PARALLEL ALGORITHMS FOR LARGE-SCALE GRAPH CLUSTERING ON DISTRIBUTED MEMORY ARCHITECTURES

By
INNA RYTSAREVA

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

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

AUGUST 2014
To the Faculty of Washington State University:

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

________________________________________
Ananth Kalyanaraman, Ph.D., Chair

________________________________________
Shira L. Broschat, Ph.D.

________________________________________
Jitesh Panchal, Ph.D.
Chair: Ananth Kalyanaraman

Graph algorithms on parallel architectures present an interesting case study for irregular applications. We address one such irregular application — one of clustering real world graphs constructed out of biological data and open-source communities data using parallel computers. While theoretical formulations of the clustering operation are either intractable or computationally prohibitive, efficient heuristics exist to tackle the problem in practice. Yet, implementing these heuristics under a parallel setting becomes a significant challenge owing to a combination of factors including: irregular data access and movement patterns, dependence of computational workload on the input, and a general need to maintain auxiliary pointer-based data structures. We present the design and evaluation of several parallel implementations of a popular serial graph clustering heuristic called the Shingling heuristic, which was originally developed by Gibson et al. We also present extensions of the standard version of the heuristic to enable the handling of edge weights in graphs and to improve the overall sensitivity of the output clustering. Our MapReduce implementation targets distributed memory clusters running Hadoop and MPI. Operating on an input graph that can be represented as a list of edges or adjacency list, our algorithm uses a combination of shuffling and sorting operations, and
pipelined MapReduce stages to implement the various phases of the algorithm. As a concrete case for application, we apply the methods developed on large-scale biological graphs obtained from a metagenomic community. Experimental results show both qualitative and performance improvements over previous executions of a baseline version of the clustering method. We also compare our results against other popular generic tools designed for community detection. As another applied case study of our research, we design and evaluate a cluster-based approach for socio–technical coordination in open-source community networks. The research experience in both these domains serve to demonstrate the high utility of cluster–based approaches in scientific domains.
Dedicated to my family
Acknowledgements

Foremost, I would like to thank my advisor, Dr. Ananth Kalyanaraman for his support, enthusiasm and patience throughout my Ph.D. years. He is a brilliant, extraordinary, and inspiring professor, mentor and advisor. I especially grateful to Dr. Kalyanaraman for introducing me to the fascinating field of High Performance Computing that quickly became my passion. I’m very thankful to him for sharing his knowledge and expertise in this area, and for giving me the right amount of freedom and guidance. He always made himself available to discuss any questions and I appreciate his constant encouragement and willingness to help during my graduate studies. He challenged me to be a better scientist and better person and I feel very fortunate to have an opportunity to be a part of his research group.

I would also like to thank my committee members, Dr. Shira Broschat and Dr. Jitesh Panchal, for their time, positive and constructive feedback, and knowledge that they shared with me that helped me considerably improve this dissertation. Dr. Broschat continuously encouraged me in my research and I was always excited about discussing my progress with her. Her expertise in the field of Bioinformatics also tremendously helped me with understanding some of the biological concepts. Our collaboration with Dr. Panchal became an important part of my dissertation. His scientific advice and thoughtful comments allowed me to explore new scientific areas and add significant ideas to my project.

I would like to thank my family for their life-long support; my parents, Svetlana and Viktor, for encouraging me in all of my pursuits and inspiring me to follow my dreams, sister, Yuliya, for her emotional support throughout all the ups and downs on
this journey, my grandmother, Amaliya, for always believing in me and encouraging me
to start this Ph.D., and my husband, David, for his love, understanding, patience and
great help in editing this manuscript.

A very special thanks goes out to my fellow labmates: Andy Wu, Benjamin Latt,
Hao Lu, Meenakshi Rameshkumar, Jeff Daily, Xiyo Xie, George Gruber, Tristan Mullis
and Paola Pesantez. I sincerely hope I continue to have opportunities to interact with
all of you for the rest of my research career.

I also would like to acknowledge the funding agencies that have made my thesis
possible. This research was supported by DOE award DE-SC-0006516 and NSF grant
IIS 0916463.
# List of Figures

1. Illustration of the different steps in the Shingling heuristic. Typically, $s_1 = s_2 = s$. ................................................................. 16

2. Cases illustrating the decisions to be made on the clustering of low degree vertices. ................................................................. 21

3. Illustration of our MapReduce algorithm for the Shingling heuristic. The figure shows the detailed Map-Reduce stages for Shingling Phases I and II. The function $f()$ (inside Map()) takes as argument a vertex $v$ in the form of its integer id and generates $c$ forms of $v$, viz. \{$v^1, v^2, \ldots v^c$\}; where $v^j = (A_j \times v + B_j) \% P$, $j \in [1 \ldots c]$ corresponds to the $j^{th}$ trial number, $< A_j, B_j >$ denote the $j^{th}$ pair of random numbers, and $P$ is a big prime number. Unless otherwise explicitly noted, the tuples emitted by the mappers are grouped by the first field — e.g., for tuples of the form $< u, \ldots >$, the intermediate key is field $u$. .................................................. 29

4. Illustration of the MapReduce algorithm for connected component detection. The algorithm terminates when $L(i) == L(i - 1)$. ................................ 30

5. Plot for number of protein clusters vs. $Q_e$ score. .......................... 44

6. Plot for number of protein clusters vs. $Q_p$ score. .......................... 45

7. Speedup of the Shingling Phase I. .................................................. 48

8. Effect of varying the number of map tasks on Shingling Phase I’s runtime for the Pfam-A-11M input. It can be observed that the optimal setting for the number of map tasks is $2K$ for this input. .......................... 49
9 Generation of the hybrid networks with three types of links 60

10 Clustering results: Cluster graphs showing the inter-community relationships for the a) people network, b) project network, and c) bipartite network. Each node in the cluster graph corresponds to one community/cluster that is detected in the underlying network, and an edge between two cluster nodes corresponds to the presence of (inter-cluster) edges between any two vertices of those clusters. The thickness of an edge is a relative indication of the number of inter-cluster edges between the two clusters. 70

11 Clustering results: cluster size distribution for the a) 1-mode people network, b) 1-mode product network, and c) bipartite network. 72
Contents

1 Introduction 1

2 Background and Related Work 7
   2.1 Clustering: Algorithms, Approaches and Applications . . . . . . . . . . . . . 7
   2.2 Parallel programming models . . . . . . . . . . . . . . . . . . . . . . . 10
      2.2.1 Shared memory . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
      2.2.2 Distributed memory . . . . . . . . . . . . . . . . . . . . . . . . . 12
      2.2.3 Accelerated computing . . . . . . . . . . . . . . . . . . . . . . . . 14

3 Algorithms for Large-Scale Graph Clustering 15
   3.1 Problem definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   3.2 Overview of the Shingling heuristic . . . . . . . . . . . . . . . . . . . . . . 16
   3.3 The design of a new variant for the Shingling heuristic . . . . . . . . . . 19
   3.4 Parallel clustering algorithms . . . . . . . . . . . . . . . . . . . . . . . . 23
      3.4.1 pClust-mr: A MapReduce algorithm for the Shingling heuristic . . . 24
      3.4.2 A MapReduce algorithm for the normalized Shingling heuristic . . . 28
      3.4.3 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

4 Experimental Results 31
   4.1 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
   4.2 Qualitative results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
      4.2.1 Qualitative evaluation of the Standard Shingling heuristic . . . . . 32
4.2.2 Qualitative evaluation of the Normalized Shingling heuristic . . . 35
4.3 Metrics of cluster annotation quality . . . . . . . . . . . . . . . . . . . 38
4.4 Performance results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
  4.4.1 Performance of the Hadoop-based implementation . . . . . . . . . 45
  4.4.2 Performance of the MPI-based implementation . . . . . . . . . . . 50
4.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

5 Cluster-Based Approach for Socio-Technical Coordination in Open-
Source Communities 55
  5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
    5.1.1 Socio-technical coordination . . . . . . . . . . . . . . . . . . . . 55
    5.1.2 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
    5.1.3 Research gap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
  5.2 Proposed Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
    5.2.1 Modeling products and communities as hybrid networks . . . . . 60
    5.2.2 Clustering techniques for bipartite and 1-mode networks . . . . . 62
    5.2.3 Cluster Analysis Methodology . . . . . . . . . . . . . . . . . . . . 63
  5.3 Case study: DRUPAL . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
    5.3.1 Data collection . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
    5.3.2 Summary of Network Clusters . . . . . . . . . . . . . . . . . . . . 69
    5.3.3 Link–Link comparison . . . . . . . . . . . . . . . . . . . . . . . . 72
    5.3.4 Cluster–Cluster comparison . . . . . . . . . . . . . . . . . . . . . 74
    5.3.5 Link–Cluster comparison . . . . . . . . . . . . . . . . . . . . . . 74
  5.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
6 Conclusions and Future Research Directions 78

Bibliography 80
Chapter 1

Introduction

Biological data, both naturally occurring and synthetically generated, lend themselves well to graph-based representations. Vertices can be used to represent experimentally acquired data and edges (weighted or unweighted, directed or undirected) can be used to represent pairwise relationships between the data. Graph-based representations are a popular way to model problems in computational biology. Once modeled as a graph, various scientifically interesting questions can be posed on the data and they typically translate into performing some kind of graph operations — e.g., performing an Euler tour or Hamiltonian path for genome assembly, finding hubs and critical paths in gene regulatory networks, finding connected components to group expressed sequences (transcriptomics), and clustering, which forms the focal point for this study.

Loosely defined, the clustering formulation targeted in this study is as follows: Given an input graph $G(V, E)$ with $n$ vertices and $m$ edges, “clustering” is the act of grouping vertices into tight-knit clusters, where the members of each cluster are closely related to most (if not all) other members of the same cluster, and sparsely related (if at all) to the members of other clusters. A scalable clustering solution that models this formulation can be useful to a number of applications within computational biology. For instance: it can be used to reduce redundancy within sequence repositories; identify complexes within metabolic networks [6]; identify core groups of proteins that constitute a protein family [73, 91, 94] and in the process also help assign family memberships for newly
found peptide candidates [94]; help in the construction of mass spectral libraries for peptides [53]; and can be used to condense the space of plausible computer-generated phylogenetic trees [63].

In standard literature, the above clustering formulation is also sometimes referred to as *dense subgraph detection*. It is to be noted that this is subtly different from community detection, and from graph partitioning to a larger degree. In the case of graph partitioning, vertices are partitioned into a pre-specified number of roughly equal-sized groups. In clustering though, clusters are allowed to have different sizes, and the number of clusters and their size distribution are both unknown at input. While this is also the case with community detection, such methods generally do not allow for control over the densities of the output communities. Instead their objective function is to optimize the overall clustering modularity [70] — a related measure which seeks to identify natural divisions that exist in a given network.

Despite its potential to address a broad range of problems, the use of clustering in real world bioinformatics applications has been rather limited, with only a handful of projects benefiting from it at large-scale (e.g., [94]). The reason for this limited usage is the lack of scalable computational tools. Finding clusters is a data-intensive operation and it can easily become compute-intensive as well, depending on the heuristics used. The problem is equivalent to the problem of finding variable-sized dense subgraphs (or quasi-cliques), and theoretically speaking, several of the corresponding optimization problems are computationally hard problems [1, 30, 50] or with large degree polynomial methods [69, 73]. Therefore, faster approximation heuristics need to be used in practice. However, even such heuristics can be difficult to implement in parallel because of the irregular data access and computation patterns that they generate for different inputs.
In 2005, Gibson et al. developed an efficient graph clustering heuristic called Shingling [34]. Posed as a dense subgraph detection problem for web community detection, their approach uses a randomized sampling method to iteratively identify and group vertices that share subsets of neighbors in common. The serial version of this heuristic was applied in the context of metagenomic protein family detection [91]. Put briefly, this approach, called pClust, transforms the problem into one of bipartite graph clustering so that the approach developed by Gibson et al. can be used. The results [90, 91] on input sets of size up to 1.2 million amino acid sequences showed both run-time and quality (sensitivity) advantage over approaches that use other heuristics. Despite its advantages, the implementation of the clustering step (i.e., pClust) is serial and does not scale beyond a graph containing $15K-20K$ vertices on a desktop computer with 2 GB RAM due to memory requirement. To make it scalable for larger inputs, a two-step, albeit indirect, approach was devised by which the large graph problem is first broken into connected components, and subsequently the sequential code is run on the individual connected components to output clusters. Owing to the simple observation that dense subgraphs cannot cut across connected components and the expectation that the connected components in real world graphs tend to be large in number and small in sizes, this approach worked for clustering a set containing 1.2 million sequences (vertices). However, there is no guarantee it will work for larger inputs. In the worst case, the size of the largest connected component could become comparable to the size of the original input graphs.

While the Shingling heuristic is effective to detect dense clusters, the analysis also revealed a tendency of the heuristic to leave out a significant fraction of vertices as singletons (i.e., vertices not recruited as part of any cluster). In other words, the heuristic
was more effective in capturing core elements of a cluster in an attempt to improve intra-cluster edge density albeit at the expense of peripheral elements. This is expected as the original heuristic was designed for the problem of link spam detection in internet graphs, which tend to be represented in the densest parts of the networks. However, for most biological graph inputs, it is desirable to allow some fringe elements to become part of larger, dense clusters. This is because clustering is often used as a way to functionally annotate new members of a family (e.g., proteins), or to consolidate large volumes of sequence data into fewer, non-redundant subgroups for facilitating further downstream processing and discovery. The importance of recruiting members into clusters (without significantly diluting the density of clusters) becomes even more pronounced while analyzing data from environmental microbial communities (i.e., metagenomics data), where it is typically challenging to annotate a significant fraction of the input data owing to the scattered nature in the sequencing (sampling) procedures [37].

Furthermore, for many biological graphs, there is information available on edge weights that need to be incorporated during clustering. For instance, in protein family identification, the degree of sequence similarity between two protein sequences could be used as edge weights (between the corresponding two vertices) in determining the composition of a family.

In this dissertation, we present novel clustering heuristics that are better suited to overcome the above outlined scalability limitations while also significantly enhancing the quality of the output clustering.

Graph-based representation is also widely used in social sciences to represent communication between individuals. Clustering of social networks helps researchers to capture the properties of social communities and identify underlying mechanisms that could be
overlooked by traditional methods. The analysis of rapidly growing virtual communities represent a significant challenge due to their large size and indirect ways of interaction between members. We present a cluster-based approach to study these networks that allows to interpret communication patterns within the community.

The key contributions in the dissertation can be summarized as follows:

**Algorithm development:**

1. We present novel MapReduce algorithms and implementations to efficiently parallelize the serial Shingling heuristics on large-scale distributed memory clusters running as either Hadoop or MPI platforms. Experimental results show linear scaling on thousands of processors, and multiple orders of magnitude improvement in both on time to solution and problem scaling.

2. We present a new variant of the Shingling heuristic that shows significant qualitative improvements over the standard serial version. We also present an extended version that can handle weighted graph inputs — something that the standard version of the algorithm is not equipped to handle.

**Application and evaluation:**

1. We introduce a new formulation for clustering protein sequences based on domains, and present a novel application of MapReduce implementation to address this formulation. Experimental results show Hadoop implementation scales linearly in its most time-consuming phase on up to 64 cores on a small real world graph containing 8.41M vertices (8,407,839 proteins and 11,823 domains) and 11M edges (protein to domain connections). Relative to the serial implementation \( pClust \), this
was sufficient to enhance the problem size reach by about two orders of magnitude (\(\times 10^4\) to \(\times 10^6\) vertices) in roughly the same amount of time.

2. We present extensive experimental results for MPI implementation on a real world metagenomics graph containing 10.3M vertices and 640M edges and its subsets. These results demonstrate a) significant qualitative improvements over the standard heuristic, both with and without using edge weights; and b) significant performance improvements over the previous Hadoop implementation, resulting in the analysis of the entire test graph (640M edges) in less than a minute on 4,096 core of a distributed memory cluster.

3. We develop and evaluate two approaches to measuring socio-technical coordination in virtual communities based on clustering techniques. We illustrate the approaches using a case study from an open source software development community. The proposed approaches present a broader and more encompassing view of coordination within open source communities.

The dissertation is organized as follows: Chapter II presents a brief overview of the related literature on clustering and parallel programming models. In Chapter III, we first describe the clustering heuristic and then present our new variant of the heuristic and MapReduce algorithms. Experimental results are presented in Chapter IV. In Chapter V, we present our cluster-based approach for socio-technical coordination in open-source communities and Section VI concludes the dissertation.
Chapter 2

Background and Related Work

2.1 Clustering: Algorithms, Approaches and Applications

The problem of finding a maximum dense subgraph within an input graph is solvable in polynomial time [18, 35, 55]. However, the more practically appealing constrained variants of this problem, viz. of finding a densest subgraph of size equal to k, or at least k, or at most k, have all shown to be NP-Hard [1, 30, 50]. Our problem represents a more generalized version of these variants, wherein the goal is to find multiple, variable-sized dense subgraphs, satisfying density and size cutoffs. Consequently, approximation heuristics need to be pursued. Dense subgraph detection problems can also be defined over bipartite graphs. This way of modeling the problem is particularly effective when relationships are defined over data of two different types, and find frequent usage in the context of web communities in the Internet data (e.g., [34]). It turns out that the bipartite graph formulations are also NP-Hard [30, 56].

There is a rich body of clustering related literature in the context of biological applications. A considerable segment is devoted to gene expression/microarray analysis and transcript/genome assembly (reviewed in [24, 28, 46]). For these applications, however, clustering is not generally modeled as a graph problem (except for those that
involve string graphs in short read assembly), and simpler agglomerative techniques (e.g., neighbor joining) and single linkage clustering suffice in practice due to the nature of sampling. Such methods also tend to create loose clusters and suffer from error propagation during incremental construction. A different class of applications benefit from a clustering formulation based on detecting denser communities within biological data (e.g., [6, 29, 44, 58, 94]). Brohee et al. [12] evaluate clustering techniques for analyzing protein-protein interaction networks. Andreopoulos et al. [2] present a broader, and more conceptual survey of different clustering applications that benefit from tighter cluster techniques.

Independently, in other areas of computing such as social and cyber networks, numerous algorithms have been developed for community detection (reviewed in [54, 56, 68]). M.E.J. Newman, in his pioneering work on discovering community structure from networks [69], developed a divisive clustering method that detects and removes edges, one at a time, that are most likely to cut across cluster partitions. To detect such edges, the approach calculates the betweenness centrality index for all edges in the graph. However, removal of an edge introduces the need to recompute the centrality index for all edges. While this approach has been demonstrated to be highly effective in discovering community structure [69], the cost of computing centrality index and the need for recomputing after every step make the algorithm slow (Ω(n^3) even for sparse graphs with n vertices) and practical for only up to n ≈ 10^4 on single compute nodes. Nevertheless, there are shared memory parallel algorithms such as [60] for efficiently calculating betweenness centrality on graphs. A different approach [73] works on weighted graphs, where edge weights are distance measures, and uses Minimum Spanning Trees (MST) for clustering by taking advantage of the property that closely related groups tend to map to subtrees
within an MST. However, this method can also be time consuming ($\Omega(n^2)$) and the method has not been compared with other methods making it difficult to assess its quality. More recently, Blondel et al. [8] developed another community detection method, which is loosely based upon one of Newman’s previous modularity-based methods [19]. This method, referred to as the Louvain method, has now emerged to be a widely used method for community detection.

Gibson et al. developed the Shingling heuristic for identifying dense internet communities [34]. The underlying method (described in Section 3.2) uses a bipartite graph approach along with a random sampling procedure and secondary sorting. In [91], this method was adapted to work for graphs constructed using protein sequences as vertices and the presence or absence of pairwise full-length similarity (or homology) to mark the presence or absence of edges, respectively. Given an input homology graph $G(V, E)$ with $n$ vertices and $m$ edges, $pClust$ implementation’s runtime is dominated by sorting step which sorts $O(n \times c)$ values, where $c$ is a parameter (typically $\geq 100$); and its memory complexity is $O(n \times c^2)$.

Chapman and Kalyanaraman [17] developed a parallelization of the Shingling heuristic using OpenMP for shared memory machines. The shared memory implementation reduces this runtime through the use of a hash table and by parallelizing under the OpenMP model. It also reduces the memory complexity to $O(n \times c)$. While the complete details of this algorithm are not within scope of this dissertation manuscript, the main ideas behind the shared memory parallelization are as follows: Individual threads can be tasked with generating shingles from the input graph stored on the shared RAM, and the intermediate grouping of source vertices by their shingles was implemented using a shared implementation of the hash table data structure. And for the final cluster
10

enumerating phase, the algorithm implements a shared implementation of the union-find data structure. While this work was able to produce scalable results, the scale of its application is limited to the input sizes that can be supported on a shared memory platform, and the scaling is limited to the number of cores available on such platforms. For this reason, we present fully re-engineering parallel solutions that are suited for distributed memory machine architectures, which have orders of magnitude larger number of cores and memory capacity. The solution space significantly differs from that of shared memory implementations and presents new challenges - for instance, data structures such as hash tables and union-find do not map well into distributed settings.

2.2 Parallel programming models

Parallel computing provides a way to execute many calculations simultaneously, taking advantage of the fact that large problems can often be divided into parts, which are then solved concurrently. A useful way to classify parallel computing architectures is based upon the number of concurrent instruction and data streams available [32]. SISD (Single Instruction, Single Data) architecture behaves as a sequential machine, whereas MISD (Multiple Instruction, Single Data) uses pipelining. The two most important categories are SIMD (Single Instruction, Multiple Data) and MIMD (Multiple Instruction, Multiple Data). The SIMD architecture consists of several processors that perform the same instruction synchronously on different data and is suited for specialized problems such as graphic processing [3]. In the MIMD architecture, processors are capable of performing different instructions on different data sets. MIMD is the most used parallel architecture [7] and can behave at certain times as any of the other discussed architectures [5]. The
following sections will describe the ways parallel computing differs in terms of their architectural organization.

2.2.1 Shared memory

In this model, a small number of processors have access to the same single memory. Writing code that uses the shared memory is considered easier than writing for other architectures because it is closer to a sequential process. However, there are still synchronization problems between processors [7], which often reduces scalability.

One method to implement parallelism with shared memory is by using threads, each storing its own environment variables. The main advantage of threads is that it is typically easy to add thread structure to previously available sequential code to provide parallelism. However, multiple threads read and write concomitantly to the same memory location, which leads to unpredictable execution and race conditions where one thread corrupts the value of a variable for other threads [71]. In order to solve this problem, serial code sections must be defined to enforce mutual exclusion of threads. In a similar fashion, when some tasks need to be completed in a given order [64], barriers and conditional variables need to be used in order to assure synchronization. Although these mechanisms make the program to run correctly, they create additional delays when threads need to wait to enter the sequential sections.

An example of the thread framework is OpenMP, where thread management and workload partitioning is handled implicitly (without programmer involvement) by the compiler to produce optimal performance. Another common thread framework is Pthreads,
which asks the programmers to declare thread management explicitly in order to distribute work evenly [49]. Although this obviously requires more work from part of the programmer, it provides flexibility to deal with the underlying hardware architecture.

2.2.2 Distributed memory

Each processor on a distributed memory system has its own separate memory and connect to each other via network. Distributed memory machines can be further classified into massively parallel processors (MPPs) and clusters. MPP’s are computer-intensive parallel machines which are extremely scalable, and may have thousands of processors. The main disadvantage of MPP’s is that they are only suitable for certain tasks and are not used in general purpose development.

Clusters are more common distributed machines, made up of smaller multi-core machines, referred to as nodes, which can work independently or together as one computer system, something which MPPs cannot do. Communication between nodes takes place by passing messages across a network, and locally on the same node via the shared memory bus. Therefore, clusters are hybrids that allow flexibility for different algorithmic purposes and can produce very high performance, having the advantages of both shared memory architectures (fast local communication) and distributed architectures (scalability) [7].

**Message Passing Interface.** The most commonly used framework is MPI (message passing interface) [65], which implicitly creates workers and synchronizes them. The main advantage of the MPI model is its extreme scalability and flexibility, allowing the
programmer to manage workloads according to the desired architecture. However, work-
load management creates noticeable coding difficulties and, given that it is hard to fit
MPI to previously available sequential code, it is often better to rewrite the algorithm
implementation. More importantly, there are performance costs in sending messages
across a slow network and often processors must wait for a response from other pro-
cessors. In order to solve these problems, programmers can use buffers and different
communication topologies, but this comes at the expense of higher code complexity
and, therefore, increased development time.

MapReduce. Recently, distributed programming model called MapReduce [23]
raised in prominence among researchers and developers. MapReduce helps to process
and generate large data sets, where computational processes may use data stored on
both structured or unstructured systems. Briefly, MapReduce makes use of two func-
tions: The map function takes a series of key/value pairs and gives zero or more output
key/value pairs, whereas the Reduce function is applied to all the processors in parallel,
and it returns a list of final output values. MapReduce was introduced by Google to pro-
cess large amount of data on parallel computers and has several advantages compared
to other models. First, the model is easy to use, since it hides the details of paral-
lelization, fault-tolerance, locality optimization, and load balancing. MapReduce thus
allows developers without experience with distributed systems to write applications in
their programming language of choice. Second, a large variety of problems are easily ex-
pressible as MapReduce computations, as can be shown by the diverse applications (e.g.
generation of data for Google’s web search service, sorting, data mining, machine learn-
ing, and many other systems). Third, MapReduce enables scaling of applications across
large clusters of machines comprising thousands of nodes, with built-in fault-tolerance
allowing very high performance.

2.2.3 Accelerated computing

Another type of parallel hardware architecture is the Graphics Processing Unit (GPU or graphics card). The architecture consists of a number of small processors performing the same instruction synchronously on different data, which is especially suited for problems characterized by a high degree of regularity such as image analysis and visualization [3].

The main advantage of GPUs is the high processing power that is obtained at a small cost. This performance gain can be achieved by using hundreds of very small processing cores which have regions of fast shared memory [72]. GPU vendors such as NVIDIA and AMD developed their own parallel programming tools. The most commonly used GPU programming tool is CUDA, which has the great advantage that the developer does not need to allocate threads. However, there are disadvantages of compared to other programming frameworks, as sequential programs have to be re-written for CUDA and optimal performance gains require a high level of knowledge of each particular graphics card.
Chapter 3

Algorithms for Large-Scale Graph Clustering

3.1 Problem definition

Let $G = (V_l, V_r, E)$ denote an undirected bipartite graph\footnote{Note that if the input is not a bipartite graph but a 1-mode graph $G' = (V', E')$, then it can be trivially converted into an equivalent bipartite graph $G = (V_l, V_r, E)$ by setting $V_l = V_r = V'$ and drawing edges from $E'$ between the two vertex partitions.} with $n$ vertices ($n = |V_l| + |V_r|$) and $m$ edges. Let $n_l = |V_l|$ and $n_r = |V_r|$. Let $\Gamma(u) = \{v \mid (u, v) \in E\}$ denote the set of links for vertex $u$. As a convention, we will use $u$ to denote a vertex from $V_l$ and $v$ to denote a vertex from $V_r$.

Defined loosely, a dense subgraph in a bipartite graph is a subgraph containing subsets $V'_l \subseteq V_l$ and $V'_r \subseteq V_r$ such that each vertex in $V'_l$ is connected to most of the vertices in $V'_r$ and sparsely (if at all) connected to vertices in $V_r \setminus V'_r$. In other words, the vertices in $V'_l$ share most of their links in $V'_r$.

Given $G(V_l, V_r, E)$, our goal is to find a set of variable-sized dense subgraphs within $G$. Although one could associate a notion of maximality for this problem definition, wherein each group identified cannot be expanded with more vertices without becoming less dense, we avoid such a definition because that would necessitate a fixed density-cutoff and the heuristic we implement does not use such a cutoff.
3.2 Overview of the Shingling heuristic

Definition 3.1 Given an integer constant $s$, a “shingle” of a vertex $u$ [11] is defined as an arbitrary $s$-element subset of $\Gamma(u)$.

If two vertices are part of a dense subgraph, they can be expected to share a large fraction of their neighbors in common. Therefore, a brute-force way to detect vertices that are part of the same dense subgraph would be to compute $|\Gamma(u_i) \cap \Gamma(u_j)| / |\Gamma(u_i) \cup \Gamma(u_j)|$ for every pair of vertices $u_i, u_j \in V_t$. The Shingling heuristic takes a randomized sampling approach to reduce this quadratic search space. More specifically, it obtains random samples of size $s$ (called shingles) from $\Gamma(u)$ for every vertex $u \in V_t$, and compares them against one another. If two vertices are part of the same dense subgraph, then by definition they should also share most of their links and hence with a high probability are also expected to share a shingle [10]. However, this cannot be guaranteed because the shingles are small, fixed-size samples obtained randomly (using a pair of random numbers $A$ and $B$). Therefore, to improve the probability that such vertex pairs are detected, the algorithm generates shingles over $c$ random trials.

Generation of $c$ shingles for any vertex $u \in V_t$ that has at least $s$ links is achieved as follows [34]: First, $c$ random permutations of the vertices in $\Gamma(u)$ are obtained using
a fixed set of \( c \) random number pairs \( \{<A_j, B_j>| j \in [1, c]\} \). The top \( s \) elements within each permutation are then said to represent a shingle. The random permutation of \( \Gamma(u) \) for a given random trial \( j \in [1 \ldots c] \) is obtained as follows: Assume that every \( v \in \Gamma(u) \) is associated with a unique integer id. Then a bijection from the set \( \Gamma(u) \) to a new set \( \Gamma^j(u) \) is computed by taking every \( v \in \Gamma(u) \) and mapping it to an element \( v^j \in \Gamma^j(u) \) such that \( v^j = (A_j \times v + B_j)\%P \), where \( P \) is a big prime number. Consequently, sorting \( \Gamma^j(u) \) yields the random permutation for \( \Gamma(u) \) for trial \( j \). A permutation thus obtained preserves the min-wise independent property that guarantees, with high probability, that vertices of a densely connected subgraph would also share a shingle [10, 11, 34].

The above heuristic is implemented in \textit{pClust} [91] in three phases (also see Figure 1):

**Shingling Phase I:** Using input \( G(V_l, V_r, E) \) in its adjacency list form, the algorithm first generates \( c \) shingles for each vertex in \( V_l \) as described above. In our implementation, the sorting required to generate a shingle from \( \Gamma^j(u) \) is implemented by performing an on-the-fly enumeration of \( \Gamma^j(u) \) and alongside keeping track of an \( s \)-sized array that records the minimum \( s \) elements at any point of time through a simple insertion sort. The small values of \( s \) expected to be used in practice (typically under 10) justify a simple insertion sort-based approach. Let \( s_j \) denote a shingle generated for some vertex during the \( j^{th} \) random trial, and assume that it is in an integer representation obtained using a hash function. Since the same shingle \( s_j \) could have been generated by multiple vertices in \( V_l \), a sorting is done to gather all vertices that generated each shingle. This shingle is done once for each random trial (so that shingles from different trials do not get mixed). Let \( L(s_j) \) denote the set of vertices which generated a shingle \( s_j \). The algorithm then outputs tuples of the form \(<s_j, L(s_j)>\). Note that these tuples collectively define a new
bipartite graph $G_I(S_1, V'_l, E')$ in its adjacency list form, such that $S_1$ represents the set of distinct shingles generated during this phase, and $V'_l \subseteq V_l$ represents the subset of vertices that contributed to at least one shingle. Therefore, the output of this phase is $G_I$. We call the shingles in $S_1$ first level shingles.

**Shingling Phase II:** Using $G_I$ as the new input, the algorithm executes the same series as steps as in Phase I. This generates a new bipartite graph $G_{II}(S_2, S'_1, E'')$ in its adjacency list form, such that $S_2$ represents the new set of shingles generated during this phase (referred to as the second level shingles), and $S'_1 \subseteq S_1$ represents the subset of first level shingles that contributed to at least one second level shingle in $S_2$.

**Phase III - Connected component detection:** In the final reporting step, all connected components in $G_{II}$ are reported. Note that the connected components will be defined by first to second level shingle connections. To enumerate connected components, this step uses the classic union-find data structure. Consequently, the union of vertices in $G$ within each connected component of $G_{II}$ is reported as the output set of dense subgraphs.

The implementation has the following runtime complexity by stages:

- For Shingling Phase I, all edges are traversed $c$ times to generate $\Gamma^i(.)$ from all vertices, and the subsequent insertion sort to derive the shingles entails $s$ comparisons per edge form in every $\Gamma^i(.)$. One global sort of all shingles produced for each trial is performed using quick sort. Note that the phase outputs a total of $n_t \times c$ output shingles (although not necessarily distinct) across all trials. This yields a
run-time complexity of $O(m \times c \times s + c \times T_{\text{sort}}(n_l))$. It is worth noting that if $S_1$ denotes the set of first-level shingles, then $c \leq |S_1| \leq n_l \times c$, and the denser the input graph is, the smaller the value of $|S_1|$ tends to be;

- Similarly, for Shingling Phase II, the runtime complexity is $O(|E'| \times c \times s + c \times T_{\text{sort}}(|S_1|))$, where $E'$ is the set of edges in $G_I$ output by phase I; note that $|E'| \leq n_l \times c$ (or more precisely, equal to $|V'| \times c$). And if $S_2$ denotes the set of second-level shingles then $c \leq |S_2| \leq |S_1| \times c$.

- For Phase III, the runtime is $O((|S_1| + |S_2|) \times s \times \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function which is a small constant for all practical purposes.

The peak memory complexity of the algorithm is $O(max\{m + n, |E'|, |E''|\})$.

### 3.3 The design of a new variant for the Shingling heuristic

Although the standard heuristic has been shown to be effective at capturing a majority of the dense subgraphs [34], its application to biological graphs such as protein sequence homology graphs reveals that a significant fraction (up to a third) of the vertices do not get recruited into any of the clusters for some of the real world data sets we tested (please see experimental results section). Such vertices, referred to as “singletons”, typically tend to be low degree vertices. The potential cause for these singletons is that either these vertices have degree less than parameter $s$ and as a result no shingles get generated, or these vertices with low degree have larger degree neighbors which in
turn fail to generate the same shingle as their low degree neighbors during the random sampling method.

Figure 2 illustrates a couple of contrasting instances on how low degree vertices should be handled during clustering. In Figure 2a, vertices $u_1 \ldots u_4$ are low degree vertices who should be assigned the same clusters as their larger degree neighbors $u'_1 \ldots u'_4$. The standard heuristic, however, will mark these peripheral elements of the cluster as singletons (for $s > 2$). On the other hand, not all low degree vertices should follow their larger degree neighbors in their cluster assignment. For example, in Figure 2b, the vertex $v_1$ is a weak bridge between the two dense clusters containing its neighbors $v_2$ and $v_3$ respectively, and therefore should be left out as a singleton, keeping the two clusters separate.

**Algorithm 1** Normalized Shingling (Input: $G(V, E), s, c$)

```
Generate $c$ random number pairs $\{< A_j, B_j > \}$;
$P \leftarrow$ a big prime number
$C \leftarrow$ normalizing constant (default to 100)
for $u \in V$ do
    Let $\Gamma(u) \leftarrow \{v|v$ is a neighbor of $u\}$
    for $v \in \Gamma(u)$ do
        /* Normalize edge weight for every edge*/
        $w_{u,v} \leftarrow w_{u,v} \times C / \sum_{v' \in \Gamma(u)} w_{u,v'}$
    end for
    for $j = 1 \rightarrow c$ do
        Generate multi-set $\Gamma^j(u) \leftarrow \{w_{u,v}$ copies of $v^j|v^j = (A_j \times v + B_j) \% P, v \in \Gamma(u)\}$
        Let shingle $s^j(u) \leftarrow$Minimum $s$ elements of $\Gamma^j(u)$
        Store tuple of the form $< s^j(u), u >$
    end for
end for
for $j = 1 \rightarrow c$ do
    Sort all tuples $< s^j(u), u >$ by their shingle ids
    Output transformed graph $G_I(V', E')$, where $V'$ is the set of all shingles and $E' = \{(s, u)|u$ generated $s\}$
end for
```
In Algorithm 1, we present a new algorithmic variant of the standard shingling heuristic. This new algorithmic variant is not only better suited to overcome the challenges outlined above, but also automatically extends the scope of clustering to graphs with edge weights. If the graph is unweighted, then we trivially assign a weight of 1 to every edge. In what follows, we explain the algorithm assuming weighted inputs.

The main idea of the new algorithm is as follows. Let $u$ be a source vertex. Given that a shingle is a random sample of size $s$ from $u$’s neighbors, the probability that a neighbor $v$ becomes part of a shingle of $u$ should be dictated by the weight of the edge connecting $u$ to $v$ ($w_{u,v}$). This led us to develop a simple normalization method, in which the edge weights of all edges connected to $u$ are recomputed to represent their relative importance to that vertex and the normalized weights are subsequently used for shingle generation. As the original algorithm is oblivious to edge weights during shingle generation, we modified the shingle generation process so that it now conceptually treats an edge with an integer weight $x$ as implicitly containing $x$ copies of unit weight edges (i.e., multi-edge), prior to applying the standard shingle generation method from the original algorithm. To guarantee integer edge weights, we multiply the normalized edge weights by an integer $x$. 

![Figure 2](image.png)

Figure 2: Cases illustrating the decisions to be made on the clustering of low degree vertices.
weight by a constant $C$ and ignore the decimal places. In other words, for $C = 100$, this implies that the edge weights are normalized to the scale of 100. Ideally, one would like to set the value of $C$ to the value of the maximum degree (or weighted degree) of a vertex, as otherwise there is a possibility that the normalized weight value evaluates to less than 1 (and therefore interpreted as 0). For instance, in an unweighted graph, if a vertex has a degree of more than $C$, then each edge’s normalized weight will evaluate to zero, which implies that these edges will never participate in the shingling process. Forcibly resetting the normalized weight to 1 in such cases intuitively provides those edges a chance to participate in the shingling process. (In all our experimental studies (please see Section 4), we used this trick combined with a value of $C = 100$ and this combination yielded good results.)

The above idea of normalization also helps in the recruitment of singletons into clusters. For instance, a low degree vertex which has less than $s$ neighbors, would now be able to generate shingles due to normalized weights and subsequently participate in clustering. A caveat is that the new “shingles” that we generate using this new variant could potentially have a size smaller than $s$ (as duplicate occurrences of a vertex within a shingle should be removed). For instance, in Figure 2a all shingles generated from vertex $u_1$ will be the set $\{u'_1\}$. Yet, from our experiments, we observe that this is not a problem and that the large number of random trials ($c \approx 100$) is still sufficient to cluster these low degree vertices into their appropriate neighboring clusters.

We also note that the normalization approach introduces a possibility, although with very low probability, that two disparate cluster structures get combined into a larger cluster. For the example shown in Figure 2b, let us assume unit weight edges connected originally to $v_1$. After normalization to the scale of 100, these edges will be assigned
weights of 50 each. This leads to the possibility of $v_1$ enumerating any of these three shingles: \{$v_2, v_3$\}, \{$v_2$\} or \{$v_3$\} during different random trials. Given the same possibility persists for the other neighbors of $v_2$ and $v_3$ within each of their individual clusters, it becomes theoretically possible that the clusters are combined based on common shingles. However, the probability of such merging events are expected to be extremely low owing to the larger degrees of the vertices in the individual clusters, and the fact that these shingles need to be produced during the same random trial in order to lead to merges. Although we acknowledge this to be a theoretical possibility for false merging by the normalization-based method, we did not observe such events in our empirical evaluation.

### 3.4 Parallel clustering algorithms

We present parallel algorithms for the Shingling heuristic for unweighted and weighed graphs using the MapReduce paradigm [23]. We chose to evaluate the MapReduce framework for distributed memory not only because it is rapidly becoming a popular standard for data-intensive applications but also because the fundamental operations required to implement the Shingling heuristic naturally lend themselves to a map-reduce structure. In designing our MapReduce algorithms for the above dense subgraph detection algorithms, we preserved the overall algorithmic structure in $pClust$ but implemented each phase in a way that is better suited to the distributed memory setting of MapReduce. In what follows, we describe the proposed parallel algorithms along with associated design challenges for the three phases.
3.4.1  \textit{pClust-mr: A MapReduce algorithm for the Shingling heuristic}

\textbf{Design challenges:} The serial algorithm assumes an adjacency list format. However, an adjacency list representation may not be appropriate under a MapReduce in some cases. Assuming each vertex’s list occupies a line in the input file, the MapReduce framework would distribute the input lines in parallel to the map tasks. Because a line is viewed as an atomic unit, this assumes that there is sufficient memory on the mappers to store at least one line of the input. And since in the worst-case, a vertex’s adjacency list could be $\Theta(n)$, it may not be a scalable option. Secondly, the adjacency lists of vertices could vary in length, implying a potential scenario where map tasks with long lists could become a parallel run-time bottleneck. This provides direct and easy access to all of the links for any given vertex to generate the shingles. An alternative option is to use the adjacency matrix representation, which would make the line lengths uniform, but becomes too expensive in storage ($\Theta(n^2)$).

Another option is to store the graph as a \textit{simple list of edges}. This representation is naturally suited for MapReduce because each edge $<u,v>$ is in the form of a $<\text{key},\text{value}>$ tuple that conforms to the input/output types of the framework. This representation also automatically keeps the line length short and uniformly sized. However, the challenge rests on being able to generate a shingle because the edges of a given vertex could potentially be scattered across different map tasks.

We devise algorithms for both - edge list and adjacency list - formats and compare their performance.
Shingling Phases I and II (edge list-based format): Our algorithm is illustrated in Figure 3. The main idea is as follows: The input is a simple list of edge tuples \(<u, v>\), where \(u \in V_l\) and \(v \in V_r\), one edge per line. Instead of attempting to generate the \(c\) shingles corresponding to each input vertex \(u\) at the Map phase, we defer that task to the Reduce phase and instead only generate all the \(c\) forms for every link from \(u\), viz. \{\(v_1, v_2, \ldots, v_c\)\} at the Map phase. Basically, each mapper at any given point of time, takes as input an edge tuple \(<u, v>\) and emits \(c\) tuples of the form \(<u, v_j>\), where \(j \in [1 : c]\). This can be implemented as a strictly local operation, assuming all the \(c\) random number pairs \(<A_j, B_j>\) are made available at initialization time at each mapper.

Next, the tuples emitted by the mappers are grouped using the source vertex \(u\) as the intermediate key. This sends the list of all tuples generated for a given \(u\) to a single reducer. Note that the length of this list could be anywhere in the range \([c \ldots (c \times |\Gamma(u)|)]\).

At the reducer designated for \(u\), if one were to store the entire list of its tuples to sort and generate the \(c\) shingles, it would imply a memory complexity of \(\Theta(c \times n)\) in the worst-case. Therefore, we devised a different approach in which the memory complexity per reducer becomes independent of the length of the tuple list. This is achieved as follows: Every reducer maintains a set of \(c\) arrays of size \(s\) each. We call these “trial arrays”, and denote the arrays as \(T_1 \ldots T_c\). Each \(T_j\) keeps track of the minimum \(s\) elements seen so far for that trial. When a new \(<u, v_j>\) tuple is streamed in, the element \(v^j\) is inserted into \(T_j\) so as to maintain the sorted order within \(T_j\). Given the small values expected for \(s\) (typically, \(s \leq 5\) in practice), this is implemented using a simple linear scan as in insertion sort, taking at most \(s\) comparisons per insert. After all the tuples have been processed, the elements stored in the trial arrays represent the \(c\) target shingles.
The reducer then assigns each shingle an integer id (using a hash function). To ensure that different permutations of the same $s$ elements are assigned the same shingle id, we internally sort each shingle by its constituent vertex names. This adds another $O(s)$ to the processing time per trial at output. Consequently, the reducer emits tuples of the form $<s, u>$ from each $T_j$, where $s$ is a first level shingle and the $u$ is a vertex that generated it.

Note that the global set of all tuples $<s, u>$ emitted by all reducers is the edge list representation for graph $G_I$. It is possible that this list emitted by the reducers contain some duplicate tuples (as different trials of the same source vertex could have identified the same shingle). To eliminate such duplicates, we implemented a simple Map-Reduce phase where identical tuples (lines) are grouped and only one copy of each tuple is emitted. This is identified as the Unique I phase in Figure 3. The resulting non-redundant list of edges is passed as input to Phase II. It should be easy to observe that the algorithm for Phase II is identical to that of Phase I.

**Analysis of the Shingling phases**

The disk storage complexity is $\Theta(m)$, because the algorithm operates using an edge list. The memory complexity at each mapper is $O(1)$ and each reducer is $\Theta(c \times s)$. As for the run-time complexity, let $p$ denote the total number of processors. Let us also assume, for sake of analysis, that the number of map tasks ($p_m$) plus the number of reduce tasks ($p_r$) is equal to $p$. Then the run-time complexity for each mapper is expected to be $O(\frac{m \times c}{p_m})$. As for the reducers, assuming a balanced distribution of the grouped tuples, each reducer is expected to take $O(\frac{m \times c \times s}{p_r})$ time. The overall run-time complexity for Phase I is expected to be $O(\frac{m \times c}{p_m} + T_{group}(m \times c) + \frac{m \times c \times s}{p_r})$, where $T_{group}(m \times c)$ represents the time for the MapReduce framework to group $m \times c$ tuples.
Phase II’s analysis is similar except that the input number of edges in $G_I$ is bounded by $O(n_l \times c)$ (worst-case represented by a sparse graph). Therefore, a run-time complexity of $O\left(\frac{n_l \times c^2}{p_m} + T_{\text{group}}(n_l \times c^2) + \frac{n_l \times c^2 \times s}{p_r}\right)$ follows. In practice, graphs are expected to be connected enough to imply significantly smaller than the theoretical bound.

**Phase III: Connected component detection**

For connected component detection, the idea of using an irregular data structure such as the union-find is not suitable under distributed memory setting. We developed a MapReduce algorithm which implements a technique that was originally used for conducting Breadth First Traversals for the MPI model [93]. The MapReduce algorithm is illustrated in detail in Figure 4. Define the label of vertex $u$ to be equal to the maximum vertex id in $u$’s connected component. Since the connected components are unknown initially, the algorithm starts by assigning the labels at every vertex to itself. Then an iterative approach follows: At the end of iteration #0, each vertex exchanges information with its immediate neighbors and accordingly updates its label. Subsequently, the labels from iteration $i - 1$ are used to update the labels at iteration #i. The algorithm terminates when the set of labels converges.

We note here that, after we developed this approach under MapReduce, we found out that there is a nearly-identical algorithm reported by Kang et al.[47]. The only difference between our algorithm and theirs is that our algorithm has a better memory complexity at the reducers, but uses a MapReduce sort (as opposed to a MapReduce group). More specifically, our reducer in $\text{UpdateLabels}(i)$ will work even under a streaming model, implying a $O(1)$ memory requirement (as opposed to $O(n)$ in [47]).
3.4.2 A MapReduce algorithm for the normalized Shingling heuristic

The new variant of the Shingling heuristic using normalization also maps directly to the MapReduce model. Each Map process takes as input the adjacency list for a given source vertex \( u \), and performs the following tasks:

Step 1) Normalize all the edge weights for \( u \).

Step 2) Generate \( c \) shingles for \( u \), and emit the shingles in the form of tuples \( < s, u > \),

where \( s \) is a shingle generated by \( u \).

The shingle structure contains information on the shingle id and the set of at most \( s \) vertices constituting that shingle. The shuffle stage sorts the tuples by their shingle ids.

Each Reduce process takes the list of all source vertices that generated a given shingle id and emits tuples of the form \( < s, u > \) where \( s \) is the shingle and \( u \) is every distinct source vertex in the list. The union of all the tuples emitted by all the reducers constitute the transformed graph \( G_I(V', E') \), which is passed as input to the next Shingling phase.

3.4.3 Implementation

All our implementations for the standard and normalized algorithms were in C/C++ using MapReduce-MPI library [76] that is written with standard MPI calls and compiled into a regular MPI program. The initial implementation of the standard Shingling heuristic was written in Java combined with Hadoop MapReduce API calls.
Shingling Phase I:

Input edge list (m edges):
\[ < u, v_1 >, \ldots, < u, v_m > \]

Map() \( f(v_1) \):
\[ < u, v_1^1 >, < u, v_1^2 >, \ldots, < u, v_1^c > \]

Reduce:\( Map() f(v_1)\):
\[ < u, v_1 > \]

\( T_1^c \):
\[ s, u \]

Unique I: (remove duplicate entries)

Shingling Phase II:

Input edge list (m edges):
\[ < u, v_2 >, \ldots, < u, v_m > \]

Map() \( f(v_2) \):
\[ < u, v_2^1 >, < u, v_2^2 >, \ldots, < u, v_2^c > \]

Reduce:\( Map() f(v_2)\):
\[ < u, v_2 > \]

\( T_2^c \):
\[ s, u \]

Unique II: (remove duplicate entries)

Connected component detection & cluster enumeration

Output clusters \( \{ C_1, \ldots, C_k \} \)

Figure 3: Illustration of our MapReduce algorithm for the Shingling heuristic. The figure shows the detailed Map-Reduce stages for Shingling Phases I and II. The function \( f() \) (inside Map()) takes as argument a vertex \( v \) in the form of its integer id and generates \( c \) forms of \( v \), viz. \( \{ v^1, v^2, \ldots, v^c \} \); where \( v^j = (A_j \times v + B_j) \mod P \), \( j \in [1 \ldots c] \) corresponds to the \( j^{th} \) trial number, \( < A_j, B_j > \) denote the \( j^{th} \) pair of random numbers, and \( P \) is a big prime number. Unless otherwise explicitly noted, the tuples emitted by the mappers are grouped by the first field — e.g., for tuples of the form \( < u, \ldots > \), the intermediate key is field \( u \).
CreateInitialLabels():
Input: Edge list (m edges)
Output: L(0)
Map()
Input: <u, v>
if u < v then
  emit <u, v>
else
  emit <u, u>
end if
Reduce()
Input: <u, list of values v>
Let v_i = arg max_v v
emit <u, v_i, ℓ_i>
Let L(0) = \{<u, v_i, ℓ_i> | \forall u \in V(G)\}

UpdateLabels(i):
Input: {Edge list (m edges)} concatenated with L(i - 1)
Output: L(i)
Map()
Input: <u, v, tag >
if tag is empty then
  emit <v, 1>, u >
else
  emit <u, 0>, v >
end if
Reduce()
Input: << u, tag >, list of values v >
Let u_i ← v_i for <u, 0>, v >
emit <u, u_i>
\forall << u, 1 >, v >:
emit <v, u_i>
Map()
Input: <u, list of values v>
Let v_i = arg max_v v
emit <u, v_i, ℓ_i>
Reduce()
Input: <u, list of values v>
emit <u, v_i, ℓ_i>

Figure 4: Illustration of the MapReduce algorithm for connected component detection. The algorithm terminates when L(i) == L(i - 1).
Chapter 4

Experimental Results

4.1 Experimental setup

The experiments were performed on 2 test platforms: i) the Hopper supercomputer at the National Energy Research Scientific Computing Center (NERSC). Hopper is a 1.28 petaflop/sec Cray XE6 consisting of 6,392 compute nodes made up of 2 twelve-core AMD ‘MagnyCours’ 2.1GHz processors and 32 GB RAM per node. Hopper’s compute nodes are connected by the Cray Gemini Network which is a custom high-bandwidth (8.3 GB/s), low-latency (<1µs) network with network topology of a 3D torus. The MPI library is a custom version of mpich2 for Cray XE systems, version 5.5.5. For MapReduce on this MPI platform, we used the MapReduce-MPI library [76, 77]; ii) the Magellan Hadoop cluster at the National Energy Research Scientific Computing Center (NERSC). The cluster has 78 nodes with a total of 624 cores dedicated for Hadoop, where each node has 2 quad cores Intel Nehalem 2.67 GHz processors and 24 GB DDR3 1333 MHz RAM. These nodes run Cloudera’s distribution for Hadoop 0.20.2+228.
4.2 Qualitative results

4.2.1 Qualitative evaluation of the Standard Shingling heuristic

We present here the results of our analysis on three real world networks. The three inputs represent three different types of networks — CAMERA-100M and Pfam-A-11M are biological graphs but with two different connectivity footprints (higher and lower mean vertex degree, respectively) and graph topology (unipartite and bipartite, respectively); whereas cnr-2000 [22] is an example of a non-biological network.

CAMERA is a sequence homology graph constructed for a set of $1.28 \times 10^6$ amino acid sequences [91]. The sequence data were originally downloaded from the CAMERA [13] portal and represent a collection of open reading frames extracted from an ocean metagenomics project [94]. The homology graph, constructed using the pGraph method [91, 92], contains $1.28 \times 10^6$ vertices which correspond to the amino acid sequences and $100 \times 10^6$ edges which correspond to those pairs of sequences which are homologous.

Pfam-A-11M is a bipartite graph that we constructed after downloading the entire Pfam-A protein-domain database [31, 75]. This database contains a set of proteins and a set of domains contained in those proteins. Each protein contains only a small subset of domains, and a domain can be contained in multiple proteins. Therefore, this data automatically renders itself in the form of a bipartite graph, where edges are between a protein and a domain if that protein contains that domain. Applying our clustering method to this input would yield a novel way to characterize proteins into “families” based on multiple conserved domains. Pfam-A graph contains $8.42 \times 10^6$ vertices (=
8,407,839 proteins + 11,823 domains), and $11.41 \times 10^6$ edges.

cnr-2000 is a crawl of the Italian Cisco Network Registrar domain that consists of 325,557 vertices and $2.73 \times 10^6$ edges.

We use two measures as an indication of the clustering quality – the average density of clusters and cluster modularity. For modularity calculations, Newman’s formula [70] was used for unipartite graphs (CAMERA-100M and cnr-2000) and Murata’s formula [66] for bipartite inputs (Pfam-A-11M). The density of a cluster is defined as the ratio between the number of its intra-cluster edges and the theoretical maximum on the number of such edges. Note that the upperbound for the density and modularity is 1. It is also to be noted that while modularity is a standard measure to assess community detection algorithms, the Shingling heuristic which we implemented is a density driven method, although without any density cutoff.

We also compared the results obtained using our implementation of the standard Shingling heuristic against the Louvain method [8]. The method is sequential and is one of the most widely used community detection methods. For fairness of comparison, we ran both methods on the same platform.

Table 1 shows the results of our analysis. It can be observed that, with the exception of the clustering modularity for CAMERA-100M, the Louvain tool delivers consistently higher density and modularity values and faster runtimes. More specifically, the standard Shingling method tends to create clusters that are more inclusive and hence less dense than the clusters generated by the Louvain tool. Depending on the nature of input this may or may not be desirable. For instance, while for inputs such as cnr-2000, a higher density (and modularity) indicates stronger cohesiveness, for CAMERA-100M inclusion of peripheral vertices is likely to be more biologically relevant due to the evolutionarily
Table 1: A summary of clustering results obtained from the standard Shingling heuristic with input parameters $s = 3$ and $c = 100$ and the Louvain method. For larger values of $s$, the density improves only marginally with a more substantial decrease in modularity (e.g., for CAMERA-100M: setting $s = 5$ increases the density to 0.5843 while decreasing the modularity to 0.9179).

<table>
<thead>
<tr>
<th>Input graph</th>
<th>Density</th>
<th>Modularity</th>
<th>Time (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Shingling</td>
<td>Louvain</td>
<td>Shingling</td>
</tr>
<tr>
<td>CAMERA-100M</td>
<td>0.5540±0.2487</td>
<td>0.7955±0.2319</td>
<td>0.9459</td>
</tr>
<tr>
<td>cnr-2000</td>
<td>0.5457±0.3288</td>
<td>0.6134±0.3213</td>
<td>0.6615</td>
</tr>
<tr>
<td>Pfam-A-11M</td>
<td>0.9552±0.1211</td>
<td>0.9932±0.0463</td>
<td>0.3858</td>
</tr>
</tbody>
</table>

diverse composition in metagenomic communities and low sampling rates. Although these results are presented with a few standard measures such as density and modularity, the appropriateness of a clustering tool to a particular application can be best judged by comparing against curated benchmarks. Without such an evaluation, it is difficult to assess the efficacy of individual clustering methods. That being said, based on our observations we posit that for traditional networks where community structure is sought after, a more traditional community detection tool such as the Louvain method could be better suited. A clustering heuristic such as Shingling could be better suited for graph inputs where it becomes a challenge to include nodes at the periphery of a cluster to accommodate low sampling rates and data diversity [67, 84]. The latter has been corroborated in our earlier studies with the heuristic [90, 91], where the method delivered better results than known clustering results.
4.2.2 Qualitative evaluation of the Normalized Shingling heuristic

We used a metagenomics sequence homology graph for testing our methods (see Table 2). This graph contains open reading frame/amino acid sequences obtained from a variety of environmental microbial community sources, originally constructed in a database by a collaborator’s group at University of British Columbia [38] (henceforth, referred as the “UBC data set”). Homology detection was performed on this set using pGraph [92] and the resulting graph contains a total of 10.3M vertices (sequences) and 640M edges (homologous pairs). For edge weight calculations, we used the degree of sequence similarity computed through Smith-Waterman alignments. Clustering this graph would yield core members of protein families [91, 94]. Smaller subgraphs were extracted from this large graph. A value of $s = 2$ and $c = 100$ was used in all our experiments, based on our earlier experiments with pClust [90, 91].

For the quality analysis of our new Shingling heuristic variant we also used a combination metrics — such as cluster modularity, the average density of clusters (which is agnostic to edge weights), and other clustering statistics such as the number of clusters, number of singletons and number of vertices in the largest clusters — as indicators

<table>
<thead>
<tr>
<th>Input label</th>
<th>Number of edges ($m$)</th>
<th>Number of vertices ($n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UBC-25M</td>
<td>$25 \times 10^6$</td>
<td>$3,965 \times 10^3$</td>
</tr>
<tr>
<td>UBC-50M</td>
<td>$50 \times 10^6$</td>
<td>$4,525 \times 10^3$</td>
</tr>
<tr>
<td>UBC-100M</td>
<td>$100 \times 10^6$</td>
<td>$6,795 \times 10^3$</td>
</tr>
<tr>
<td>UBC-200M</td>
<td>$200 \times 10^6$</td>
<td>$7,336 \times 10^3$</td>
</tr>
<tr>
<td>UBC-400M</td>
<td>$400 \times 10^6$</td>
<td>$8,958 \times 10^3$</td>
</tr>
<tr>
<td>UBC-640M</td>
<td>$640 \times 10^6$</td>
<td>$10,346 \times 10^3$</td>
</tr>
</tbody>
</table>
of quality. For modularity calculation, we used Newman’s formula \cite{70} as shown in Equation 4.1, taking edge weights into account.

\[
Q = \frac{1}{2m} \sum (A_{ij} - k_ik_j/2m) \delta(c_i, c_j),
\]

(4.1)

where \(A_{ij}\) represents the weight of the corresponding edge (or 0 if there is no edge), \(k_i\) is the total weight of edges adjacent to vertex \(i\), \(k_j\) is the total weight of edges adjacent to vertex \(j\) and \(m\) is the total weight of all edges.

For qualitative evaluation, we compared the quality of the clusters (by the above metrics) generated by the following methods: A) the normalized implementation of the Shingling heuristic, B) our previous implementation of the standard Shingling heuristic, C) the Louvain method \cite{8}, which is one of the most widely used (sequential) methods for detecting communities based on the modularity-maximization procedure, D) the Markov Cluster Algorithm (MCL), an unsupervised cluster algorithm for graphs, based on simulation of (stochastic) flow in graphs \cite{29}, E) the Multi-Level Regularized MCL algorithm (MLR-MC) \cite{82}. We also compared the quality difference between the unweighted and weighted inputs of the same graph. All analysis was performed on two subgraphs from the UBC data set — containing 25M and 100M edges, and using shingling parameters \(s = 2\) and \(c = 100\). Also we compared the three methods, that have shown best experimental results on the 25M and 100M edges subsets, for analyzing the entire UBC data set — containing 640M edges - A) the normalized implementation of the Shingling heuristic, B) the Louvain method and C) the MCL method.

Table 3 shows the results for analyzing the 25M and 100M data sets. A comparison of the results from our standard vs. normalized implementations of the Shingling
heuristic shows the effectiveness of our normalized heuristic in detecting significantly denser clusters than the standard heuristic. It can also be observed that the normalized heuristic significantly improves modularity over the standard heuristic for the 25M data set (0.84 to 0.99). Given that the Shingling heuristic is a density-driven approach, it is important to note that all the modularity figures reported are directly comparable to that of the Louvain method, which is solely a modularity-driven approach.

The results also show that normalization is effective in reducing the number of singletons drastically relative to the standard heuristic (35% to ∼2% for 25M, ∼28% to <1% for 100M) without compromising on the clustering quality. Reducing singletons is necessary for reducing the loss of sequence information. However, the main challenge is to recruit more sequences into clusters without adversely affecting the overall quality of clustering, especially cluster density. To this effect, it is noteworthy that our normalized heuristic achieves this goal by significantly increasing the cluster density.

Even though the Louvain method achieves a comparable or slightly higher modularity than our implementations, it is interesting to note that this improvement comes with an increase in the total number of non-singleton clusters. This shows that the modularity driven codes could detect large number of small clusters in the interest of increasing the modularity. However, from a clustering point of view, it is desirable to keep the number of output clusters smaller in order to reduce the burden on the downstream processing of clusters. It is also noteworthy that our normalized implementation is able to increase the size of the largest cluster by factors between 2x to 3x compared to the Louvain method, and doing so by still maintaining a high cluster density.

Finally, Table 3 shows the effect of incorporating edge weight information into the normalized heuristic method is minimal (please compare Unweighted, Normalized vs.
Weighted, Normalized). More specifically, adding edge weight information results only in a marginal improvement in modularity and number of singletons recruited, while maintaining density. This effect, however, is input dependent. For this particular sequence homology graph, the graph construction procedure retained only those edges corresponding to a strong similarity based on a predefined cutoff. We expect the edge weight information to play a more significant role in clustering decisions for inputs where there is a wider divergence in the quality of edges.

Tables 4 shows the results for analyzing the 640M data sets. The normalized heuristic consistent in producing fewer clusters with larger number of elements while maintaining a high cluster density. Further composition validation shows high quality of clustering (see section 4.3).

4.3 Metrics of cluster annotation quality

Here we explain the concepts of cluster and annotation in the context of our current analysis. A cluster $C$ is a group of protein or nucleotide sequences that are homologous to one another. Each member sequence $m$ in $C$ is associated with an annotation, a descriptive string, which we denote by $a_m$. For a sequence in a reference protein database, such as, COG [88], KEGG [45], MetaCyc [48], SEED [74], the annotation usually describes the functional, and often taxonomic, information about the the protein sequence. The functional part might be described in terms of an enzyme name or function, EC numbers, etc., and the taxonomic component many be the name of a taxon or species. These annotations are created in a number of ways, ranging from manual to automated identification and curation. However, for the purpose of our analysis, and for these
Table 3: Qualitative results of the clustering obtained from our different implementations, the Louvain, MCL and MLR-MCL methods for two subsets of the UBC data set.

<table>
<thead>
<tr>
<th>Input</th>
<th>Data set, Algorithm</th>
<th>Quality metrics</th>
<th>Clustering statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Modularity</td>
<td>Density</td>
</tr>
<tr>
<td>UBC-25M</td>
<td>Unweighted, Standard</td>
<td>0.8454</td>
<td>0.3126±0.2506</td>
</tr>
<tr>
<td></td>
<td>Unweighted, Normalized</td>
<td>0.9928</td>
<td><strong>0.5603±0.3866</strong></td>
</tr>
<tr>
<td></td>
<td>Weighted, Normalized</td>
<td><strong>0.9937</strong></td>
<td><strong>0.5603±0.3866</strong></td>
</tr>
<tr>
<td></td>
<td>Weighted, Louvain</td>
<td>0.9695</td>
<td>0.5309±0.3864</td>
</tr>
<tr>
<td></td>
<td>Weighted, MCL</td>
<td>0.9383</td>
<td>0.5259±0.3837</td>
</tr>
<tr>
<td></td>
<td>Weighted, MLR-MCL</td>
<td>0.8335</td>
<td>0.0304±0.1280</td>
</tr>
<tr>
<td>UBC-100M</td>
<td>Unweighted, Standard</td>
<td>0.9299</td>
<td>0.6715±0.2526</td>
</tr>
<tr>
<td></td>
<td>Unweighted, Normalized</td>
<td>0.8839</td>
<td><strong>0.8658±0.2466</strong></td>
</tr>
<tr>
<td></td>
<td>Weighted, Normalized</td>
<td>0.8988</td>
<td><strong>0.8658±0.2466</strong></td>
</tr>
<tr>
<td></td>
<td>Weighted, Louvain</td>
<td><strong>0.9628</strong></td>
<td>0.8634±0.2505</td>
</tr>
<tr>
<td></td>
<td>Weighted, MCL</td>
<td>0.9389</td>
<td>0.8584±0.2514</td>
</tr>
<tr>
<td></td>
<td>Weighted, MLR-MCL</td>
<td>0.8589</td>
<td>0.8414±0.2875</td>
</tr>
</tbody>
</table>
Table 4: Qualitative results of the clustering obtained from our Normalized implementation, the Louvain and MCL methods for the UBC data set.

<table>
<thead>
<tr>
<th>Input</th>
<th>Data set, Algorithm</th>
<th>Quality metrics</th>
<th>Clustering statistics</th>
<th>Largest cluster size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Modularity</td>
<td>Density</td>
<td># non-singleton clusters</td>
</tr>
<tr>
<td>UBC-640M</td>
<td>Weighted, Normalized</td>
<td>0.4763</td>
<td>0.9651±0.1202</td>
<td>1,254,268</td>
</tr>
<tr>
<td></td>
<td>Weighted, Louvain</td>
<td>0.9719</td>
<td>0.9773±0.0817</td>
<td>1,437,238</td>
</tr>
<tr>
<td></td>
<td>Weighted, MCL</td>
<td>0.9478</td>
<td>0.9759±0.0791</td>
<td>1,441,384</td>
</tr>
</tbody>
</table>
reference sequences we will not question the validity of the methods used in generating the annotations. The annotations of two highly homologous sequences are usually expected to have similar or related function. Due to very diverse sources of these reference databases, the annotations are not always in standard format. From a straightforward string comparison the annotations of two functionally similar proteins may appear different, although a human reader is very likely to perceive it correctly. This confusion comes from use of order of the words in the annotation, use of synonyms, additional information, word repeats, presence of stop words, etc. If we assume that the member sequences in a cluster are functionally similar, it is desirable to have an annotation associated with each cluster that summarizes or captures the common functionality.

For experimental validation of our clustering of protein sequences, we analyzed the clustering results of the UBC-640M dataset. The analysis was performed in collaboration with Dr. Kishori Konwar and Dr. Steven J. Hallam from University of British Columbia. The validation methodology was also developed by their group.

For the validation of clustering we consider only the clusters with at least two members. The member sequences of the cluster are expected to have similar functions. Therefore, a simple way to validate the quality of clustering would be to compare the annotations corresponding to the members of each cluster. Unfortunately, the number of clusters in a protein sequence database can be tens of millions (and many are expected to soon increase by an order of magnitude), an algorithmic method seems to be the only feasible way of evaluating their quality and assigning annotations. Here we formulate two metrics motivated by information theoretic concepts of entropy. These metrics can be programmatically computed for each cluster annotation to reflect its quality. Thinking along the lines of information theory, we would expect a cluster with a highly variable
set of member annotations to have high *entropy*, which is described later.

As an illustrative example, consider the four annotations below:

1. sensory transduction histidine kinase
2. sensory transduction histidine kinase
3. sensory transduction for histidine kinase ;
4. sensory transduction histidine kinase ;

transduction histidine

Observe that the annotations (1), (2), (3) and (4) contain the same functional information. However, annotations (1) and (2) are similar word for word; annotation (3) differs from those of (1) and (2) by the stop word for; and annotation (4) has the same set of words as the first two but contains repeated words.

In order to measure the uniformity of the annotations of the members sequences of a cluster, first we preprocess the annotations, where we remove stop words, punctuations and the repeats of a word. Repeated words in an annotation are recorded only once. Let us denote the set of initial annotations for the members of a cluster \( C \equiv \{m_1, m_2, \ldots, m_k\} \), of size \( k \), as \( A_C \equiv \{a_1, a_2, \ldots, a_k\} \). After an annotation \( a_i \) has been processed, we get a set of words \( \mathbf{a}_i \). The set \( W \) consists of words that have appeared in some member of \( C \), i.e. \( W \equiv \bigcup_{i=1}^{k} \mathbf{a}_i \). We now define two metrics, \( Q_p \) and \( Q_e \), for assessing the quality of annotation within a cluster \( C \).

**Definition 4.1 Quality score \( Q_p \):** For a cluster \( C \) we define the quality score \( Q_p \) as

\[
4 \frac{1}{|W|} \sum_{w \in W} f_w(1 - f_w)
\]
where

\[ f_w = \frac{\{a : a \in A, s.t. w \in W_a\}}{|W|}. \]

\( f_w \) represents the fraction of members in cluster \( C \) that contain word \( w \) in their annotations after filtering, or alternatively, the estimated probability of a word appearing in a member of \( C \) picked at random.

The metric \( Q_p \) has a strong similarity to the Gini index, often used for assessing the quality of a decision node based on the \textit{a priori} classification of the training set. Note that for an individual word \( w \), if the word appears in every set \( a \), of every member then \( f_w(1 - f_w) \) becomes 0. However, for the case where the word only appears only half of the \( a \) sets, we get \( f_w(1 - f_w) = 0.25 \). Therefore, in order to define \( Q_p \) in the interval \([0, 1]\), where 0 corresponds to perfect uniformity of the annotations and 1 the worst we have 1, we have the factor \( \frac{4}{|W|} \), to normalize to a per word score.

\textbf{Definition 4.2 Quality score} \( Q_e \): For a cluster \( C \) we define the quality score \( Q_e \) as

\[ -\frac{1}{|W|} \log 2 \sum_{w \in W} (-f_w \log f_w - (1 - f_w) \log (1 - f_w)) \]

where \( f_w \) is defined above.

The quality score \( Q_e \) is essentially the average of the entropies of the individual words in \( W \). Here we define the entropy of a word \( w \) in the cluster annotations as \( f_w \log f_w + (1 - f_w) \log (1 - f_w) \). For simplicity, we assume words in annotations are independent. This definition corresponds with the popular \textit{binary entropy function}, as there are two possible states (either a word appears in an annotation or does not). As such, general definition of entropy \[83\] \( \sum_i p_i \log p_i \) reduces to \( p \log p + (1 - p) \log (1 - p) \).
Note that the normalizing constant \( \frac{1}{\log 2} \) comes from \( x = \frac{1}{2} \), where \( f(x) = -x \log x - (1 - x) \log (1 - x) \) is maximized. This can be observed from \( \frac{df(x)}{dx} = -\log x + \log (1 - x) \equiv 0 \), and \( \frac{d^2f}{dx^2} |_{x=\frac{1}{2}} < 0 \) and note that \( f\left(\frac{1}{2}\right) = \frac{1}{\log 2} \). Therefore, in order to map the value of \( Q_e \) in the range \([0, 1]\) we have use the normalizing factor \( \frac{1}{\log n} \). Observe that clusters whose members contain almost the same set of words (after filtering) will have for \( Q_e \), a value closer to 0. Below we provide scatter plots, Fig. 5 and 6, with the vertical scale showing number of protein clusters and the horizontal axis the score for \( Q_e \) and \( Q_p \), respectively. Note that in both cases most of the clusters have values close to 0, showing a high degree of uniformity among the annotations coming from the members of a cluster.

Figure 5: Plot for number of protein clusters vs. \( Q_e \) score.
4.4 Performance results

4.4.1 Performance of the Hadoop-based implementation

We studied the performance of our MapReduce Hadoop implementation, on the Magellan cluster using the Pfam-A-11M input graph and its subgraphs as an input (see Table 5). The numbers of map tasks and reduce tasks are specified at input. Table 6 shows the phase-wise breakdown of the total runtime as a function of the number of map tasks. It can be observed that Shingling Phase I is the dominant phase for this input. This is because the transformed graphs ($G_I$ and $G_{II}$) become orders of magnitude smaller due to the highly sparse nature of the original input graph. Table 6 also shows that the Hadoop implementation scales linearly up to 64 map tasks for Shingling Phase I.
The lack of scaling for the remaining phases can be attributed to the small sizes of their corresponding input graphs.

While the code exhibits linear scaling at least for Shingling Phase I, the runtimes are substantially higher than the runtimes obtained using the shared memory implementation for the Pfam-A-11M input. For instance, the shared memory implementation took only 192 seconds on 8 cores of the commodity node, while Hadoop implementation took 8,587 seconds on 16 cores of the Hadoop cluster (implying a slowdown factor of roughly $\sim 80x$). The larger runtime on the Hadoop cluster is expected because of two factors: i) the I/O bound nature of the Hadoop environment; and ii) the algorithmic level improvements of the shared memory implementation (such as use of hash tables and combining the last two phases) could not be carried over to the distributed memory setting of Hadoop. To quantify these factors, we performed more diagnostics and found the following: Of the 3,599 seconds for Shingling Phase I consumed by the Hadoop implementation on the Pfam-A-11M input using 32 cores of the Hadoop cluster, nearly all of it (95.71%) was consumed by the longest running map task\(^1\) (implying a negligible fraction of time needed for the reducers to complete). And the average breakdown for

\(^1\)While the mappers varied in their respective runtimes, we did not identify any compelling evidence for a nonuniform workload distribution. The mean completion time for an average mapper was more than 50 minutes.

Table 5: Input graph statistics: The first part of the table shows statistics for input graphs constructed from UBC data. The second part shows the statistics for the graphs constructed from the Pfam-A protein-domain database.

<table>
<thead>
<tr>
<th>Input label</th>
<th>Number of edges (m)</th>
<th>Number of vertices (n)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#proteins</td>
<td># domains</td>
</tr>
<tr>
<td>Pfam-A-11M</td>
<td>$11 \times 10^6$</td>
<td>8,407,839</td>
<td>11,823</td>
</tr>
<tr>
<td>Pfam-A-8M</td>
<td>$8 \times 10^6$</td>
<td>6,639,087</td>
<td>3,537</td>
</tr>
<tr>
<td>Pfam-A-4M</td>
<td>$4 \times 10^6$</td>
<td>3,681,067</td>
<td>665</td>
</tr>
</tbody>
</table>
Table 6: Breakdown of runtime (in seconds) by phases for the input graph Pfam-A-11M.

<table>
<thead>
<tr>
<th>Phase</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shingling Phase I</td>
<td>6,874</td>
<td>3,599</td>
<td>1,701</td>
<td>1,241</td>
</tr>
<tr>
<td>Unique I</td>
<td>26</td>
<td>34</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>Shingling Phase II</td>
<td>982</td>
<td>797</td>
<td>682</td>
<td>319</td>
</tr>
<tr>
<td>Unique II</td>
<td>26</td>
<td>28</td>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>Connected component</td>
<td>679</td>
<td>733</td>
<td>705</td>
<td>763</td>
</tr>
<tr>
<td>detection</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>8,587</td>
<td>5,191</td>
<td>3,137</td>
<td>2,375</td>
</tr>
</tbody>
</table>

the map task runtime is as follows: 31.34% in local calculations, 47.76% for the grouping of mappers’ intermediate key-value pairs, and the remaining 20.90% towards I/O. It is to be noted that even the grouping task that the Hadoop framework carries out is performed in I/O (HDFS). These diagnostics provide a clear picture of the different overheads associated with the Hadoop framework on the clustering application.

In what follows, we present the results of analyzing the scalability of the Hadoop implementation for its time-dominant phase, which is Shingling Phase I. Table 7 shows Shingling Phase I’s run-time as a function of the number of map tasks. The results show linear scaling up to 32 cores for the smaller inputs Pfam-A-4M and Pfam-A-8M, and that the linear scaling behavior extends to 64 cores for the larger input (Pfam-A-11M). Figure 7, which shows the relative speedup calculated using the 16 tasks run as the reference, confirms this scaling trend. A peak speedup of ~90x is obtained for the 11M input running using 128 map tasks.

**Task granularity study:** MapReduce framework supports a feature whereby one could run an arbitrary number of map and reduce tasks, independent of the number of cores. This can be used to determine task granularity that is empirically optimal. To study
Table 7: The run-time (in seconds) for the Shingling Phase I as a function of input and system sizes on the Magellan Hadoop cluster. All runs were performed using $s = 2$ and $c = 100$.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_m=16, p_r=11$</td>
<td>2,188</td>
<td>4,397</td>
<td>6,874</td>
<td></td>
</tr>
<tr>
<td>$p_m=32, p_r=11$</td>
<td>985</td>
<td>1,962</td>
<td>3,599</td>
<td></td>
</tr>
<tr>
<td>$p_m=64, p_r=11$</td>
<td>684</td>
<td>1,223</td>
<td>1,701</td>
<td></td>
</tr>
<tr>
<td>$p_m=128, p_r=37$</td>
<td>531</td>
<td>1,740</td>
<td>1,241</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: Speedup of the Shingling Phase I.

this effect, we varied the number of map tasks from 64 up to 16K. The results are shown in Figure 8 for the Pfam-A-11M graph. Note that in our experimental Hadoop cluster, there is a system-imposed cap of 480 cores that can be used for map tasks. It can be observed that the run-time decreases gradually until 2K map tasks and then starts to increase again. The reason for the initial decline in run-time is because with
Figure 8: Effect of varying the number of map tasks on Shingling Phase I’s runtime for the Pfam-A-11M input. It can be observed that the optimal setting for the number of map tasks is 2K for this input.

increased number of map tasks, reducers are able to start earlier. However that benefit is soon offset by the increase in system overhead that is required to manage an increasing number of map tasks. For this set, we find that the empirical optimal value for task granularity is occurring approximately at 5,500 edges per map task ($\approx \frac{11M}{2K}$).

Reducing space-complexity at the reducers: Each reducer task stores an $O(c \times s)$ matrix to keep track of the running minimum $s$ elements generated for each random trial corresponding to a given source vertex $u$. This space complexity can be further reduced to $O(s)$. However, observe that there is no interdependency among elements inserted across different random trials. Therefore, an alternative is to dedicate a reducer task to every unique combination of $< u, j >$, where $j$ is the random trial id. This would reduce the space complexity for each reducer task to $O(s)$. We implemented this alternative
version and upon experimenting found that this space improvement comes at the cost of increased runtime — for instance, on the same input and on the same number of cores, reducers took 20 m 6 s using our old configuration, whereas 33 m 5 s using the new modification. The increase in runtime is probably due to an increased overhead in handling more reducer tasks by the Hadoop framework. It is noteworthy that owing to fact that both $c$ and $s$ are relatively small constants (e.g., $c \approx 100$, $s \leq 10$) in practice, retaining the original space complexity of $O(c \times s)$ per reducer task is not prohibitive.

### 4.4.2 Performance of the MPI-based implementation

We studied the performance of our MapReduce-MPI implementation, on the Hopper cluster using the UBC input graph and its subgraphs as an input (see Table 2). To enable comparison between two platforms – Hadoop and MPI, we evaluated the runtime results for our standard heuristic implementation. It is noteworthy that our new implementation built over the MapReduce-MPI library is significantly faster when compared to our implementation on Hadoop platforms. For instance, the analysis of a smaller input with only 11M edges took about 1,612 seconds on 128 cores of Hadoop cluster; whereas the same analysis on the same number of cores using our new implementation took only 41 seconds. We attribute the superior performance of the MapReduce-MPI implementation to its ability to perform in-core operations — i.e., if the intermediate temporal data owned by processor fits within an allocated pages of memory, the library operates on the data in-core with no disk files are written or read [76].

Tables 8 and 9 show the runtime of MPI implementation of the standard and normalized heuristics respectively as a function of the input and system sizes. As can
Table 8: Run-time (in seconds) for our MPI implementation as a function of the input and system sizes on the Hopper supercomputer for unweighted graphs.

<table>
<thead>
<tr>
<th>Input graph</th>
<th>Time using $p$ cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$p = 64$</td>
</tr>
<tr>
<td>UBC-25M</td>
<td>104.4</td>
</tr>
<tr>
<td>UBC-50M</td>
<td>159.87</td>
</tr>
<tr>
<td>UBC-100M</td>
<td>158.22</td>
</tr>
<tr>
<td>UBC-200M</td>
<td>110.57</td>
</tr>
<tr>
<td>UBC-400M</td>
<td></td>
</tr>
<tr>
<td>UBC-640M</td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Run-time (in seconds) for our implementation as a function of the input and system sizes on the Hopper supercomputer for weighted graphs.

<table>
<thead>
<tr>
<th>Input graph</th>
<th>Time using $p$ cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$p = 64$</td>
</tr>
<tr>
<td>UBC-25M</td>
<td>821.02</td>
</tr>
<tr>
<td>UBC-50M</td>
<td>1282.32</td>
</tr>
<tr>
<td>UBC-100M</td>
<td>1381.44</td>
</tr>
<tr>
<td>UBC-200M</td>
<td>973.68</td>
</tr>
<tr>
<td>UBC-400M</td>
<td>905.89</td>
</tr>
<tr>
<td>UBC-640M</td>
<td>1022.54</td>
</tr>
</tbody>
</table>

be observed, in both cases the runtime decreases as more cores are added to the system. Also, the scaling improves as the input size increases. For the largest input size (640M) we observe near-linear scaling up to 4K cores. More notably, even for the smallest data set (25M) the implementations deliver considerable performance improvement until 512 cores, beyond which the work becomes too small to benefit from parallelism. Also note that values along the diagonal of the table are roughly constant, which is a good indicator that when the input size is doubled alongside doubling the system size, the runtime is roughly maintained. These results demonstrate the overall scaling potential of our implementation to even larger number of cores as larger inputs are analyzed.
Table 10: Run-time (in seconds) for our normalized heuristic for varying system sizes on the weighted and unweighted inputs for UBC-25M.

<table>
<thead>
<tr>
<th># cores</th>
<th>Unweighted, Normalized</th>
<th>Time</th>
<th>Weighted, Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>450.55</td>
<td>458.73</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>237.96</td>
<td>249.42</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>126.87</td>
<td>132.22</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>73.64</td>
<td>78.81</td>
<td></td>
</tr>
</tbody>
</table>

Effect of adding edge weights on performance:

We also evaluated the performance of our normalized parallel implementation separately on both weighted and unweighted inputs. Table 10 shows the run-times for both inputs for varying number of cores. Note that the implementation scales near-linearly up to the system size of 1,024 cores tested on both weighted and unweighted inputs. Furthermore, as to be expected from the effect of normalization, the runtimes for the weighted and unweighted inputs were nearly identical. The marginal increase in the run-times for the weighted inputs can be attributed to the increased number of floating point operations performed during the summation of edge weights prior to normalization.

Adjacency list vs. Edge list evaluation: All the tests that were run in our MPI experiments are using our parallel implementation that assumes that the input graph is made available as an adjacency list. However, we also re-implemented the edge-list based algorithm for Hadoop systems using MapReduce-MPI, and subsequently compared their performance. Our results (see Table 11, 12) show that the adjacency list implementation was consistently twice as fast as the edge list implementation, and is also more memory efficient. More importantly, our results led us to conclude that the adjacency list based implementation is better suited for the MapReduce-MPI model.
where it is desirable to keep the volume of the intermediate <key,value> pairs as small as possible so as to enable in-core access.

### 4.5 Conclusion

We presented a new variant of the shingling heuristic for clustering weighted and unweighted biological graphs. This new algorithmic heuristic is designed to overcome the qualitative problems encountered in the standard version, and also extends its application to weighted graphs. When applied to real world metagenomics graphs, our implementation demonstrated significant improvements in quality than the previous version, and in performance under the MapReduce parallel model. From a practical standpoint, it is desirable for a clustering method to detect as fewer number of non-singleton clusters and singletons as possible, while maintaining a highly density within clusters. The newly proposed clustering heuristic using normalization has been demonstrated to be highly...
effective in all these respects.
Chapter 5

Cluster-Based Approach for Socio-Technical Coordination in Open-Source Communities

5.1 Introduction

In collaboration with Prof. Panchal, we also investigate clustering-based approaches and methodologies applied to networks from other scientific domain – more specifically, for measuring the socio-technical coordination within virtual (open source development) communities as observable through the cluster level linkages between unipartite and bipartite networks [81]. This work represents an application use-case for our clustering methodologies developed for both unipartite and bipartite networks.

5.1.1 Socio-technical coordination

Coordination within an organization can be defined as "integrating or linking together different parts of an organization to accomplish collective sets of tasks" [95], or "management of dependencies" [61, 62]. Various methods, including informal communication, group meetings, email and forum discussions, and development of plans and rules can be employed to achieve coordination within an organization. Three types of coordination approaches are defined by Herbsleb and Grinter [39] as:

1. architecture-based coordination,
2. plan-based coordination,

3. process-based coordination.

Architecture-based coordination is associated with product modularization, which is applied to reduce dependencies across modules in a product. Plan-based coordination involves the development and integration of plans and timelines to ensure the accomplishment of interdependent tasks. Process-based coordination is achieved through development and modification of management processes.

A widely adopted strategy to achieve coordination is the alignment of organizational structures with product structures. The coordination achieved through the alignment of organizational structures and product structures is referred to as socio-technical coordination [15]. Recently, a measure called socio-technical congruence has been proposed in the literature to quantify the extent of socio-technical coordination [15]. Existing studies in organization science indicates that the alignment between design of organizational structures and product structure is beneficial for product development. High level of socio-technical congruence has been shown positive influence in productivity and product development rate [15, 16], while a lack of coordination has negative effects on productivity [27, 85]. Besides, existing literature also suggests that organizational structure affects the structures of products developed by that organization. The effect is commonly known as the Conway’s law: ”any organization that designs a system will inevitably produce a design whose structure is a copy of the organization’s communication structure” [21]. Finally, it has been hypothesized that because the communication patterns between individuals are driven by the product dependencies, the social structure of organization matches the structure of product [20]. This hypothesis is known as
the mirroring hypothesis [59].

Our goal is to develop approaches for computing socio-technical coordination in virtual communities involved in developing open-source products. Understanding of the socio-technical coordination is important in open source communities because it helps in determining the impacts of product structures on community structures and vice versa. Furthermore, understanding of socio-technical coordination may help in increasing the success of open source projects, particularly, the open source hardware projects.

5.1.2 Related work

The literature on socio-technical coordination is associated with two concepts: mirroring hypothesis and socio-technical congruence. The mirroring hypothesis states that the technical dependencies and communication patterns are mirrored with one another in the product development processes [20]. Existing studies on mirroring hypothesis are focused on analyzing the correspondence between technical dependencies and communication patterns based on empirical case studies. Different levels of correspondence between technical dependencies and communication patterns are displayed in existing empirical studies. Colfer and Baldwin [20] present a detailed literature review by comparing 102 empirical studies ranging from different levels of organizations, including a single firm, across multiple firms and open communities. Based on the studies, the authors conclude that mirroring hypothesis is more prominent in single firms, less prominent in the studies across multiple firms, and are the least prominent in open communities. Camuffo and Cabigisu [14] and Hoetker [41] analyze the extent to which modular products are associated with modular organizations. Camuffo and Cabigiosu
[14] study the inter-organizational relationships between the manufacturers and suppliers based on an example of air conditioning industry. Kratzer et al. [51] analyze multi-institutional communication patterns in the space industry and conclude that the informal communications play a strong role in achieving coordination between different activities.

While mirroring hypothesis is focused on correspondence between technical and organization structure during the product development process, socio-technical coordination [40] is based on the assumption that the organizational structures should match the technical dependencies. Cataldo et al. [15, 16] introduced the concept of congruence as a measure for coordination activities carried out within an organization. The congruence measurement is based on the comparison of matrices representing coordination requirements and actual coordination [16]. Coordination requirements refer to coordination activities that should happen due to the technical dependencies of a product. The actual coordination is the set of coordination activities that individuals are engaged in through multiple communication methods such as face-to-face discussions, emails, and online forums. Sosa [86] extends the analysis of congruence by analyzing not only the coordination requirement that are met but also unpredicted and unintended integrations. The impacts of socio-technical congruence on performance are also highlighted in existing studies. Different measures of performance are used by different authors. For example, Cataldo et al. [16] employ resolution time as a measure of performance, and show that lower resolution time, namely better performance, is observed when the congruence are high. Kwan et al. [52] apply the success of software builds as a measure of performance and show the congruence is proportional to build success by collocated teams.
5.1.3 Research gap

The limitation of existing approaches for socio-technical coordination when applied to open source processes is that existing approaches compare the information exchange between teams, which are not well defined in loosely coupled open source communities. The congruence measures are calculated by analyzing the direct interactions between individuals. However, in open source processes, the number of individuals is generally very large. The values of socio-technical congruence are very small due to the fact that participants work on different aspects of the code. The values less than 1% have been observed in the studies of some open source software [52]. However, the small value of congruence does not necessary mean that the coordination is not taking place. As the number of participants increases, the coordination requirements grow at a faster rate than the actual coordination. Hence, the socio-technical congruence calculated according to direct communication actually reduces.

Additionally, the existence of dependencies between modules does not necessarily imply that the corresponding individuals must communicate. Coordination may also happen through

- shared knowledge within communities,
- prior communication on related dependencies.

Such mechanisms of coordination are not captured by the socio-technical congruence measure. Hence, the existing socio-technical congruence measure provides inadequate information about coordination in open source communities.

To address these limitations, we present an alternate approach to measuring coordination. The approach is based on identification of communities via clustering analysis
and comparison of ideal communities with expected communities. The underlying hypothesis is that if individuals have closer communications with each other, they are part of the same community, and they share some common knowledge. Because of this common knowledge, they do not need to communicate for each and every dependency. In the proposed approach, instead of comparing networks at the individual link level, the networks are first clustered into communities and then the resulting clusters are compared to measure the socio-technical coordination.

\section*{5.2 Proposed Approach}

\subsection*{5.2.1 Modeling products and communities as hybrid networks}

Let $V_p = p_1, p_2, p_m$ denote a set of open source software developers (referred to as people), and $V_f = f_1, f_2, f_n$ denote a set of source files (referred to as products). The input to our analysis contains three networks:

1. people network,
2. project network,

3. people-project bipartite network.

A combination of the three networks with the three types of links, as shown in Figure 9, is called the hybrid network. The files are represented as circles and the links are represented as arrows. Ideally, the communication between individuals and dependencies between files are directed in nature. However, we model them as undirected edges for initial analysis presented in this study. The approach will be extended to account for the directed nature of edges in the future.

1. People network $G_p(V_p, E_p)$: An edge $(p_i, p_j)$ in $E_p$ (undirected, weighted) represents the communication between developers $p_i$ and $p_j$, and the edge weight (an integer) represents the volume of that communication, i.e., the number of times two individuals communicate with each other. This is shown as links between individuals in Figure 9.

2. Product network $G_f(V_f, E_f)$: This network contains information about the product interdependency. An edge $(f_i, f_j)$ in $E_f$ (undirected, weighted) implies that there exists a dependency between two files $p_i$ and $p_j$, and the edge weight represents the degree of that dependency. Two files are considered dependent if there is at least one function call between the two files. The edge weights correspond to the number of function calls across the files. Edges $E_f$ are shown as arrows between circles (projects) in Figure 9.

3. People-product bipartite network $G_b(V_p, V_f, E_b)$: This is a bipartite network where each node represents either a developer or a source file, and an edge $(p_i, f_j)$ in
\(E_b\) (undirected, unweighted) represents the information that the developer \(p_i\) has contributed to the product \(f_j\). The edges are shown as dashed lines in Figure 9.

5.2.2 Clustering techniques for bipartite and 1-mode networks

Our first step is to individually cluster the bipartite network and two 1-mode networks. To perform clustering, there are many well known community detection methods that can be used (e.g., [8, 19, 78]). We use the Louvain method, [8], which is one of the more recent widely-used algorithms. This algorithm is an adaptation of one of Newman’s classical methods [19], and implements a greedy heuristic to optimize modularity [70] of clustering, which is a quality measure of the division of a network. Modularity measures the fraction of the edges in the network that connect vertices of the same type (i.e., within-community edges) minus the expected value of the same quantity in a network with the same community divisions but random connections between the vertices [70].

The main idea of the clustering algorithm is as follows: Starting with each element in a community for itself, the algorithm deploys an iterative approach to migrate elements into communities so as to optimize the increase in modularity at every stage. At the start of every iteration, all elements are visited in a sequential order, and for each element \(i\), a community is determined, which by recruiting \(i\) would result in the largest positive value increase of modularity. Note that this search needs to be restricted only to those communities with which \(i\) shares a neighbor and to those in which the change in modularity is positive. Furthermore, the value by which the modularity would change owing to the migration of a given element \(i\) to a give community can be calculated in constant time. The sequential order processing is repeated iteratively until there is no further
change in the community compositions. This completes the first phase. In the second phase, a new meta-network is constructed by representing each community resulting from the previous phase by one meta-node and connecting those pairs of meta-nodes which share at least one edge in common. The algorithm terminates when communities stop changing.

Although the Louvain method does not guarantee optimality of modularity, it is effective in producing high quality clusters in practice [8], and the tool is available as part of Gephi [33], which is a popular open source graph visualization and analysis toolkit.

Modularity of the output clusters were computed differently for the unipartite and bipartite networks. For unipartite networks, modularity was calculated as per the standard Newman’s formula [70], which calculates the fraction of intra-cluster edges obtained by the clustering and then subtracts the fraction as would be obtained in a random network with identical vertex degree distribution (to negate for the random factor). There are multiple ways to calculate modularity in bipartite networks [4, 36, 66]. We used the expression given by Murata [66] because it does not enforce a one-to-one mapping between clusters from both sides.

5.2.3 Cluster Analysis Methodology

In this section, we outline the steps carried out in analyzing the results of clustering. We present three levels of comparisons between the two individual networks (people and product) and the bipartite network. Each of these three comparative methods provides a different insight as explained below. The primary question driving our analysis is:
what is the extent of socio-technical coordination within open source communities?

Link–link comparison. The first way is based on the existing work by Cataldo et al. [15, 16] and Sosa [86]. As discussed earlier, the existing approach to measure socio-technical coordination is to calculate congruence [15, 16] based on coordination requirements and actual coordination activities among individuals. The coordination requirement network consists of communication links that should ideally be present between individuals in order to resolve product dependencies. While actual coordination records the communication that is actually happening among individuals. The matched interactions [86] are defined as the number of links that occur in both coordination requirement network and actual communication network. The ratio of matched interactions to the total number of links in the coordination requirement network is the socio-technical congruence [15, 16]. Sosa’s approach [86] is another approach for link-link comparison where, matched interactions are defined in the same way as congruence. The potential unattended interactions are defined as the links that present in the coordination requirements network but absent in the actual communication network. Finally, the links present in the actual communication network but absent in coordination requirements network are called unpredicted interactions. Both these approaches are based on direct comparison of the links between ideal communication network and the actual communication network. In the context of our open software community data, this can be achieved by deriving the ideal communication structure between individuals based on the bipartite network and the 1-mode product network, and comparing the ideal communication structure with the actual 1-mode people network. This approach assumes a strict definition of coordination and uses direct communication between individuals as an indication of coordination. This approach provides a way to quantify the number of
conserved as well as unique links at the finer-level of individual people and their connections. This does not provide any insights into the underlying community structure as it uses only the original input networks as input. Note that a similar approach can also be applied in the other direction comparing the ideal dependencies between products based on the communication between individuals with the actual dependencies. This can be used to check the validity of the Conway’s law for this project. However, we only focus on coordination in this study.

**Cluster–cluster comparison.** An orthogonal approach is to compare the communities detected within the people network against the communities detected from the other network. In the context of our analysis, this is achieved by generating clusters (aka. communities) for all three input networks and then comparing the clusters of each of the 1-mode (people, product) networks against the clusters obtained for the bipartite (people-project) network. The clusters obtained from the bipartite network indicate natural groupings of people and projects based on the commonality of tasks that people are working on. The individuals clustered together within the bipartite network are the ones who are working closely together on related tasks. Hence, it can be assumed that these individuals share common knowledge. On the other hand, the clusters of individuals in the 1-mode people network indicate that individuals are related with each other through direct or indirect communication. A higher overlap between clusters obtained from the 1-mode network and the clusters obtained from bipartite network indicates a higher extent of implicit coordination. Specifically, based on these clusters, we ask the question: what fraction of pairs of individual entities (pairs of developers or pairs of products) have been clustered together (or separated) consistently between the two clustering results (1-mode networks and bipartite network)? We implement this comparison
using a Rand Index calculation [79]. The Rand index can be expressed as:

$$R = \frac{(a + b)}{n^2}$$  \hspace{1cm} (5.1)

where $a$ is the number of pairs of nodes in the same cluster in both partitions being compared, $b$ is the number of pairs of nodes in different clusters in both partitions; $n$ is the total number of nodes. Other measurements for comparing clusters include the adjusted rand index [79], Jaccard index [43], the Wallace index [89], and the normalized Lerman index [57]. However, in this study, we only use the Rand index due to its simplicity of calculation and intuitiveness. The comparison of clusters could lead to a better understanding of the degrees of similarities and differences underlying the community structures of both networks (1-mode vs. bipartite). Yet the obtained information, which is a Rand index, is only a summary figure.

**Link–cluster comparison.** The link-link and cluster-cluster comparison approaches are two extremes where link-link approach uses a strict definition of coordination whereas the cluster-cluster comparison uses a broader and more encompassing view of coordination. The third approach is an intermediate approach where links in one network are compared against communities detected in another network. In the context of our analysis, we implement this approach by comparing the connections implied/induced by the bipartite network clustering among the people (or alternatively, products) against the observed connections in the corresponding input network of people (or alternatively, products). For instance, let $(p_i, p_j)$ and $(p_i, p_k)$ be two edges present in the input people network, such that $p_i$ and $p_j$ are clustered together in the bipartite network but $p_i$ and $p_k$ are not. This implies that $p_i$ and $p_j$ are both communicating with one another and are
expectedly grouped together in product space; whereas \( p_i \) and \( p_k \) are communicating and yet their product affiliations do not offer sufficient basis for them to be grouped together in the bipartite setup. Henceforth, for sake of convention, we will refer to edges such as \((p_i, p_j)\) as *conserved* edges, and all edges such as \((p_i, p_k)\) as *discrepant* edges. The same analysis when applied to the product network has a different meaning. Two products \( f_i \) and \( f_j \) which are both dependent (i.e., share an edge in the product network) and also get clustered together in the bipartite network implies that there is a group comprising of one or more developers who are jointly contributing to these two interdependent products. However, if there are two products \( f_i \) and \( f_k \) which are inter-dependent in the product network but not clustered together in the bipartite network that indicates a potential gap (or a disconnect) in communication among those products contributors. In other words, this approach of comparing links to clusters to determine conserved and discrepant edges can serve as an effective way to flag anomalies as well as formulate hypothesis about the level of organization and communication in an open source software community. An alternative albeit less interesting outcome of the analysis is perhaps that the quality of the output clusters is not adequate, which in itself could be another layer of useful information.

### 5.3 Case study: DRUPAL

In this section, we present the approach using an open source software project Drupal [26]. Drupal is a content-management system used for the creation of community-based websites. Drupal has been under development since 2001. We analyze a recent major version of Drupal core (version 7.7). Drupal is well developed with over 7000
community-contributed add-ons, known as contrib modules. Besides, over 1000 developers are involved in the project development. Drupal is selected as a case study because of its maturity, availability of code and the availability of information about communication between participants.

5.3.1 Data collection

The software structure of Drupal is modeled as a weighted network where nodes represent files and the links represent function-calls between files. A documentation generator tool, Doxygen [25], is used to extract functions and corresponding function calls from the source code. The strength of the interface between files is defined by the number of function calls between two files. The community structure of Drupal is also modeled as a weighted network. The communications between participants are derived from on-line forums. The communication links between participants are determined by analyzing each post on the forum. A post on the forum contains information about the names of participants and the software version they are discussing about. Relationships are built among participants discussing on the same post. The participation links are created by analyzing file modifications and issue/patch records, recorded on the Drupal website. A file modification indicates which files are modified to resolve a specific issue. An issue/patch record indicates who worked on the specific issue/patch. Combining the information derived from file modifications and issue/patch records, the participation links representing participants working on files are generated. The generation of the networks with the three types of links is illustrated in Figure 9. The basic statistics of these networks is presented in Table 13. Isolated nodes have been eliminated for
clustering purposes. Hence, the number of nodes in the table is based on the nodes with at least one edge. The 1-mode people network contains 5,180 nodes with 148,102 edges. The 1-mode product network contains 167 source files which are linked through function calls. The files that do not have function calls (e.g., image files, .css files, etc.) are isolates and are not captured in the 1-mode product network. The bipartite network captures all files linked to at-least one individual. The number of nodes in the bipartite network is 6,572 and the number of edges is 95,988. The edges in the bipartite network only represent affiliation relationship. Hence, these edges are not weighted.

### 5.3.2 Summary of Network Clusters

We cluster the three input networks separately using the methodology described earlier. The results are summarized in Table 14. The table shows the modularity of clustering achieved for each of the three networks. In practice, a modularity of 0.3 or more is generally considered indicative of a well-defined community structure in unipartite networks [19]. As can be observed the modularity is higher for the people network (0.307) than it is for the other networks, indicating a more well-defined organization of the software developers into communities. This observation is further illustrated by
Figure 10: Clustering results: Cluster graphs showing the inter-community relationships for the a) people network, b) project network, and c) bipartite network. Each node in the cluster graph corresponds to one community/cluster that is detected in the underlying network, and an edge between two cluster nodes corresponds to the presence of (inter-cluster) edges between any two vertices of those clusters. The thickness of an edge is a relative indication of the number of inter-cluster edges between the two clusters.

Figure 10, which shows a lower degree of inter-cluster connectivity across communities in the people network (i.e., more modular) than it is for the other networks. On the other hand, the 1-mode product appears to have a lower modularity ($< 0.3$; also see Figure 11b), implying a weaker organization as communities.

Table 14 also shows the average number of edges that ended up within clusters (intra-cluster: $K_1$) and between pairs of clusters (inter-cluster: $K_2$). A larger modularity would result in a higher ratio for $K_1 : K_2$. And as can be observed this ratio is highest (6.218:1) for the people network. Put another way, there are over 6 discussions happening between people of the same community, for every discussion that involves people from two different communities.

Figure 11 shows the cluster size distribution for the three networks. In the clustering for the people network, the top two largest clusters accounted for almost 60% of the nodes, with the remaining 13 clusters significantly smaller. This skewed cluster size distribution coupled with the high modularity of clustering suggests the high degree of communication that characterizes people within the same cluster. As for other two
Table 14: Summary of clustering results

<table>
<thead>
<tr>
<th>Input network</th>
<th>Modularity (Q)</th>
<th># clusters</th>
<th>Average # intra-cluster edges per cluster ($K_1$)</th>
<th>Average # inter-cluster edges between any two clusters ($K_2$)</th>
<th>Ratio $K_1 : K_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>People (1 mode)</td>
<td>0.307</td>
<td>15</td>
<td>4,644</td>
<td>747</td>
<td>6.218:1</td>
</tr>
<tr>
<td>Product (1-mode)</td>
<td>0.208</td>
<td>7</td>
<td>46</td>
<td>27.4</td>
<td>1.678:1</td>
</tr>
<tr>
<td>Bipartite</td>
<td>0.334</td>
<td>9</td>
<td>4,887</td>
<td>1,444</td>
<td>3.384:1</td>
</tr>
</tbody>
</table>
networks (1-mode project network and the bipartite network), the cluster size distribution is more uniform, implying a reduced degree of community organization (as also corroborated by the smaller values of modularity).

5.3.3 Link–Link comparison

The value of congruence measure and Sosa’s approach for Drupal version 7.7 are shown in Table 15. From this table, it is observed that congruence value is 0.0099 (i.e., 0.99%), which indicates that less than 1% of the expected interactions between people are actually present in the 1-mode people network. The ideal congruence value is 1, which implies that people are communicating corresponding to all the technical dependencies. Clearly, 0.99% is a very low congruence value. Besides, the number of unpredicted and potential unattended interactions is significantly higher than matched interactions, which means the socio-technical coordination through direct communication is not a significant part in Drupal development.

Low values of socio-technical congruence have been reported in other studies on
Table 15: Link–Link Comparison for People vs. Product networks

<table>
<thead>
<tr>
<th>Version</th>
<th>Congruence</th>
<th>Matched interactions</th>
<th>Unpredicted interactions</th>
<th>Potential unattended</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.0178</td>
<td>12,555</td>
<td>2,956</td>
<td>693,699</td>
</tr>
<tr>
<td>5</td>
<td>0.014</td>
<td>21,008</td>
<td>7,648</td>
<td>1,478,563</td>
</tr>
<tr>
<td>6</td>
<td>0.0117</td>
<td>72,160</td>
<td>17,581</td>
<td>6,119,698</td>
</tr>
<tr>
<td>7</td>
<td>0.0099</td>
<td>127,983</td>
<td>20,119</td>
<td>12,782,290</td>
</tr>
</tbody>
</table>

open-source software. For example, Kwan et al. [52] observed that over 75% of software builds in an IBM project involving distributed developers have congruence value of less than 25%. Even for software builds with a congruence of 0%, the authors in [52] observed a build success of 93%. Low values of socio-technical congruence do not necessary mean a lack of coordination since coordination happens through implicit communication also. For example, by sharing workspace, individuals can get information about addressing dependencies between technical aspects. Bolici and co-authors [9] argue that within open source community, the coordination primarily happens via indirect communication, which is referred as stigmergic coordination. Some of these limitations of the congruence measure can be addressed by assuming that individuals who are within a community (cluster) share common knowledge and indirectly communicate with other individuals.

One way to identify the indirect communication among individuals is through community identification. In this approach, people are clustered into communities, indicating that there are strong interactions between individuals within the community and weak interactions between communities. Instead of studying links between individuals, we assume that individuals belong to the same cluster communication with one another by direct or indirect communication.
Table 16: Cluster–cluster comparison: Rand indices for comparing the individual people and product communities against the communities detected in the bipartite network. A perfect agreement between the two community sets implies a Rand index of 1.

<table>
<thead>
<tr>
<th>Rand Index</th>
<th>People clustering vs. Bipartite Clustering</th>
<th>People clustering vs. Bipartite Clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.718</td>
<td>0.736</td>
</tr>
</tbody>
</table>

5.3.4 Cluster–Cluster comparison

We compare the clusters of the 1-mode networks against the clusters of the bipartite network and report the Rand index for those comparisons in Table 16. As can be observed the Rand index for both comparisons are above 70%, which is a strong indication of overlap at the cluster-to-cluster level. In other words, the probability that two nodes are clustered the same way (either together or separate) in both the 1-mode network and the bipartite network is above 0.7. Note that Rand index does not take into account the actual node-level connections. It only captures the information of cluster memberships for every pair of nodes.

5.3.5 Link–Cluster comparison

We compared the direct links (i.e., edges) of each of the 1-mode networks to the bipartite clustering (see Table 17).

Two sets of results are presented. First, the 1-mode network edges are categorized into conserved and discrepant categories. Each conserved edge is one that is present in the 1-mode network as well as induced by the clustering of the bipartite network. An edge in the 1-mode network that is not induced by the bipartite clustering is referred to as discrepant. As can be observed, a vast majority of edges in both 1-mode networks
Table 17: Link–cluster analysis: Partitioning of the input (i.e., people or product) network edges into conserved and discrepant edge categories, based on their presence or absence within the same cluster of the bipartite graph, respectively.

<table>
<thead>
<tr>
<th>Input network</th>
<th>Partitioning of the 1-mode network edges by the bipartite clustering</th>
<th>Distribution of the 1-mode network edges among bipartite clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># conserved edges (%)</td>
<td># discrepant edges (%)</td>
</tr>
<tr>
<td>People vs. Bipartite clustering</td>
<td>45,779 (30.91%)</td>
<td>102,323 (69.09%)</td>
</tr>
<tr>
<td>Product vs. Bipartite clustering</td>
<td>(17.55%)</td>
<td>(82.45%)</td>
</tr>
</tbody>
</table>

are discrepant edges. In the case of the people network, only 30.91% of the edges are conserved (i.e., maintained by the bipartite clustering); whereas for the product network this figure is further lower (17.55%). While the fraction of conserved edges may appear low, it is much higher when compared to the congruence (0.99%) observed through direct link–level comparison (shown in Table 15). This attests to the value added to the overall understanding of community structure by augmenting 1-mode networks with bipartite information. The large fraction of discrepant edges, on the other hand, can be attributed to the lack of coordination in developing open source software. We also measured the distribution of the 1-mode network edges among the bipartite clusters (see the second half of Table 17). As can be observed, the frequency of conserved edges per cluster is only marginally higher than the proportion of discrepant edges across two clusters.
5.4 Conclusion

The primary contribution in this study is a broader view of coordination within virtual community based product development. In contrast to the existing approaches that only focus on explicit communication as a means for coordination, our approach accounts for implicit and indirect communication between individuals and the presence of shared knowledge possessed by individuals within a community. In terms of the methodology, the proposed approach is unique because it is based on clustering techniques rather than direct comparison of links within networks. As the number of nodes \(n\) in a network increases, the number of possible links between nodes increases at a faster rate \(n^2\). Because of this, as the number of individuals grows, the congruence measures become significantly smaller. Hence, if the actual coordination increases at a constant rate with increasing number of nodes, the congruence measure would decrease. Because of this, existing measures may provide misleading information about changes in coordination over time in evolving communities. The advantage of the proposed clustering-based approach is that it is less sensitive to the increase in the network size and can be used for comparison purposes as the communities grow. The three approaches for comparison discussed in this study (link-link, link-cluster, and cluster-cluster) are based on different assumptions about the extent of indirect communication and shared knowledge. If the communities are strongly tied with each other and it can be assumed that there is significant indirect communication, then cluster-cluster comparison approach should be used. On the other hand, if the communities are such that coordination only happens through explicit communication, the link-link comparison is more applicable. In the intermediate scenarios, the link-cluster based comparison should be used. The availability
of three approaches at different points in the spectrum provides analysts the flexibil-
ity to choose the approach which is more applicable to a particular scenario. One of
the limitations of the results presented in this study is that we model the networks as
undirected networks. More insights into coordination can be gained by modeling the
communications and dependencies as directed networks and using clustering approaches
for directed networks.
Chapter 6

Conclusions and Future Research Directions

Graph clustering is an efficient tool for analysis, modeling and prediction of the evolution of networks in a wide variety of scientific domains. Although theoretical formulations of clustering are either intractable or computationally prohibitive, there are efficient algorithmic heuristics that solve this important problem in a satisfactory fashion. Nonetheless, the implementation of practical heuristics to process large datasets still remains a very difficult challenge due to several factors such as: vast amount of the data, irregular data access patterns and compute-intensive operations to improve the approximation.

In the dissertation, we have successfully:

1. Presented novel MapReduce algorithms and implementations to efficiently parallelize the serial Shingling heuristics on large-scale distributed memory clusters running as either Hadoop or MPI platforms.

2. Developed a new variant of the Shingling heuristic that shows significant qualitative improvements over the standard serial version.

3. Introduced a new formulation for clustering protein sequences based on domains, and presented a novel application of MapReduce implementation to address this formulation.

4. Presented extensive experimental results for MPI implementation on a real world metagenomics graph containing 10.3M vertices and 640M edges and its subsets.
5. Developed and evaluated two approaches to measuring socio-technical coordination in virtual communities based on clustering techniques.

Future research includes an integration of our clustering algorithms with available graph-construction methods. This tool will provide biologists and other non-HPC users with a transparent scientific workflow for automatic homology graph construction and clustering given a set of sequences in FASTA format. Continued parametric studies and benchmark tests would be also beneficial for relevant real world networks.

A challenge that remains due to the nature of biological data is the lack of objective measurements of clustering quality and significance. Nonetheless, there is a need for automatic selection of the cluster(s) that achieve the highest agreement with the biological variables being studied. We are currently working on development of such techniques in order to aid the researcher better navigate clustering results.

Future research opportunity for virtual communities includes utilizing different clustering algorithms for clustering the networks. The use of the Louvain method was appropriate for the relatively small size networks, when larger graphs are available our parallel methods would be more beneficial. In this study, we let the clustering algorithm determine the number of clusters based on maximizing the modularity. The effect of number of clusters on the results of comparison is also an important avenue for future work.

There is also an increasing need to handle large volumes of dynamically changing graph-structured data and development of fast and accurate parallel algorithms to cluster such networks would provide an efficient tool to analyze them in a timely manner.
Bibliography


and Awareness Tools’, *ACM Conf. on Computer-Supported Work*, Banff, Canada, pp. 353-362.


[38] Personal communication, The Hallam lab at University of British Columbia, Vancouver, BC. (2012)


