner (2007) "FASPAD: fast signaling pathway detection," *Bioinformatics*, **23**(13), pp. 1708–1709.

[106] Huang, D. W., B. T. Sherman, and R. A. Lempicki (2009) "Systematic and integrative analysis of large gene lists using DAVID bioinformatics resources," *Nature Protocols*, **4**(1), pp. 44–57.

[107] ——— (2009) "Bioinformatics enrichment tools: paths toward the comprehensive functional analysis of large gene lists," *Nucleic Acids Research*, **37**(1), pp. 1–13.

[108] Deininger, M. W. N., J. M. Goldman, and J. V. Melo (2000) "The molecular biology of chronic myeloid leukemia," *Blood*, **96**(10), pp. 3343–3356.

[109] Xie, A. and P. Beerel (2000) "Implicit enumeration of strongly connected components and an application to formal verification," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, **19**(10), pp. 1225–1230.

[110] Edwards, J. A. and U. Vishkin (2012) "Better speedups using simpler parallel programming for graph connectivity and biconnectivity," in *Proc. 2012 Int'l. Workshop on Programming Models and Applications for Multicores and Manycores*, ACM, pp. 103–114.

[111] Agarwal, V., F. Petrini, D. Pasetto, and D. Bader (2010) "Scalable Graph Exploration on Multicore Processors," in *Proc. Supercomputing*.

[112] Beamer, S., K. Asanović, and D. Patterson (2012) "Direction-Optimizing Breadth-First Search," in *Proc. Supercomputing (SC)*.

[113] Chhugani, J., N. Satish, C. Kim, J. Sewall, and P. Dubey (2012) "Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency," in *Proc. Supercomputing*.

[114] Hong, S., N. C. Rodia, and K. Olukotun (2013) "On fast parallel detection of strongly connected components (SCC) in small-world graphs," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, p. 92.

[115] Shun, J. and G. Blelloch (2013) "Ligra: A Lightweight Graph processing Framework for Shared Memory," in *Proc. 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 135–146.

[116] Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1983) *Data Structures and Algorithms*, Addison-Wesley.

[117] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger (2005) "Finding strongly connected components in distributed graphs," *Journal of Parallel and Distributed Computing*, **65**(8), pp. 901–910.

[118] Barnat, J. and P. Moravec (2006) "Parallel Algorithms for Finding SCCs in Implicitly Given Graphs," *Formal Methods: Applications and Technology*, **4346**, pp. 316–330.

[119] Plimpton, S. J. and K. D. Devine (2011) "MapReduce in MPI for Large-scale graph algorithms," *Parallel Comput.*, **37**(9), pp. 610–632.

[120] Intel (2011) *Intel 64 and IA-32 Architectures Software Developer's Manual, System Programming Guide, Part 1*, vol. 3A, Intel Press.

[121] Mehlhorn, K., S. Näher, and P. Sanders (2007), "Engineering DFS-Based Graph Algorithms," .

[122] Madduri, K. and D. A. Bader, "GTgraph: A suite of synthetic graph generators," `http://www.cse.psu.edu/~madduri/software/GTgraph/`, last accessed Aug 25, 2014.

[123] Leskovec, J., D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani (2010) "Kronecker Graphs: An Approach to Modeling Networks," *Journal of Machine Learning Research*, **11**, pp. 985–1042.

[124] "Graph 500," `http://www.graph500.org`, last accessed Aug 25, 2014.

[125] Cong, G. and D. A. Bader (2005) "An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs)," in *Proc. Int'l. Parallel and Distributed Processing Symp. (IPDPS)*.

[126] Nagamochi, H. and T. Ibaraki (1992) "A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph," *Algorithmica*, **7**(1-6), pp. 583–596.

[127] Cong, G. and D. A. Bader (2004), "TV-Filter Biconnected components implementation," `http://www.cc.gatech.edu/~bader/code.html`, last accessed Aug 25, 2014.

[128] Bader, D. A. and J. JáJá (1999) "SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs)," *Journal of Parallel and Distributed Computing*, **58**(1), pp. 92–108.

[129] Eckstein, D. M. (1979) *BFS and biconnectivity*, *Tech. Rep. 79-11*, Dept. of Computer Science, Iowa State University of Science and Technology.

[130] Savage, C. and J. JáJá (1981) "Fast, efficient parallel algorithms for some graph problems," *SIAM J. Computing*, **10**(4), pp. 682–691.

[131] Tsin, Y. H. and F. Y. Chin (1984) "Efficient Parallel Algorithms for a Class of Graph Theoretic Problems," *SIAM J. Computing*, **31**(2), pp. 245–281.

[132] Maon, Y., B. Schieber, and U. Vishkin (1986) "Parallel ear decomposition search (EDS) and st-numbering in graphs," *Theoretical Computer Science*, **47**, pp. 277–298.

[133] Miller, G. L. and V. Ramachandran (1992) "A new graph triconnectivity algorithm and its parallelization," *Combinatorica*, **12**(1), pp. 53–76.

[134] Hsu, T.-S., V. Ramachandran, and N. Dean (1994) "Implementation of parallel graph algorithms on the MasPar," in *Third DIMACS Implementation Challenge: Parallel Algorithms* (N. Dean and G. E. Shannon, eds.), vol. 15 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, AMS, pp. 165–198.

[135] Ausiello, G., D. Firmani, L. Laura, and E. Paracone (2012) "Large-Scale Graph Biconnectivity in MapReduce," *Department of Computer and System Sciences Antonio Ruberti Technical Reports*, **4**(4).

[136] Ausiello, G., D. Firmani, et al. (2012) "Real-time monitoring of undirected networks: Articulation points, bridges, and connected and biconnected components," *Networks*, **59**(3), pp. 275–288.

[137] Westbrook, J. and R. E. Tarjan (1992) "Maintaining bridge-connected and biconnected components on-line," *Algorithmica*, **7**(1-6), pp. 433–464.

[138] Wikimedia Foundation (2013), "Wikipedia links, English network dataset – KONECT," .
URL `http://konect.uni-koblenz.de/networks/wikipedia_link_en`

[139] Janna, C., M. Ferronato, and G. Gambolati (2012) "Parallel inexact constraint preconditioning for ill-conditioned consolidation problems," *Computational Geosciences*, **16**(3), pp. 661–675.

[140] Burtscher, M., R. Nasre, and K. Pingali (2012) "A Quantitative Study of Irregular Programs on GPUs," in *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)*.

[141] Davidson, A., S. Baxter, M. Garland, and J. D. Owens (2014) "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*.

[142] Banerjee, D. S., S. Sharma, and K. Kothapalli (2013) "Work efficient parallel algorithms for large graph exploration," in *Proc. Int'l. Conf. on High Performance Computing (HiPC)*.

[143] Zhong, J. and B. He (2014) "Medusa: Simplified Graph Processing on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, **25**(6), pp. 1543–1552.

[144] Chakaravarthy, V. T., F. Checconi, F. Petrini, and Y. Sabharwal (2014) "Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*.

[145] Raghavan, U. N., R. Albert, and S. Kumara (2007) "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, **76**(3), p. 036106.

[146] McLaughlin, A. and D. A. Bader (2014) "Scalable and High Performance Betweenness Centrality on the GPU," in *Proc. Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*.

[147] Dal, G. H., W. A. Kosters, and F. W. Takes (2014) "Fast diameter computation of large sparse graphs using GPUs," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, IEEE, pp. 632–639.

[148] Montresor, A., F. D. Pellegrini, and D. Miorandi (2011), "Distributed k-Core Decomposition," ArXiv:1103.5320.

[149] Gregor, D. and A. Lumsdaine (2005) "Lifting sequential graph algorithms for distributed-memory parallel computation," in *Proc. ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*.

[150] Berry, J. W., B. Hendrickson, S. Kahan, and P. Konecny (2007) "Software and Algorithms for Graph Queries on Multithreaded Architectures," in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP)*.

[151] Kulkarni, M., K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew (2007) "Optimistic parallelism requires abstractions," in *Proc. ACM SIGPLAN Conf. on Programming language design and implementation (PLDI)*.

[152] Staudt, C. L., A. Sazonovs, and H. Meyerhenke (2014), "NetworKit: An Interactive Tool Suite for High-Performance Network Analysis," ArXiv:1403.3005.

[153] Carter Edwards, H., C. R. Trott, and D. Sunderland (2014) "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*.

[154] Bacon, D. F., S. L. Graham, and O. J. Sharp (1994) "Compiler transformations for high-performance computing," *ACM Computing Surveys*, **26**(4), pp. 345–420.

[155] (2009) *Optimizing Loop-Level Parallelism in Cray XMT Applications*, *Tech. rep.*, Cray Inc.

[156] Gao, T., Y. Lu, B. Zhang, and G. Suo (2014) "Using the Intel Many Integrated Core to accelerate graph traversal," *International Journal of High Performance Computing Applications*, p. 1094342014524240.

[157] Checconi, F. and F. Petrini (2014) "Traversing Trillions of Edges in Real-time: Graph Exploration on Large-scale Parallel Machines," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*.

[158] Nguyen, D., A. Lenharth, and K. Pingali (2013) "A Lightweight Infrastructure for Graph Analytics," in *Proc. ACM Symp. on Operating Systems Principles (SOSP)*.

[159] Quick, L., P. Wilkinson, and D. Hardcastle (2012) "Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis," in *Proc. Int'l. Conf. on Advances in Social Networks Analysis and Mining (ASONAM)*.

[160] Guo, Y., M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke (2013) "Towards Benchmarking Graph-processing platforms," in *Proc. Supercomputing (SC), poster*.

[161] Wang, L., Y. Xiao, B. Shao, and H. Wang (2014) "How to partition a billion-node graph," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, IEEE, pp. 568–579.

[162] Meyerhenke, H., P. Sanders, and C. Schulz (2014) "Partitioning Complex Networks via Size-Constrained Clustering," in *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, pp. 351–363.
URL `http://dx.doi.org/10.1007/978-3-319-07959-2_30`

[163] ——— (2015) "Parallel Graph Partitioning for Complex Networks," in *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pp. 1055–1064.
URL `http://dx.doi.org/10.1109/IPDPS.2015.18`

[164] Ugander, J. and L. Backstrom (2013) "Balanced Label Propagation for Partitioning Massive Graphs," in *Proc. Web Search and Data Mining (WSDM)*.

[165] Vaquero, L., F. Cuadrado, D. Logothetis, and C. Martella (2013) "xDGP: A Dynamic Graph Processing System with Adaptive Partitioning," *CoRR*, **abs/1309.1049**.

[166] Martella, C., D. Logothetis, A. Loukas, and G. Siganos (2014) "Spinner: Scalable Graph Partitioning in the Cloud," *arXiv preprint arXiv:1404.3861*.

[167] Karypis, G. and V. Kumar, "MeTis: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. Version 5.1.0," `http://glaros.dtc.umn.edu/gkhome/metis/metis/download`, last accessed July 2014.

[168] ——— (1996) "Parallel Multilevel K-way Partitioning Scheme for Irregular Graphs," in *Proc. ACM/IEEE Conference on Supercomputing (SC)*.

[169] ——— (1998) "Multilevel Algorithms for Multi-constraint Graph Partitioning," in *Proc. ACM/IEEE Conference on Supercomputing (SC)*.

[170] Schloegel, K., G. Karypis, and V. Kumar (2000) "Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning," in *Proc. Euro-Par 2000 Parallel Processing*.

[171] Sanders, P. and C. Schulz (2013) "Think Locally, Act Globally: Highly Balanced Graph Partitioning," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, vol. 7933 of *LNCS*, Springer, pp. 164–175.

[172] Boman, E. G., K. D. Devine, and S. Rajamanickam (2013) "Scalable matrix computations on large scale-free graphs using 2D graph partitioning," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, p. 50.

[173] Pearce, R., M. Gokhale, and N. M. Amato (2013) "Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*.

[174] Pinar, A. and B. Hendrickson (2001) "Partitioning for Complex Objectives," in *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, IEEE Computer Society, Washington, DC, USA, pp. 121–.

[175] Uçar, B. and C. Aykanat (2004) "Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies," *SIAM Journal on Scientific Computing*, **25**(6), pp. 1837–1859.

[176] Catalyürek, Ü. V., M. Deveci, K. Kaya, and B. Uçar (2013) "UMPa: A Multi-objective, multi-level partitioner for communication minimization," *Contemporary Mathematics*, **588**.

[177] Fiduccia, C. M. and R. M. Mattheyses (1982) "A linear-time heuristic for improving network partitions," in *Proc. Conf. on Design Automation*.

[178] George, A. and J. W. Liu (1981) *Computer solution of large sparse positive definite systems*, Prentice-Hall.

[179] Boldi, P. and S. Vigna (2004) "The WebGraph Framework I: Compression Techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, ACM Press, Manhattan, USA, pp. 595–601.

[180] Boldi, P., M. Rosa, M. Santini, and S. Vigna (2011) "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks," in *Proceedings of the 20th international conference on World Wide Web*, ACM Press.

[181] Cong, G. and K. Makarychev (2012) "Optimizing Large-scale Graph Analysis on Multithreaded, Multicore Platforms," *Parallel and Distributed Processing Symposium, International*, **0**, pp. 414–425.

[182] Frasca, M., K. Madduri, and P. Raghavan (2012) "NUMA-aware graph mining techniques for performance and energy efficiency," in *Proc. Supercomputing (SC)*.

[183] Cuthill, E. and J. McKee (1969) "Reducing the Bandwidth of Sparse Symmetric Matrices," in *Proc. 1969 24th Nat'l. Conf.*, ACM '69, ACM, New York, NY, USA, pp. 157–172.

[184] Amestoy, P. R., T. A. Davis, and I. S. Duff (2004) "Algorithm 837: AMD, an Approximate Minimum Degree Ordering Algorithm," *ACM Trans. Math. Softw.*, **30**(3), pp. 381–388.

[185] Neumann, T. and G. Weikum (2010) "The RDF-3X engine for scalable management of RDF data," *VLDB J.*, **19**(1), pp. 91–113.

[186] Karantasis, K. I., A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali (2014) "Parallelization of Reordering Algorithms for Bandwidth and Wavefront Reduction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, IEEE Press, pp. 921–932.

[187] Eisenbrand, F. and F. Grandoni (2004) "On the complexity of fixed parameter clique and dominating set," *Theoretical Computer Science*, **326**(1–3), pp. 57–67.

[188] Buluç, A. and K. Madduri (2011) "Parallel breadth-first search on distributed memory systems," in *Proc. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*.

[189] ——— (2013) "Graph Partitioning for Scalable Distributed Graph Computations," in *Graph Partitioning and Graph Clustering* (D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, eds.), chap. 6, AMS, pp. 81–100.

[190] Panitanarak, T. and K. Madduri (2014) "Performance Analysis of Single-source Shortest Path Algorithms on Distributed-memory Systems," in *Proc. SIAM Workshop on Combinatorial Scientific Computing*.

[191] Meyer, U. and P. Sanders (2003) "Δ-stepping: a parallelizable shortest path algorithm," *J. Algs.*, **49**(1), pp. 114–152.

[192] (2004), "RDF Primer, W3C Recommendation." `http://www.w3.org/TR/rdf-primer`.

[193] Cudré-Mauroux, P., I. Enchev, S. Fundatureanu, P. T. Groth, A. Haque, A. Harth, F. L. Keppmann, D. P. Miranker, J. Sequeda, and M. Wylot (2013) "NoSQL Databases for RDF: An Empirical Evaluation," in *International Semantic Web Conference (2)* (H. Alani, L. Kagal, A. Fokoue, P. T. Groth, C. Biemann, J. X. Parreira, L. Aroyo, N. F. Noy, C. Welty, and K. Janowicz, eds.), vol. 8219 of *Lecture Notes in Computer Science*, Springer, pp. 310–325.

[194] Sakr, S., A. Liu, and A. G. Fayoumi (2013) "The family of mapreduce and large-scale data processing systems," *ACM Comput. Surv.*, **46**(1), p. 11.

[195] Chirravuri, S. K. (2014) *RDF3X-MPI: A Partitioning RDF Engine for Data-parallel SPARQL Querying*, Master's thesis, The Pennsylvania State University.

[196] Huang, J., D. J. Abadi, and K. Ren (2011) "Scalable SPARQL Querying of Large RDF Graphs," *PVLDB*, **4**(11), pp. 1123–1134.

[197] Bizer, C. and A. Schultz (2009) "The Berlin SPARQL Benchmark," *Int. J. Semantic Web Inf. Syst.*, **5**(2), pp. 1–24.

[198] Guo, Y., Z. Pan, and J. Heflin (2005) "LUBM: A benchmark for OWL knowledge base systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, **3**(2-3), pp. 158–182.

[199] Morsey, M., J. Lehmann, S. Auer, and A.-C. N. Ngomo (2011) "DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data," in *Proc. Int'l. Semantic Web Conf. (ISWC)*.

[200] Chierichetti, F., R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan (2009) "On compressing social networks," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, pp. 219–228.

[201] Hendrickson, B. and R. W. Leland (1995) "A Multi-Level Algorithm For Partitioning Graphs." in *Supercomputng*.

[202] Deveci, M., S. Rajamanickam, K. Devine, and Ü. Çatalyürek (2014) "Multi-jagged: A scalable parallel spatial partitioning algorithm," *IEEE Transactions on Parallel and Distributed Systems (In revision)*.

[203] Boman, E. G., K. D. Devine, V. J. Leung, S. Rajamanickam, L. A. Riesen, M. Deveci, and U. Catalyurek (2012) *Zoltan2: Next-Generation Combinatorial Toolkit*, *Tech. rep.*, Sandia National Laboratories.

[204] Catalyürek, U. V. and C. Aykanat (1999) "PaToH: a multilevel hypergraph partitioning tool, version 3.0," *Bilkent University, Department of Computer Engineering, Ankara*, **6533**.

[205] Deveci, M., K. Kaya, B. Uçar, and U. V. Catalyurek (2015) "Fast and high quality topology-aware task mapping," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, IEEE, pp. 197–206.

[206] Singh, J. P., C. Holt, T. Totsuka, A. Gupta, and J. Hennessy (1995) "Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity," *Journal of Parallel and Distributed Computing*, **27**(2), pp. 118–141.

[207] Ou, C.-W., M. Gunwani, and S. Ranka (1995) "Architecture-independent locality-improving transformations of computational graphs embedded in k-dimensions," in *Proceedings of the 9th international conference on Supercomputing*, ACM, pp. 289–298.

[208] Strout, M. M. and P. D. Hovland (2004) "Metrics and models for reordering transformations," in *Proceedings of the 2004 workshop on Memory system performance*, ACM, pp. 23–34.

[209] Mueller, C., B. Martin, and A. Lumsdaine (2007) "A comparison of vertex ordering algorithms for large graph visualization," in *Visualization, 2007. APVIS'07. 2007 6th International Asia-Pacific Symposium on*, IEEE, pp. 141–148.

[210] Safro, I. and B. Temkin (2011) "Multiscale approach for the network compression-friendly ordering," *Journal of Discrete Algorithms*, **9**(2), pp. 190–202.

[211] Pingali, K., D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. (2011) "The tao of parallelism in algorithms," *ACM Sigplan Notices*, **46**(6), pp. 12–25.

[212] Ediger, D., R. McColl, J. Riedy, and D. A. Bader (2012) "Stinger: High performance data structure for streaming graphs," in *Proc. HPEC*.

[213] Rochat, Y. (2009) "Closeness centrality extended to unconnected graphs: The harmonic centrality index," in *ASNA*, EPFL-CONF-200525.

[214] Meusel, R., S. Vigna, O. Lehmberg, and C. Bizer (2015) "The Graph Structure in the Web - Analyzed on Different Aggregation Levels," *J. Web Sci.*, **1**(1), pp. 33–47.

[215] Satish, N., N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey (2014) "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. SIGMOD*.

[216] Meusel, R., S. Vigna, O. Lehmberg, and C. Bizer (2014) "Graph Structure in the Web - Revisited: A Trick of the Heavy Tail," in *Proc. WWW*.

[217] Gleich, D. F. and C. Seshadhri (2012) "Vertex neighborhoods, low conductance cuts, and good seeds for local community methods," in *KDD*.

[218] Que, X., F. Checconi, F. Petrini, T. Wang, and W. Yu (2013) *Lightning-fast Community Detection in Social Media: A Scalable Implementation of the Louvain Algorithm*, *Tech. Rep. AU-CSSE-P ASL/13-TR01*, Auburn University.

[219] Girvan, M. and M. E. Newman (2002) "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, **99**(12), pp. 7821–7826.

[220] Lu, H., M. Halappanavar, A. Kalyanaraman, and S. Choudhury (2014) "Parallel heuristics for scalable community detection," in *IEEE IPDPSW14*.

[221] Fortunato, S. (2010) "Community detection in graphs," *Physics Reports*, **486**(3), pp. 75–174.

[222] Mandala, S., S. Kumara, and T. Yao (2012) "Detecting alternative graph clusterings," *Physical Review E*, **86**(1), p. 016111.

[223] Gharaibeh, A., L. B. Costa, E. Santos-Neto, and M. Ripeanu (2013) "On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*.

Vita

George M. Slota

George grew up in Pittsburgh, Pennsylvania and received a B.S. with honors in 2009 from Penn State, where he studied computer engineering, math, physics, and materials science. After graduation, George spent two years working for the Navy at the Naval Undersea Warfare Center in Newport, RI. In 2012, George returned to Penn State to complete his PhD. Under advisement of Kamesh Madduri, George has performed research into parallel graph algorithms on HPC systems, with an emphasis on social and biological graph mining applications. In addition, George has been working at Sandia National Labs since 2013, where he has been developing scalable graph and matrix algorithms in support of scientific computing applications. George was supported during the 2014-2015 academic year by a Blue Waters Graduate Fellowship. His research efforts have been awarded *Best Paper* at the 2013 International Conference on Parallel Processing and a *Graduate Research Assistant Award* from Penn State. The dissertation research contained in this thesis was also selected for presentation at the doctoral colloquia of the 2015 International Parallel and Distributed Processing Symposium and the 2015 The International Conference for High Performance Computing, Networking, Storage and Analysis.