Iowa Research Online
The University of Iowa's Institutional Repository

University of Iowa
Iowa Research Online

Theses and Dissertations

Fall 2016

Shared and distributed memory parallel algorithms to solve big data problems in biological, social network and spatial domain applications

Rahil Sharma
University of Iowa

Copyright © 2016 Rahil Sharma

This dissertation is available at Iowa Research Online: https://ir.uiowa.edu/etd/2277

Recommended Citation
Sharma, Rahil. "Shared and distributed memory parallel algorithms to solve big data problems in biological, social network and spatial domain applications." PhD (Doctor of Philosophy) thesis, University of Iowa, 2016.
https://ir.uiowa.edu/etd/2277.

Follow this and additional works at: https://ir.uiowa.edu/etd

Part of the Computer Sciences Commons

SHARED AND DISTRIBUTED MEMORY PARALLEL ALGORITHMS TO SOLVE BIG DATA PROBLEMS IN BIOLOGICAL, SOCIAL NETWORK AND SPATIAL DOMAIN APPLICATIONS

by

Rahil Sharma

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2016

Thesis supervisor: Professor Suely Oliveira

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Rahil Sharma

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the December 2016 graduation.

Thesis committee: ______________________________
Suely Oliveira, Thesis Supervisor

____
Kasturi Varadarajan

____
Sukumar Ghosh

____
David Stewart

____
Isabel Darcy

In the loving memory of my grandfather Baldevkrishan Sharma ...

To my loving family ...

ACKNOWLEDGEMENTS

Pursuing my Ph.D. has been an exciting and challenging journey, which would not have been possible without the help and support of many people from my professional and personal life. I would like to express my gratitude to these individuals.

First and foremost I want to thank my advisor and mentor Professor Suely Oliveira, who not only provided me with great guidance and support, but who also motivated and encouraged me throughout this journey. I would like to thank Professor Kasturi Varadarajan, whose courses in Algorithms first sparked my interest in research in this area. His valuable guidance during my Masters and initial stages of the Ph.D. helped lay a solid foundation upon which I could build. I would like to thank my other committee members: Professor Sukumar Ghosh, Professor David Stewart, and Professor Isabel Darcy who were very supportive. I would also like to thank Dr. Ramanathan Sugumaran who gave me the opportunity to collaborate with "John Deere" on projects related to my research through an academic grant for the year 2015-2016.

My heartiest thanks to all members of the Department of Computer Science at The University of Iowa, including faculty, students, and program administrators. It was a privilege to be a part of this Department.

I am not the only one who has experienced the success and stress of the past few years. I am indebted to my parents Rishi & Shaila Sharma, my brother Rishabh Sharma, my remaining family (Sharma's & Batra's) and my fiancee Hemali, who

have journeyed with me and provided me with love and encouragement. Last, but not least, I want to thank all my friends in Iowa City, who have helped make this town my home for the last five and a half years.

ABSTRACT

Big data refers to information which cannot be processed and analyzed using traditional approaches and tools, due to 4 V's—sheer Volume, Velocity at which data is received and processed, and data Variety and Veracity. Today massive volumes of data originate in domains such as geospatial analysis, biological and social networks, etc. Hence, scalable algorithms for efficient processing of this massive data is a significant challenge in the field of computer science. One way to achieve such efficient and scalable algorithms is by using shared & distributed memory parallel programming models. In this thesis, we present a variety of such algorithms to solve problems in various above mentioned domains. We solve five problems that fall into two categories.

The first group of problems deals with the issue of *community detection*. Detecting communities in real world networks is of great importance because they consist of patterns that can be viewed as independent components, each of which has distinct features and can be detected based upon network structure. For example, communities in social networks can help target users for marketing purposes, provide user recommendations to connect with and join communities or forums, etc. We develop a novel sequential algorithm to accurately detect community structures in biological protein-protein interaction networks, where a community corresponds with a functional module of proteins. Generally, such sequential algorithms are computationally expensive, which makes them impractical to use for large real world networks. To

address this limitation, we develop a new highly scalable Symmetric Multiprocessing (SMP) based parallel algorithm to detect high quality communities in large subsections of social networks like *Facebook* and *Amazon*. Due to the SMP architecture, however, our algorithm cannot process networks whose size is greater than the size of the RAM of a single machine. With the increasing size of social networks, community detection has become even more difficult, since network size can reach up to hundreds of millions of vertices and edges. Processing such massive networks requires several hundred gigabytes of RAM, which is only possible by adopting distributed infrastructure. To address this, we develop a novel hybrid (shared + distributed memory) parallel algorithm to efficiently detect high quality communities in massive *Twitter* and *.uk domain* networks.

The second group of problems deals with the issue of *efficiently processing spatial Light Detection and Ranging (LiDAR) data*. LiDAR data is widely used in forest and agricultural crop studies, landscape classification, 3D urban modeling, etc. Technological advancements in building LiDAR sensors have enabled highly accurate and dense LiDAR point clouds resulting in massive data volumes, which pose computing issues with processing and storage. We develop the first published landscape driven data reduction algorithm, which uses the slope-map of the terrain as a filter to reduce the data without sacrificing its accuracy. Our algorithm is highly scalable and adopts shared memory based parallel architecture. We also develop a parallel interpolation technique that is used to generate highly accurate continuous terrains, i.e. Digital Elevation Models (DEMs), from discrete LiDAR point clouds.

PUBLIC ABSTRACT

Big data refers to information which cannot be processed and analyzed using traditional approaches and tools, due to 4 V's—sheer Volume, Velocity at which data is received and processed, and data Variety and Veracity. Today massive volumes of data originate in domains such as geospatial analysis, biological and social networks, etc. Hence, scalable algorithms for efficient processing of this massive data is a significant challenge in the field of computer science. One way to achieve such efficient and scalable algorithms is by using shared & distributed memory parallel programming models. In this thesis, we present a variety of such algorithms to solve problems in various above mentioned domains. We solve five problems that fall into two categories.

The first group of problems deals with the issue of *community detection*. Detecting communities in real world networks is of great importance because they consist of patterns that can be viewed as independent components, each of which has distinct features and can be detected based upon network structure. For example, communities in social networks can help target users for marketing purposes, provide user recommendations to connect with and join communities or forums, etc. We develop a novel sequential algorithm to accurately detect community structures in biological protein-protein interaction networks, where a community corresponds with a functional module of proteins. Generally, such sequential algorithms are computationally expensive, which makes them impractical to use for large real world networks. To

address this limitation, we develop a new highly scalable Symmetric Multiprocessing (SMP) based parallel algorithm to detect high quality communities in large subsections of social networks like *Facebook* and *Amazon*. Due to the SMP architecture, however, our algorithm cannot process networks whose size is greater than the size of the RAM of a single machine. With the increasing size of social networks, community detection has become even more difficult, since network size can reach up to hundreds of millions of vertices and edges. Processing such massive networks requires several hundred gigabytes of RAM, which is only possible by adopting distributed infrastructure. To address this, we develop a novel hybrid (shared + distributed memory) parallel algorithm to efficiently detect high quality communities in massive *Twitter* and *.uk domain* networks.

The second group of problems deals with the issue of *efficiently processing spatial Light Detection and Ranging (LiDAR) data*. LiDAR data is widely used in forest and agricultural crop studies, landscape classification, 3D urban modeling, etc. Technological advancements in building LiDAR sensors have enabled highly accurate and dense LiDAR point clouds resulting in massive data volumes, which pose computing issues with processing and storage. We develop the first published landscape driven data reduction algorithm, which uses the slope-map of the terrain as a filter to reduce the data without sacrificing its accuracy. Our algorithm is highly scalable and adopts shared memory based parallel architecture. We also develop a parallel interpolation technique that is used to generate highly accurate continuous terrains, i.e. Digital Elevation Models (DEMs), from discrete LiDAR point clouds.

TABLE OF CONTENTS

Page

LIST OF TABLES xii

LIST OF FIGURES xiii

CHAPTER

1 BIOLOGICAL DOMAIN 1

1.1 Introduction 1
1.2 Related Work 3
1.3 Contribution 6
1.4 Features of PPI Network 6
1.5 BLLP Algorithm 12
1.5.1 Preprocessing: Topological Weight Assignment (Level 1) . 12
1.5.2 Coarsening 14
1.5.3 Labeling and Interpolation 14
1.5.4 Label Propagation 16
1.6 Computational Results 18
1.6.1 BLLP on PPI yeast network 20
1.6.2 Functional Module Prediction for Uncharacterized Proteins 23
1.6.3 Comparative Study 27
1.7 Discussion 30

2 PARALLEL PROGRAMMING MODELS 32

2.1 P-threads 34
2.2 OpenMP 35
2.3 CUDA 36
2.4 Message Passing Interface (MPI) 38
2.5 Map-Reduce 38
2.6 Hadoop 40

3 SOCIAL NETWORKS DOMAIN : SHARED MEMORY 43

3.1 Introduction 43
3.2 Related Work 44
3.3 Contribution 46
3.4 MCML Algorithm 47
3.4.1 Preprocessing: Edge Strength Assignment 47
3.4.2 Remove Weak Edges 48

3.4.3 Multi-level Coarsening 49
3.4.4 Parallel Implementation 50
3.5 Computational Results 55
3.5.1 Benchmark Datasets 55
3.5.2 Facebook Forum Dataset 59
3.5.3 Amazon Product Dataset 63
3.6 Discussion 64

4 SOCIAL NETWORKS DOMAIN : DISTRIBUTED MEMORY 66

4.1 Introduction 66
4.2 Related Work 68
4.3 Contribution 72
4.4 Hybrid Algorithm 73
4.5 Computational Results 76
4.5.1 Datasets 77
4.5.2 Evaluation 79
4.6 Discussion 85

5 SPATIAL DOMAIN : DATA REDUCTION 86

5.1 Introduction 86
5.2 Related Work 87
5.3 Contribution 88
5.4 Data Reduction Algorithm 89
5.4.1 Preprocessing 89
5.4.2 Algorithm 91
5.4.3 Parallel Implementation 93
5.5 Computational Results 95
5.6 Discussion 98

6 SPATIAL DOMAIN : INTERPOLATION 100

6.1 Introduction 100
6.2 Related Work 102
6.3 Contribution 103
6.4 Spatial Interpolation 103
6.4.1 Algorithm 103
6.4.2 Parallel Implementation 105
6.5 Computational Results 106
6.6 Discussion 109

7 EXPERIMENTAL ENVIRONMENT 110

7.1 Chapter 1 : Biological Domain 110
7.2 Chapter 3 : Social Networks Domain - Shared Memory 110
7.3 Chapter 4 : Social Networks Domain - Distributed Memory . . . 110
7.4 Chapter 5 & 6 : Spatial Domain - Data Reduction & Interpolation 111

REFERENCES 112

LIST OF TABLES

Table | Page

1.1 Jaccard Index to quantify the distance between protein complexes in MIPS database and functional module partitions by BLLP algorithm 19

1.2 Number of detected communities and corresponding sizes, in PPI yeast network 21

1.3 Incorrect prediction of functional modules made by the BLLP algorithm, for uncharacterized proteins 25

1.4 Correct prediction of functional modules made by the BLLP algorithm for uncharacterized proteins 27

1.5 Comparing various community detection algorithms on PPI yeast network 28

2.1 Synchronization Functions in CUDA 36

3.1 Comparing various community detection algorithms for Karate and Dolphin club benchmark datasets 58

3.2 Comparing various community detection algorithms for Facebook Forum and Amazon datasets based on modularity and computational time using 16 cores. (The blank values are not available in the literature of this research area. To get the computational time we include all the three stages of the algorithm.) 62

4.1 Random graph datasets 77

4.2 Real world social network datasets 78

LIST OF FIGURES

Figure Page

1.1 PPI yeast network in which 2361 proteins are linked by 7182 interactions and 536 self interactions 10

1.2 Properties of PPI yeast network 11

1.3 (a) Network G (b) Pre-processing : Topological weight Assignment (Level 1) (c) Coarse graph G' (d) Labeling : Find connected components and give common label to nodes in same component (e) Interpolation : Transfer labels from G to G' (f) Label Propagation 15

1.4 (a) PPI yeast network with 123 communities : red = largest community with 1279 nodes, blue = second largest community with 602 nodes (b) Correctness of groupings made by BLLP 19

1.5 YIL070C uncharacterized protein interacts with other known proteins . . 23

1.6 (a) FC clustering with largest community having 250 proteins (b) LPA clustering with largest community having 2224 proteins 30

2.1 Programming models and their supported system architecture 33

2.2 GPU Architecture 37

2.3 Map-Reduce Computation Semantics 40

2.4 Map-Reduce Task and Job Tracking 41

3.1 MCML Algorithm : General Schema 49

3.2 Parallel Preprocessing 51

3.3 Parallel Multilevel Coarsening 52

3.4 Karate Club : (a) $\beta = 0.0$ (b) $\beta = 0.1$ (c) $\beta = 0.4$ (d) $\beta = 0.6$ (e) $\beta = 1.0$ (Note: Edges highlighted in (d) and (e) have stronger connections than other edges in the graph) 56

3.5 Non-community nodes vs strength : (a) Karate club (b) Dolphin club . . 57

3.6 Accuracy Plot : Karate club and Dolphin club 57

3.7 Dolphin club : (a) $\beta = 0.0$ (b) $\beta = 0.1$ (c) $\beta = 0.4$ (d) $\beta = 0.6$ (e) $\beta = 1.0$ (Note: Edges highlighted in (d) and (e) have stronger connections than other edges in the graph) 59

3.8 (a) 2-mode weighted network (b) preprocessed 1-mode weighted network 60

3.9 Facebook forum : (a) Computational time Vs number of cores (b) Speed-ups 61

3.10 Amazon : (a) Computational time Vs number of cores (b) Speed-ups . . 63

4.1 Example : Hybrid Algorithm 74

4.2 Run-time while scaling up the number of processor cores over varying graph sizes 80

4.3 Speedups compared to sequential hybrid algorithm while scaling up the number of processor cores over varying graph sizes 80

4.4 Run-time while scaling up the graph sizes over varying processor cores . 81

4.5 Change in error percentage of final modularity compared to that achieved by sequential execution of hybrid algorithm 82

4.6 Run-time while scaling up the number of processor cores 83

4.7 Speedups compared to base run of hybrid algorithm while scaling up the number of processor cores up to 128 84

4.8 Change in error percentage of final modularity compared to that achieved by base run of hybrid algorithm while scaling up the number of processor cores up to 128 84

5.1 (a) Imagery of the study area (b) 2-D LiDAR point cloud of the study area which is colored based on variations in elevation 89

5.2 (a) Statistical analysis of elevation data for the terrain (b) Slope-map consisting of four slope ranges 90

5.3 Data reduction algorithm : Grid overlayed on processed LiDAR data . . 92

5.4 Parallel implementation of data reduction algorithm 94

5.5 DEM generated for the original dataset having 6.94 million LiDAR points 95

5.6 (a) Parallel speed-ups for LiDAR data reduction algorithm (b) Data reduction and DEM accuracy . . . 96

5.7 (a) $\beta = 80\%$, reduced dataset comprising of 2.2 million points (b) $\beta = 90\%$, reduced dataset containing 3.1 million points . . . 98

6.1 (a) Grid of k sq.m. ($k > 0$) overlaid on the LiDAR data (b) Selecting a cut-off radii and assigning weights to grid intersections (c) Assigning weights to each grid cell . . . 104

6.2 Parallel split and merge phases of our spatial interpolation algorithm . . 106

6.3 (a) Imagery of study area for dataset 2 (b) Elevation map of dataset 2 showing terrain with less roughness, shallow valleys and flat regions . . . 107

6.4 (a) Parallel speed-ups for our spatial algorithm (b) RMSE for IDW and modified IDW at different LiDAR density levels for two datasets using validation method . . . 108

CHAPTER 1
BIOLOGICAL DOMAIN

1.1 Introduction

Most cellular processes are believed to be carried out by groups of highly interacting proteins called functional modules, protein complexes, or molecular complexes. Recent large-scale high-throughput experiments, and integration of published data, have generated large protein-protein interaction (PPI) networks. Even one of the simplest eukaryotic organism, yeast, has more than five thousand proteins. Protein complexes can be detected by identifying highly connected sets of proteins in PPI networks. Computational identification of functional modules or protein complexes can provide an inexpensive guideline for biological experiments. Protein-protein interactions can alter kinetic properties of enzymes, create new binding sites for small effector molecules, destroy or inactivate the protein, exhibit a new functionality which a single protein cannot exhibit alone, etc.

There have been many recent computational approaches to disclose the underlying biological structures ([6, 21, 42, 79, 94, 98]). These approaches are divided into two groups. One group uses machine learning approaches to construct weighted networks by integrating existing datasets to predict protein complexes ([42, 98]). Another group tries to extract highly connected subgraphs or divide a whole network into groups of clusters on a protein-protein interaction (PPI) network ([6, 21, 79, 94, 63]).

There are a number of challenges in treating protein-protein interaction data.

One is that many high-throughput experiments have high error rates, which results in a great many false positives for interactions between proteins. Another challenge is that some proteins are interaction mediating proteins that interact with very large numbers of other proteins; these might, for example, provide common services to many different parts of the cell. The former challenge can make accurate identification of functional modules difficult, while the latter challenge tends to make the entire proteome appear to be a single indivisible functional module.

Protein complexes corresponds to modules, which can be viewed as dense sub-networks or communities in PPI networks. Community detection algorithms can be used extract these dense sub-networks/communities. Modules can be defined in many ways with, for example, densely connected sub-networks with more intra-node edges as compared to inter-node edges. Some of these definitions are present in the literature ([75]). Functional module detection in PPI networks is computationally very hard. Based on different definitions of modules, there are many community detection algorithms in the current state of the art, which are used to identify functional modules in PPI networks. Trivial algorithms are not well suited for this job ([96]).

One class of the community detection technique relies on finding completely connected sub-networks (*cliques*) in PPI network ([89]). This technique is very inefficient for detecting modules for large PPI networks, and also proteins participating in a complex, rarely have interactions with all other proteins in that complex. Another class of community detection algorithms rely on finding dense sub-networks in PPI network (not essentially cliques) ([2]). Such algorithms mis-classify low-shell proteins

into distinct clusters, whereas they could have been classified in a same core cluster. Due to this weak connectivity, many biological interactions are ignored. Similar pitfalls are also been observed, when hierarchical community detection algorithms are used to find modules in PPI networks ([46, 77]). We have developed a bi-level community detection algorithm based on label propagation (BLLP) [65], which eliminates the above pitfalls in identifying modules in PPI networks, and also helps to control large communities which dominate the network. Hence we retrieve communities with high modularity, high accuracy of matching with ground truth functional modules, and in less computational time.

The remainder of this chapter is organized as follows: In Section 1.2, we describe the related work. We state our contributions in Section 1.3. In Section 1.4, we will talk about important features of PPI networks and describe the PPI yeast network that we used for all simulations, followed by the description of our bi-level label propagation algorithm in Section 1.5. In Section 1.6 we show the computational results along with a comparative study with other well known community detection algorithms. We also show the accuracy of BLLP algorithm in predicting functional modules of uncharacterized proteins. We finally end this chapter with some discussion in Section 1.7.

1.2 Related Work

Multi-level Spectral Algorithms : Pothen et al. ([79]) proposed a two-level architecture for a yeast proteomic network. They construct small networks from

a PPI network by removing proteins which interact with too many or too few proteins. Removing proteins with too few interactions can eliminate many effects of false positives. These proteins have a tendency to be classified in larger clusters to which they interacts, even if the interaction has low confidence value. Such low-shell proteins generally group with other low-shell proteins. There are specific proteins that function as substrates of protein receptors, such as G-protein coupled receptors (GPCR's). This is a specific interaction between molecules that have concordant configurations. We can also apply clustering again to these low-shell proteins which are removed, to avoid ignoring important biological processes. On the other hand, removing proteins with the largest numbers of interactions can make the finer structure of the interactions more evident. A clustering algorithm is applied to this residual network. Validation of clusters is performed by comparing the clustering result with the protein complex database, the Munich Information Center for Protein Sequences (MIPS). A spectral clustering method plays a critical role for identifying functional modules in the PPI network in their research.

S.Oliveira and S.C. Seok ([62]) successfully applied a multilevel spectral algorithm to cluster a group of documents using similarity matrices which are mostly dense with entries between 0 and 1, and has developed a matrix-based multilevel approach to identify functional protein modules ([63]). Like large-scale networks, the vertex connectivities of proteomic networks follow a scale-free power-law distribution ([13]). That means that, the proteomic network consists of a small number of high degree nodes and a majority of low degree nodes. However, the proteomic network

has no edge weights. Multilevel algorithms have a long history, mostly for partial differential equations in numerical analysis but also for network partitioning, such as METIS ([34]). Multilevel schemes have been applied to network clustering too ([20, 62]).

Community Detection Algorithms : A community in a network is a set of nodes that are densely connected with each other and sparsely connected to the other nodes in the network. A group of proteins in the same community within a PPI network, frequently coincide with known functional modules or protein complexes. Many proposed algorithms to detect community structures in complex networks are based on graph theory. Most of the work in this area is focused on enhancing the modularity [61], that is to increase the number of intra-cluster edges as opposed to inter-cluster edges. In our comparative study, we use modularity as a metric to determine the quality of our results over other well known algorithms, when applied to PPI network. Some of these algorithms use techniques like betweenness centrality ([19]), hierarchical clustering ([91]), and label propagation ([40]). A thorough review of community detection algorithms for networks is given in ([24]). It consists of various techniques, methods and datasets for detecting communities in biology, computer science and other disciplines, where the system is represented as a network. More literature review on community detection algorithms is given in Chapters 3 and 4.

1.3 Contribution

We develop a bi-level community detection algorithm based on label propagation (BLLP) [65], which we use to accurately identify functional protein modules in PPI networks. The BLLP algorithm involves a pre-processing stage, where the edge weights of the network are computed based on its topological features, a step similar to coarsening of the original network, followed by a label propagation algorithm ([76]) and a post-processing step to improve the quality of communities detected. Even though it is possible for a protein to have multiple functional modules, our algorithm does not detect overlapping communities, i.e., does not identify multiple functional module for a single protein. Instead it identifies one strong functional module for a protein under consideration. Using BLLP algorithm, we design a prediction model to predict functional module of uncharacterized proteins.

1.4 Features of PPI Network

Graph theory is commonly used as a method for analyzing PPIs in Computational Biology. Each vertex represents a protein, and edges correspond to experimentally identified PPIs. Proteomic networks have two important features ([17]). One is that the degree distribution function $P(k)$ (the number of nodes with degree k) follows a power law $P(k) \approx \text{constant}\, k^{-\alpha}$ and so is considered a scale-free network. This means that, most vertices have low degrees, called low-shell proteins, and a few are highly connected, called hub proteins. The other feature is the *small world* property which is also known as *six degrees of separation*. This means the diameter of the

network is small compared with the number of nodes.

The standard tools to understand these networks are the clustering coefficient (Cc), the average path length, and the diameter of the network. The clustering coefficient (Cc_i) is defined in terms of $E(G(v_i))$, the set of edges in the neighborhood of v_i. The clustering coefficient is the probability that a pair of randomly chosen neighbors of v_i are connected. That is,

$$Cc_i = \frac{2}{deg(v_i)[deg(v_i) - 1]} \cdot |E(G(v_i))| \tag{1.1}$$

where $G(v_i)$ is the neighborhood of v_i, and $E(G(v_i))$ is the set of edges in $G(v_i)$.

Since the denominator is the maximum possible number of edges between vertices connected to v_i, $0 \leq Cc_i \leq 1$. The global clustering coefficient can be simply the average of all individual clustering coefficients (Cc_i) like

$$\overline{Cc} = \sum_{i=1}^{n} Cc_i/n \tag{1.2}$$

But this 'average of an average' is not very informative ([17]); one alternative is to weight each local clustering coefficient

$$Cc = \sum_{i=1}^{n} \frac{deg(v_i)}{MaxDeg} Cc_i/n \tag{1.3}$$

where $MaxDeg$ is the maximum degree in the network. We use the latter Cc as our clustering coefficient for the rest of the report.

The path length of two nodes v_i and v_j is the smallest sum of edge weights of paths connecting v_i and v_j. For an unweighted network, it is the smallest number of edges connecting v_i and v_j. The average path length is the average of path lengths

of all pairs (v_i, v_j). The diameter of the network is the maximum path length. Some other important features about PPI networks are:

- The hub proteins have interactions with many other proteins, so it is hard to limit them to only one cluster and the computational complexity increases when they are included.
- There are many low-shell proteins, which increases the size of the network. These nodes are easy to cluster when the nodes they are connected to are clustered first.
- Proteomic networks are mostly comprised of one big component and several small components.

Now we shall talk about these features with our model of yeast network. There are databases which contain protein-protein interactions as well as cellular localization, gene regulation, and the context of these interactions. The best known are KEGG (`www.genome.ad.jp/kegg`), BIND (`www.bind.ca`), MIPS (`mips.gsf.de`), PROnet (`www.pronet.doubletwist.com`) and DIP (`dip.doe-mbi.ucla.edu`) which are used to test the algorithms.

Over the last decade, high throughput interaction detection approaches like yeast two-hybrid system ([90]), protein complex purification technique using mass spectrometry ([30]), correlated messenger RNA expression and genetic interaction data ([32]), etc., have created number of datasets of PPI interactions for several eukaryotic organisms, like yeast. The interactions produced by using the above

techniques have many false positives. In order to measure the accuracy of these interactions, authors of ([92]) checked 80,000 interactions amongst 5400 yeast proteins and assigned each interaction a confidence value. To eliminate the effect of false positives in our predictions, we focus on 7182 interactions with high confidence value (above 75%), among 2361 proteins. This proteomic network for yeast is given in Pajek Datasets (`http://vlado.fmf.uni-lj.si/pub/networks/data/bio/Yeast/Yeast.htm`). To evaluate the clustering results, we compare with the functional modules given in the Munich Information center for Protein Sequence (MIPS), which has a list of experimentally trusted functional modules in yeast. Each protein is given a short label for identification (called PIN on MIPS) and a functional module. We select 57 proteins from 2361 proteins, having the following functional modules: 20 r-RNA, 11 Ribosome biogenesis, 7 Splicing, 6 Lipid oxidation, 2 Protein binding, 3 t-RNA, 3 m-RNA, 2 Tricarboxylic acid pathway, 2 Nuclear degradation and 1 Catabolism. Then we mask these functional modules for the 57 proteins and term them as uncharacterized proteins. We then make predictions on which functional modules they belong to, based on the BLLP algorithm's outcome and compare it with the ground truth functional module. We also conduct network analysis to determine the following features of PPI Yeast network we use for our study:

Betweenness Centrality Distribution : Betweenness centrality is the number of shortest paths between a pair of vertices that pass through a node. There is a pair of proteins in the PPI yeast network, through which ≈ 950 shortest paths pass. These proteins are considered as a hub-proteins. In our data set, these proteins are

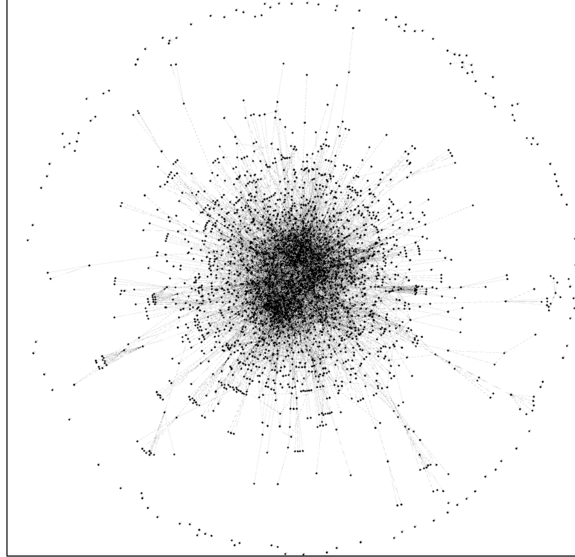

Figure 1.1: PPI yeast network in which 2361 proteins are linked by 7182 interactions and 536 self interactions

labeled YMR093W and YHR052W. Their functional module is r-RNA processing. These proteins have majority of interactions with other proteins within the largest community we detected in our dataset, being the functional module for r-RNA processing. They also have considerable interaction with proteins in the ribosome biogenesis functional module.

Clustering Coefficient : The clustering coefficient of the PPI yeast network, i.e., the measure of the degree to which nodes in this network tend to cluster together is 0.271. There are $\approx$ 1300 nodes whose clustering coefficient is 0; i.e., low-shell proteins.

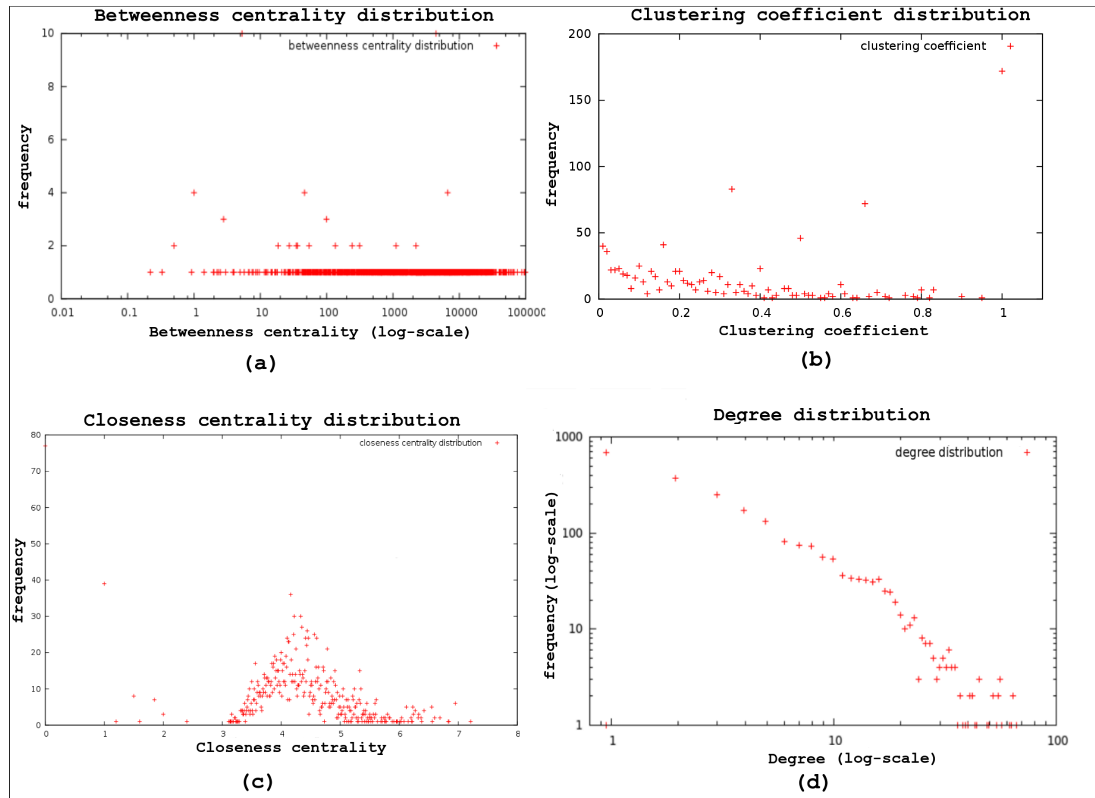

Figure 1.2: Properties of PPI yeast network

Closeness Centrality Distribution : The closeness centrality C_i of a node i in a network is the inverse of the mean shortest path distance from i to every other node in the network

$$C_i = \frac{n-1}{\sum_{i \neq j} dist(i,j)} \tag{1.4}$$

where $dist(i, j)$ is the shortest path distance between nodes i and j, and n is total number of nodes in the network. If there exists no path between i and j then, n nodes are used in Equation 4, instead of the path length.

Degree Distribution : In Figure 1.2(d), we can see that the degree distribu-

tion function for PPI yeast network $P(k)$ (the number of nodes with degree k) follows a power law $P(k) \approx \text{constant}\, k^{-\alpha}$, and hence is a scale-free network ([13, 73, 72]).

Diameter, Average Path length, Number of Shortest Paths and Connected Components : The density of the network is 0.003. The diameter of the yeast network is 11. So the diameter is much smaller as compared to the number of nodes in the PPI yeast network network, thus it exhibits the *small world property.* The average path length is 4.3762. The total number of shortest paths is $4,944,096$. There are 78 weakly connected components.

1.5 BLLP Algorithm

In this section, we present a bi-level label propagation community detection algorithm, involving a pre-processing stage where the edge weights of the network are computed based on its topological features, a step similar to coarsening of the original network (level 1), followed by applying label propagation algorithm ([76]) once on coarsened network (level 2), and then iteratively in level 1, incorporating a post-processing step to improve the quality of the communities detected.

1.5.1 Preprocessing: Topological Weight Assignment (Level 1)

The BLLP algorithm finds communities in a network $G(V, E)$ where V represents the nodes/vertices/proteins and E represents the edges/interactions between the nodes, by assigning weights to the edges and tracking the propagation of the label through the network. It is desirable to assign edge weights that most accurately

represent the topological structure of the network in the BLLP algorithm. Since we do not have any prior knowledge of the community structure, we assign weights to each edge based on the significance of that edge to the other nodes in the network, and to the nodes at the end points of that edge. For each edge $e(i, j)$ (where i and j are nodes) in the fine network G, the topological edge weight $w_{top}(i, j)$ assigned to it is the ratio of number of triangles that edge $e(i, j)$ participates in to the total number of triangles containing node i. If the weight of the edge $e(i, j)$ is greater than other edges in the 1-neighborhood of i then, node i and node j are more likely to be in the same community. Whereas on the contrary, if edge $e(i, j)$ has lower weight than most other edges in the 1-neighborhood of i, then node i and node j are less likely to be in the same community. Mathematically,

$$w_{top}(i, j) = \frac{t_{(i,j)}}{\sum_{(i,k)}(t_{(i,k)})} \quad ; \quad k \in N_i \tag{1.5}$$

where N_i is the 1-neighborhood of i, and $t_{(i,j)}$ is the total number of triangles whose sides contain edge (i, j).

The BLLP algorithm also works well with weighted networks, where the edges are assigned weights w_{input} as an input. To get the total weight of an edge, we simply have to take product of the topological weight of that edge, with its input weight.

$$w_{total}(i, j) = w_{top}(i, j) \times w_{input}(i, j) \tag{1.6}$$

We initialize the label of each node in the fine network G to their corresponding node id, which is also the label weight for that node. The propagation function used to

transfer labels between nodes in the network is given by:

$$L(i) = argmax_{j \in N_i}(L_j) \tag{1.7}$$

where L_j is the label (also referred as weight of the label) of node j in N_i (nodes one edge away from i). In the later subsections, we modify this propagation function before using it in the label propagation step.

1.5.2 Coarsening

In this step of the BLLP algorithm, we apply *coarsening* to the fine network G. For each node i of network G, we find the maximum weighted edge $e(i, j)$ in its 1-neighborhood. Now we find all such edges in the fine network G and copy them to a new network $G' = (V', E')$. That is, the set of edges E' is the set of edges $e(i, j)$ where $e(i, j)$ has the maximum weight in the 1-neighborhood of i. The set of vertices is the set of all i and j where $e(i, j)$ is in E'. Note that $|V| \geq |V'|$. There always exists some node pairs say n_1 and n_2 such that (n_1, n_2) is connected with a maximum weight edge and conversely so is (n_2, n_1). It is very important to label these nodes with the common label, to avoid *oscillatory behavior*, where these 2 nodes will keep on exchanging labels and algorithm will not converge, thus degrading the quality of the results.

1.5.3 Labeling and Interpolation

Now in *level 2* we find the connected components in this coarse network G' and then perform a label propagation algorithm until all the nodes in each component have common labels. This phase of finding locally relevant communities ensures good

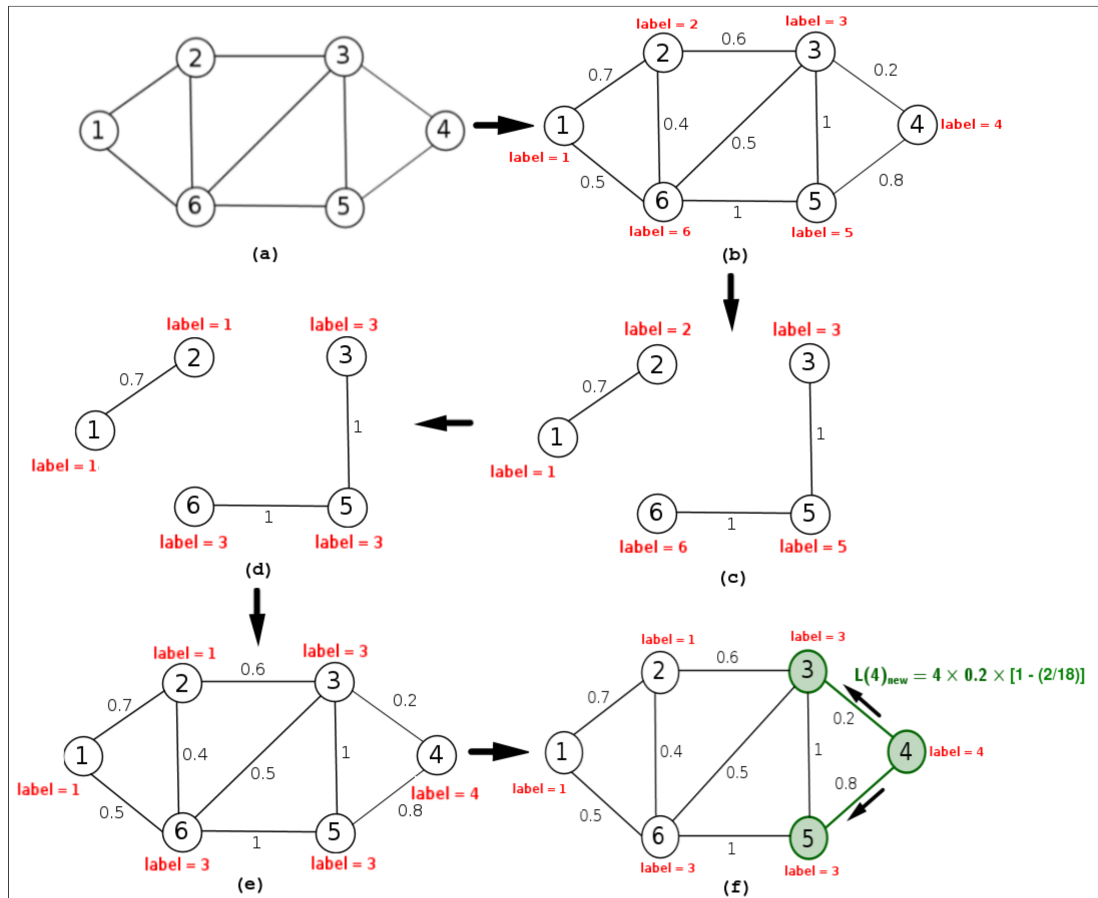

Figure 1.3: (a) Network G (b) Pre-processing : Topological weight Assignment (Level 1) (c) Coarse graph G' (d) Labeling : Find connected components and give common label to nodes in same component (e) Interpolation : Transfer labels from G to G' (f) Label Propagation

quality results. By doing this we also eliminate all the overlapping pairs. These components are local communities in the network. We then transfer the labels of the nodes in coarse network G' back to the fine network G. These are the new labels which will be used when the label propagation step begins.

Now before we go ahead let us take a small example to understand the pre-

processing, coarsening, and interpolation steps of the algorithm. We apply preprocessing step to the fine network G in Figure 1.3(a), where weights are assigned to all the edges based on the topological structure of G. Assume the weights are assigned as shown in Figure 1.3(b). All six nodes are given initial labels corresponding to their node identifier. In the coarsening step of BLLP algorithm, for each node in the fine network G, we find the maximum weighted edge in its 1-neighborhood. For example in Figure 1.3(b), $edge(1,2)$ is the maximum weighted edge for node 1 as well as node 2. Similarly, $edge(3,5)$ and $edge(5,6)$ are maximum weighted edges in the 1-neighborhoods of nodes 4 and 6 respectively. We copy all such edges and corresponding nodes in the new network G'. In the interpolation step we find connected components in this coarse network G'. Then we apply one iteration of label propagation, where all the nodes in G' send their label to every other node in their 1-neighborhood and each node assigns itself the lowest label it receives. This way all the nodes in each component have a common label. In Figure 1.3(d), we have two connected components in G' with label 1 and label 3. We then transfer these labels from coarse network G' to fine network G shown in Figure 1.3(e).

1.5.4 Label Propagation

Now we shall apply the label propagation algorithm to the fine network G. Each iteration of the label propagation algorithm includes the following steps;

1. every node i in the fine network G sends a label $L(i)_{new}$ given in Equation 1.9, to every other node in its 1-neighborhood

2. each node then assigns itself the maximum value label it receives

3. if the label of some node changed, repeat step 1

The algorithm terminates when all the nodes are stably labeled. It takes six iterations for the algorithm to terminate on the PPI yeast network. The nodes having the same labels after the termination of the label propagation algorithm, belong to the same community.

We use the label given in Equation 1.9 and not the label $L(i)$ given in Equation 1.7 because, if there is a large dominating community in the network, it will dominate all the other communities leading to the scenario where all the nodes in the network have same label. This is similar to spread of disease and is also referred to as *epidemic spread* in the network. BLLP *controls the community size* by assigning weight to each label, in each iteration of the label propagation, based on the following formula;

$$W_{label}(L_i) = 1 - \frac{d_C}{2m} \tag{1.8}$$

where d_C is the sum of degrees of all the nodes in community C with label L_i and m is total number of edges in the network. The modified label propagation function is

$$L(i)_{new} = argmax_{j \in N_i}[(L_j) \times w_{total}(j,i) \times W_{label}(L_i)] \tag{1.9}$$

where L_j is the label of node j, and $w_{total}(j,i)$ is the total weight of an $edge(i,j)$. So we can see that, if the size of the community increases the value of $W_{label}(L_i)$

decrease and so the chances of this label to propagate further in the network decreases. Conversely, if the size of the community decreases the value of $W_{label}(L_i)$ increases and so the chances of this label to propagate further in the network increases.

For more clarity on how labels are propagated, let us consider the example given in Figure 1.3(f), where node 4 propagates its label to its neighbor node 3. The label propagation function used by node 4 to propagate its label to node 3 is given by $L(4)_{new}$ = (current label of node 4) $\times$ (edge weight of $edge(4,3)$) $\times W_{label}(L_4)$, where $W_{label}(L_4)$ is determined by Equation 1.8, which prevents the epidemic spread in the network. Node 3 receives labels from nodes 2, 4, 5 and 6 and then assigns itself the maximum value label it receives.

We determine the quality of the detected communities using the modularity metric, denoted by,

$$Q = \frac{1}{2m}\Big[\sum_{i,j}[A_{(i,j)}\delta(C_i, C_j)] - \sum_{i}\frac{d_{C_i}}{d_i}\Big] \tag{1.10}$$

where $A_{(i,j)}$ is the adjacency matrix. C_i and C_j denotes the communities in which nodes i and j belong respectively. $\delta(C_i, C_j)$ is the Kronecker function such that

$$\delta(C_i, C_j) = \begin{cases} 1 & ; \; C_i = C_j \\ 0 & ; \; C_i \neq C_j \end{cases}$$

d_{C_i} denotes the degree of community C_i and d_i is the degree of node i.

1.6 Computational Results

Computational results for BLLP algorithm when applied to PPI yeast network is shown in this section. We show that, using our algorithm, highly accurate predic-

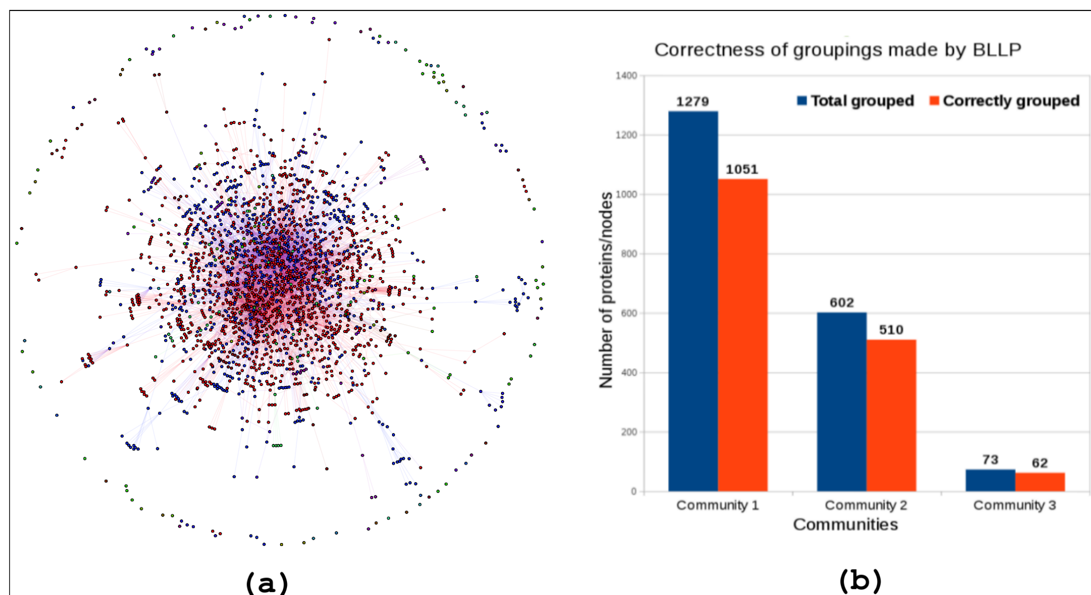

Figure 1.4: (a) PPI yeast network with 123 communities : red = largest community with 1279 nodes, blue = second largest community with 602 nodes (b) Correctness of groupings made by BLLP

Communities	**Jaccard Coefficient**	**Jaccard Distance**
Community 1 (1279 proteins)	$\frac{1501}{1276+258} = 0.6974$	$1 - 0.6974 = 0.3026$
Community 2 (602 proteins)	$\frac{510}{602+92} = 0.7348$	$1 - 0.7348 = 0.2651$
Community 3 (73 proteins)	$\frac{62}{73+11} = 0.7380$	$1 - 0.7380 = 0.2619$

Table 1.1: Jaccard Index to quantify the distance between protein complexes in MIPS database and functional module partitions by BLLP algorithm

tions are made to identify functional modules for uncharacterized proteins. We also conduct a comparative study, comparing three different community detection algorithms ([19, 40, 10]) on PPI yeast network, based on modularity of the communities discovered, and computational time.

1.6.1 BLLP on PPI yeast network

The PPI yeast network we studied involves, 2361 proteins and 7182 interactions with 536 self-interactions, and 78 weakly connected components. BLLP extracts 123 different communities from this data set. The modularity of the extracted communities is 0.592. The MIPS Comprehensive Yeast Genome Database (CYGD) provides the catalog of protein-protein interactions, the protein complex catalog and the protein localization catalog which stores information related to the proximity of proteins in yeast ([57]). The protein complexes include more than 200 manually extracted protein complexes. We check the communities obtained by BLLP against the MIPS database to determine the percentage of correct matching.

In Table 1.1, Jaccard coefficient and Jaccard distance for the largest three communities detected by BLLP, compared with MIPS protein complexes is shown. Jaccard similarity coefficient, is a statistic used for measuring similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets. The Jaccard distance, which measures dissimilarity between sample sets, is complementary to the Jaccard similarity coefficient ([81]). As shown in Figure 1.4, in the largest community having 1279 proteins; the BLLP algorithm

# of communities	**# of Nodes**	**percentage of whole network**
1	1279	54.21
1	602	25.5
1	73	3.13
1	44	1.86
1	38	1.61
1	27	1.14
1	25	1.06
1	18	0.76
1	12	0.64
1	10	0.51
2	9	0.34
2	7	0.3
5	6	0.25
6	4	0.21
5	3	0.13
17	2	0.08
76	1	0.04

Table 1.2: Number of detected communities and corresponding sizes, in PPI yeast network

matches 1051 proteins correctly to their functional module i.e.82.20% correctness. In the second largest community having 602 proteins of which 510 proteins have been correctly matched i.e. 84.71% correctness and in the third largest community having 73 proteins of which 62 proteins have been correctly matched i.e. 84.93% correctness. All the components with ≤ 25 nodes and > 1 node are grouped with $92+\%$ accuracy.

We observe that, the largest community with 1279 proteins is dominated by proteins with functional module of r-RNA processing; of which 1051 proteins are classified correctly and 91% of mis-classified proteins belong to functional module of ribosome biogenesis. The second largest community with 602 proteins is dominated by proteins within the ribosome biogenesis functional module; of which 510 proteins are classified correctly and 87% of mis-classified proteins belong to the functional module for r-RNA processing. From this observation we infer that there are many proteins in the largest two communities detected, that may have multiple functional modules i.e., ribosome biogenesis as well as r-RNA processing. The third largest community with 73 proteins is dominated by proteins in the splicing functional module, of which 62 proteins are classified correctly, and remaining 11 proteins are mis-classified to functional module for r-RNA processing and oxidation of fatty acids, lipids, and bio-synthesis. Some of these 11 mis-classified proteins are classified into larger communities, which are dominated by proteins in the r-RNA and ribosome biogenesis functional modules. This effect is much less as compared to same seen when other community detection algorithms (Subsection 1.6.3) are applied to the PPI yeast network.

1.6.2 Functional Module Prediction for Uncharacterized Proteins

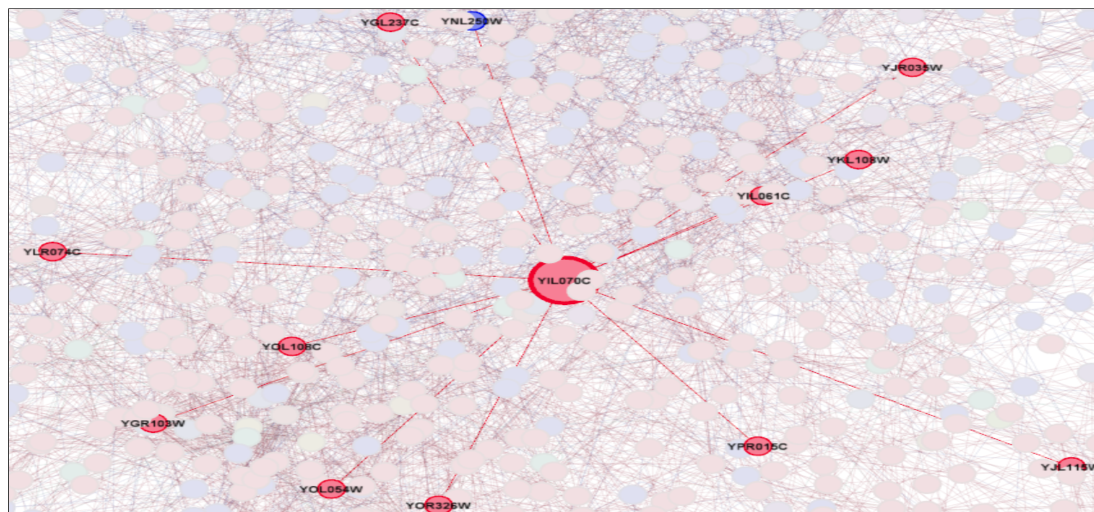

Figure 1.5: YIL070C uncharacterized protein interacts with other known proteins

We have 57 uncharacterized proteins in our data set and we shall make predictions based on the communities detected by BLLP algorithm. We check the correctness of these predictions against the MIPS database, to determine the accuracy of our results. We determine the functional modules of uncharacterized proteins, by checking its interactions with known proteins. We assign the uncharacterized protein the functional module of that of majority of the known proteins that it interacts with belonging to the same community. For example, YIL070C interacts with YGL237C, YIL061C, YKL108W, YJR035W, YOL108C, YOL054W, YPR015C, YJL115W, etc., where the majority of these have a functional module r-RNA, so we assign YIL070C

the functional module r-RNA. In case of a tie, we assign any one of the functional modules randomly to the uncharacterized protein. It is possible for a protein to have multiple functional modules, but our algorithm does not resolve the issue of overlapping communities in the network. We mainly focus on finding one strong functional module for an uncharacterized protein. We follow the same procedure for all the uncharacterized proteins and assign them a functional module based on same rule. We then verify our functional module assignment using the MIPS database/ ground-truth dataset, making 46 out of 57 correct predictions i.e. accuracy of 80% for PPI yeast dataset. Functional modules of 11 out of 57 uncharacterized proteins are incorrectly predicted. In Table 1.3, we have shown 11 uncharacterized proteins whose functional modules are incorrectly predicted by BLLP algorithm and Table 1.4 contains 46 uncharacterized proteins whose functional modules are correctly predicted, by BLLP.

No.	**Protein Pin**	**BLLP assigned modules**	**Correct modules**
1	YKL155C	r-RNA processing	Ribosome biogenesis
2	YNL177C	r-RNA processing	Ribosome biogenesis
3	YMR158W	r-RNA processing	Ribosome biogenesis
4	YPL012W	Ribosome biogenesis	r-RNA processing
5	YPR144C	Ribosome biogenesis	r-RNA processing
6	YNL002C	Ribosome biogenesis	r-RNA processing
7	YJL069C	Ribosome biogenesis	r-RNA processing

8	YGL111W	Ribosome biogenesis	r-RNA processing
9	YLR222C	Ribosome biogenesis	r-RNA processing
10	YDR428C	Splicing	Lipid oxidation
11	YLR186W	Splicing	r-RNA processing

Table 1.3: Incorrect prediction of functional modules made by the BLLP algorithm, for uncharacterized proteins

No.	**Protein Pin**	**BLLP assigned modules**
1	YGR263C	Catabolism
2	YBL004W	r-RNA processing
3	YPR034W	r-RNA processing
4	YDR449C	r-RNA processing
5	YIL070C	r-RNA processing
6	YER126C	r-RNA processing
7	YGR128C	r-RNA processing
8	YGR145W	r-RNA processing
9	YHR196H	r-RNA processing
10	YHR088W	r-RNA processing
11	YHR052W	r-RNA processing
12	YKR060W	r-RNA processing

13	YMR093W	r-RNA processing
14	YOR001W	r-RNA processing
15	YDR517W	Lipid oxidation
16	YOR093C	Lipid oxidation
17	YBL055C	Lipid oxidation
18	YDL193W	Lipid oxidation
19	YNL026W	Lipid oxidation
20	YGL211W	Protein binding
21	YGL059W	Protein binding
22	YMR310C	Ribosome biogenesis
23	YMR117C	Ribosome biogenesis
24	YMR074C	Ribosome biogenesis
25	YIL093C	Ribosome biogenesis
26	YDR036C	Ribosome biogenesis
27	YGL129C	Ribosome biogenesis
28	YJR014W	Ribosome biogenesis
29	YGR156W	Ribosome biogenesis
30	YGR285C	Splicing
31	YDL209H	Splicing
32	YKL018W	Splicing
33	YKL059C	Splicing

34	YLR424W	Splicing
35	YPL151C	Splicing
36	YGR278W	Splicing
37	YNL123W	t-RNA synthesis
38	YDR428C	t-RNA synthesis
39	YJR008W	t-RNA synthesis
40	YJL046W	Tricarboxylic acid pathway
41	YNL168C	Tricarboxylic acid pathway
42	YLR421C	Nuclear degradation
43	YPL252C	Nuclear degradation
44	YDR018C	m-RNA processing
45	YLR074C	m-RNA processing
46	YKR081C	m-RNA processing

Table 1.4: Correct prediction of functional modules made by the BLLP algorithm for uncharacterized proteins

1.6.3 Comparative Study

In this subsection, we compare the results obtained by applying the following three community detection algorithms to PPI yeast network, to that obtained by BLLP.

1. Fast unfolding of communities in large networks (FC) ([10]), which is a heuristic method based on modularity optimization.

2. Label Propagation Algorithm for Community Detection (LPA) ([76]), uses label propagation method, without the bi-level frame-work used in BLLP.

3. Generalized Louvain Method for Community Detection in Large Networks (LM) ([19]) also uses concept of network modularity optimization, while exploiting the measure of edge centrality .

Algorithm	**Modularity**	**Time (secs)**	**No. of communities**
FC	0.581	2.56	115
LPA	0.10	24.23	94
LM	0.546	7.72	111
BLLP	**0.592**	**6.23**	**123**

Table 1.5: Comparing various community detection algorithms on PPI yeast network

The computational time taken and quality of the results in terms of modularity obtained by applying the above three algorithms on PPI yeast network is given in Table 1.5. FC algorithm gives 115 communities with the largest community having 250 proteins (Figure 1.6(a)). The PPI networks has many low-shell proteins, which

increases the size of network. They generally comprise of one big component and several small components. So the idea to divide the networks based on modularity optimization doesn't yield good results for the PPI yeast network. The grouping made by FC are highly inaccurate. In the largest community with 250 proteins, 60% of proteins has functional module of r-RNA processing, 22% has functional module of ribosome biogenesis, and the remaining proteins has multiple mixed functional modules. LM algorithm behaves quite similar to FC, since both these algorithms are based on the same concept of modularity optimization. Whereas LPA algorithm gives 94 communities (Figure 1.6(b)), but there is an *epidemic spread* which gives rise to a community of size 2224 proteins. This results in low modularity communities and incorrect grouping of proteins. To avoid this situation BLLP takes care that *epidemic spread* does not take over the network.

In order to detect functional modules in PPI yeast network, one should not just focus on modularity maximization community detection algorithms. More novel community detection techniques are required for correct groupings of proteins in these networks. Novel techniques involving pre-processing the original network based on its topological features proves promising. Also, multi-level algorithmic frame-works have a better scope to achieve accurate clustering results on biological networks, thus making accurate predictions. Since we are just comparing the various algorithms on PPI medium sized yeast dataset, computational time does not play an important role.

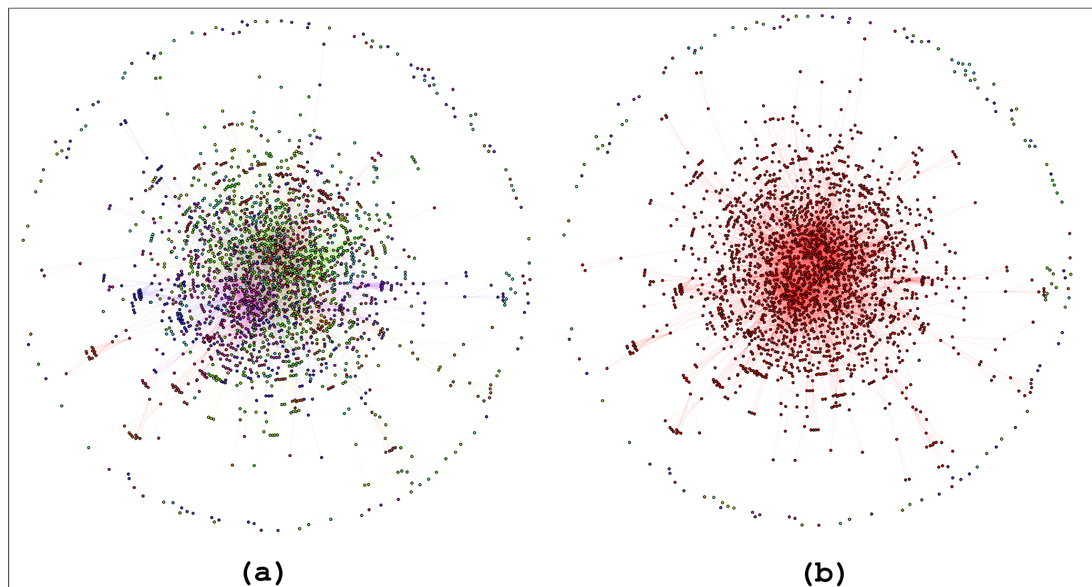

Figure 1.6: (a) FC clustering with largest community having 250 proteins (b) LPA clustering with largest community having 2224 proteins

1.7 Discussion

This research [65] focuses on matching groups of proteins which are more likely to be part of the same functional modules. Using our BLLP algorithm we achieve more accurate groupings of proteins in less computational time. We show that the predictions by the BLLP algorithm of the functional module of uncharacterized protein is also highly accurate. Our computational analysis also proves that our algorithm extracts higher modularity communities when compared to other well known community detection algorithms. Compared to the state of the art community detection algorithms, the computational time is also close to the best. The BLLP algorithm and other sequential algorithms mentioned above are designed to find community structures on small, and medium sized datasets. These algorithms prove computationally

very expensive to use on large networks. Hence we require parallel programming models to design scalable algorithms to tackle large volume of data.

CHAPTER 2
PARALLEL PROGRAMMING MODELS

Most of the sequential algorithms are computationally very expensive, and hence impractical to use on large real world datasets. Tackling large volume of data requires parallel programming models to achieve scalable algorithms. In this section, we introduce the state of the art parallel architectures, and programming models which are scalable to large real world datasets having size upto few Tera-bytes. Overview of parallel computing for graph analysis is given in ([54]). A detailed survey of parallel programming models is given in ([37]). There are six qualitative criteria to evaluate parallel programming models :

1. *System Architecture :* The two main architectures are 'shared memory' and 'distributed memory'. Shared memory architecture refers to systems like Symmetric Multi Processing (SMP)/Massive Parallel processing (MPP) nodes, where all processors share a single address space. Applications with such models can only utilize processors within a single node. Whereas distributed memory architecture refers to systems such as a cluster of compute nodes, where there is one address space per node, and applications developed with these models can run on multiple nodes as well.

2. *Worker Management :* It guides the creation of worker threads, and processors. Worker management is 'implicit' when programmers don't have to manage the lifetime of workers i.e., they only need to specify the number of

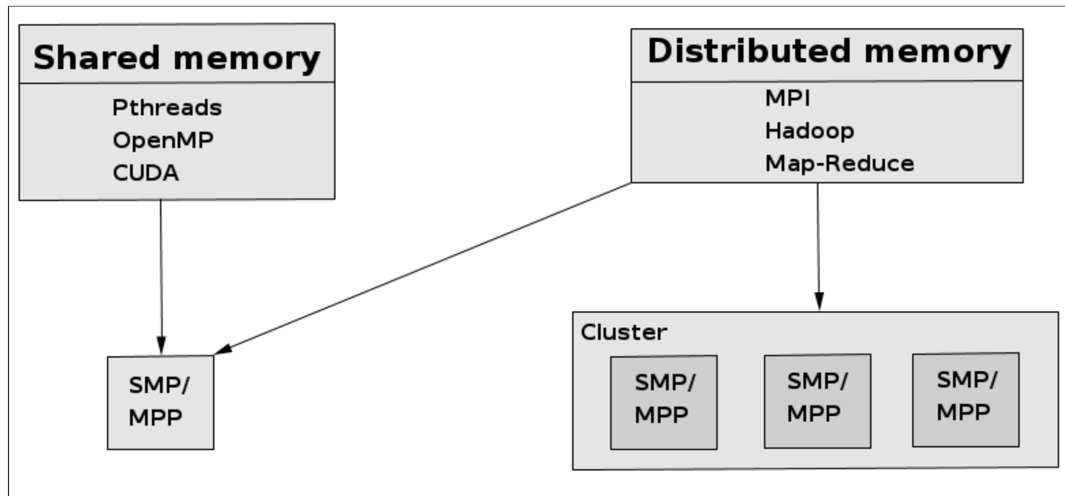

Figure 2.1: Programming models and their supported system architecture

threads/processors required for the section of code to be run in parallel; whereas in the 'explicit' approach, programmers need to code the creation and destruction of threads/processors.

3. *Task Partitioning :* It defines how the workload should be divided into smaller chunks called tasks. In the implicit approach the programmer just needs to specify whether workload can be processed in parallel. The partition of the workload into tasks need not be managed by the programmer. Whereas in the explicit approach, programmers need to manually decide how the workload should be partitioned into tasks.

4. *Task Mapping :* It defines how tasks are mapped onto threads/ processors. In the implicit approach, programmers do not need to specify which thread is responsible for a particular task; whereas in the explicit approach programmers

are required to manage how tasks are assigned to workers.

5. *Synchronization :* It defines the order and time in which threads/ processor access shared data. In the implicit approach, either there is no synchronization constructs needed, or its enough to just specify that synchronization is needed. Whereas in the explicit approach, the programmers need to manage the thread's access to the shared data.

6. *Communication Model :* It specifies the model of communication used. Ex. message passing model, shared address space, etc.

2.1 P-threads

Portable Operating System Interface (POSIX) also known as P-threads, is a set of C programming language types and procedure calls ([15]). `pthread.h` contains the implementation of P-threads and can be used as a library to create and manipulate threads/ processors. Worker management in P-threads requires the programmer to explicitly create threads using `pthread_create` and destroy threads using `pthread_exit`, functions. `pthread_create` requires the following parameters : (a) the thread used to run tasks, (b) attributes, (c) tasks to be run by thread in routine call, and (d) routine argument. The thread will terminate as soon as it encounters `pthread_exit`. Task partitioning, and mapping are specified by the programmer, as arguments to the `pthread_create` function, in the third passing parameter i.e. routine call, and first passing parameter respectively. A thread can join other threads using `pthread_join`. When this function is called, the calling thread will pause its execution until the target

thread finishes, before joining the threads. When multiple threads access the shared data, programmers need to look for data race and deadlocks conditions. To protect a critical section, P-threads provides mutex (mutual exclusion) and semaphore ([3]). Mutex allows only one thread in the critical section at a time, whereas semaphore allows multiple threads to enter a critical section.

2.2 OpenMP

It is used for shared memory parallelism ([9]). It provides a set of compiler directives, runtime libraries and environment variables that extend Fortran, C and C++ programs. OpenMP is portable across the shared memory architecture. The worker management is implicit, and special directives are used to specify which section of code is to be run in parallel. The number of threads to be used is specified using an environment variable. Here there is no need for programmers to manage the lifetime of threads. Task partitioning, and mapping requires relatively few programming effort, with programmers just having to specify compiler directives to denote a parallel region, namely `#pragma_omp_parallel{}` for C/C++. OpenMP also abstracts away how a task is divided into sub-tasks, and how these sub-tasks are assigned to threads. OpenMP supports implicit synchronization where programmers specify only where synchronization occurs, and not worry about actually dealing with synchronization mechanism.

2.3 CUDA

Compute Unified Device Architecture (CUDA) is C/C++ programming language designed to support parallel processing on Nvidia Graphics Processing Unit (GPUs) ([38]). CUDA views a parallel system as consisting of a host device (CPU), and computation resource (GPU). Set of threads in GPU run in parallel to perform all the computations. The GPU architecture for threads consist of two-level hierarchy, namely *block* and *grid*, (Figure 2.2).

Function	**Description**
`Barrier`	Allow synchronization on all threads within the same group
`Atomic`	Allow all threads execute, but only one thread of load, or store at a time
`Ordered`	Allow the block of code to be executed sequentially
`Flushed`	Ensure all threads have a consistent view of certain objects in memory

Table 2.1: Synchronization Functions in CUDA

Block is a set of tightly coupled threads where each thread is identified by a thread ID, while grid is a set of loosely coupled blocks with similar size and dimension. Worker management in CUDA is done implicitly, so the programmer just has to specify the dimensions of the grid and block required to process a certain task. Task partitioning and mapping in CUDA is done explicitly. Programmers have to define the workload to be run in parallel by using `GlobalFunction$<<<$dimGrid,`

`dimBlock$>>>$(Arguments)`, where (a) `GlobalFunction` is the function call to be run in threads, (b) `dimGrid` is the dimension and size of the grid, (iii) `dimBlock` is the dimension and size of each block and (iv) `Arguments` represent the passing value for the global function. The task mapping in CUDA programming is defined in `$<<<$dimGrid,dimBlock$>>>$`. Synchronization for all threads is done implicitly through function `syncthreads()`, which coordinates the communication among threads of the same block. Some synchronization functions along with their description is given in Table 2.1.

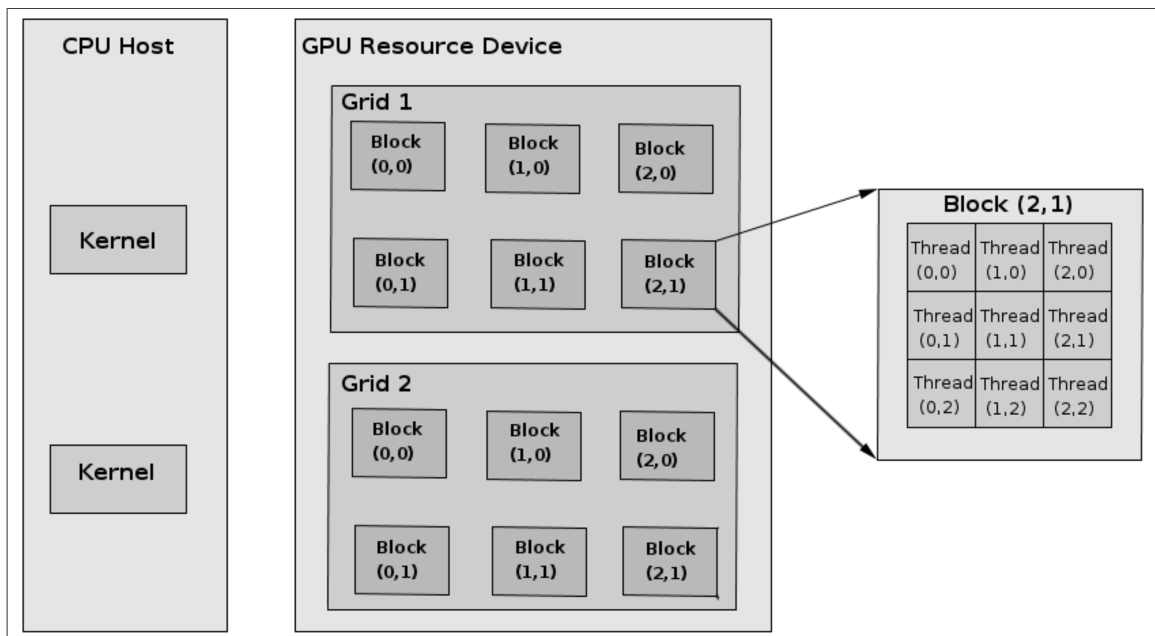

Figure 2.2: GPU Architecture

2.4 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a specification for message passing operations ([67]). MPI is currently the widely used standard for developing HPC applications on distributed memory architecture. It provides language bindings for C, C++, and Fortran. Each thread is defined as a process in MPI. Some of the well-known MPI implementation includes OpenMPI, MVAPICH, MPICH, GridMPI, and LAM/MPI. Worker management is done implicitly in MPI, so programmer only needs to use a command *mpirun*, to determine the number of processors needed, and to (optionally) map the tasks to processors. Task partitioning and mapping have to be done by programmers. Communication among processes adopts the message passing paradigm where data sharing is done by one process sending the data to other processes. Message passing operations in MPI are classified as *point-to-point* and *collective*. Point-to-point operations such as the `MPI_Send`/`MPI_Recv` pair facilitate communications between processes, whereas operations such as `MPI_Bcast` facilitate communications involving more than two processes. `MPI_Barrier` specifies the need for synchronization. It blocks each process from continuing its execution until all processes have entered the barrier. It is used to verify whether global data is distributed to appropriate processes.

2.5 Map-Reduce

Map-Reduce is a framework for processing massive data sets on certain kinds of distributable problems using a large number of nodes, collectively referred to as

a cluster. It offers clean abstraction between data analysis task and the underlying systems challenges involved in ensuring reliable large scale computation. Distributed File System (DFS) is the primary storage system used by Map-Reduce applications. It creates multiple replicas of data blocks and distributes them on compute nodes throughout a cluster, to enable reliable and rapid computations.

Map-Reduce computations are defined by the developed `map` and `reduce` functions mentioned in [78]. It executes as follows (shown in Figure 2.3): In the 'Map task', initially the *Input Reader* divides the input data into appropriate size splits and assigns one split to each map function. The 'Map tasks' converts the data into *key-value pairs*, based on the code written by the programmer for the `map` function. Multiple such `map` functions running in parallel on the data that is partitioned across the cluster. Key-value pairs are collected by these 'Map Tasks' and sorted by keys. These keys are divided and distributed to the 'Reduce tasks', leading to all the key-value pairs with same key to fall in the same 'Reduce task'. These 'Reduce Tasks' work on one key at a time, combining all the values associated with them in a way determined by the programmer in the `reduce` function. It then writes the output, which is much smaller than the input file, to the DFS or other databases. The 'Reduce Tasks' can run on one or more processor at a time.

Map Processing : The main job of a Map-Reduce programmer is to program *map* and *reduce* functions, such that *map* function outputs key-value pairs, which are processed by *reduce* functions to generate final output. The *map* function is defined with a key-value pair as an input, representing some part of the original

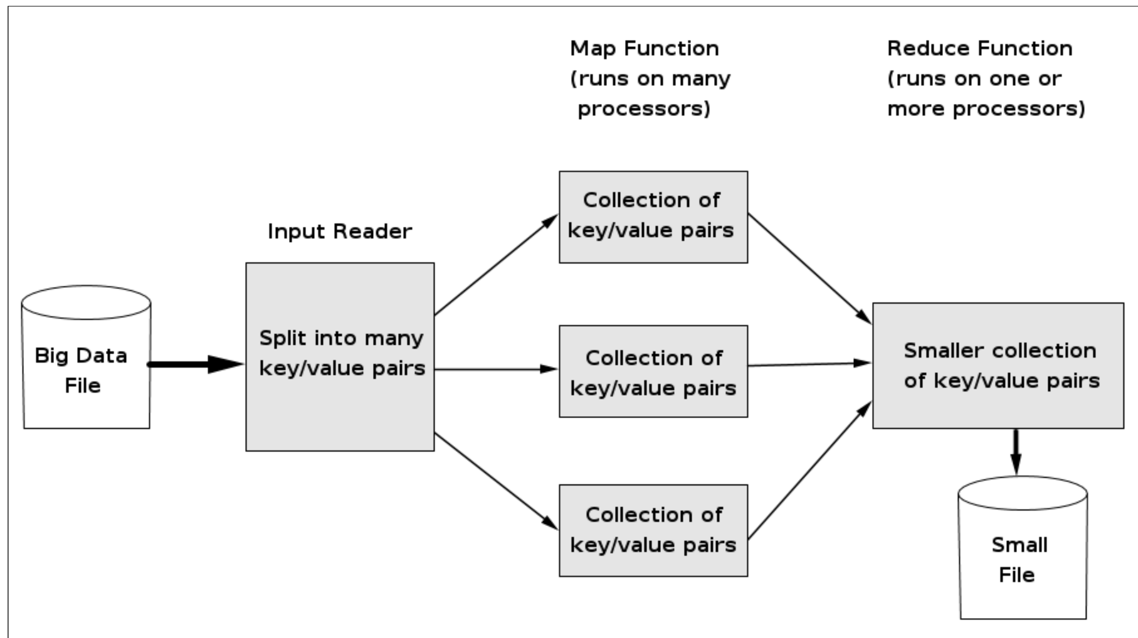

Figure 2.3: Map-Reduce Computation Semantics

file. It generates zero or more key-value pair from that input.

Reduce Processing : The *reduce* function is called for each key outputted by the *map* function. The input to the *reduce* function is all the values outputted by the *map* function for a certain key. The *reduce* function also generates zero or more key-value pair from that input and writes the output to the DFS or other database.

2.6 Hadoop

Hadoop is a framework developed for running applications on large clusters. Apache Hadoop is an open source implementation of Google's Map-Reduce methodology, where application is divided into several chunks of tasks which may be executed on any node in the cluster. Hadoop provides with Distributed File System (DFS) a

way to store data on several nodes. Node failure instances regarding Map-Reduce tasks is automatically handled in Hadoop. Map-Reduce is an infrastructure to parse and build large datasets. A `map` function creates *key-value pairs* from input data, and this data is reduced using `reduce` function that merges all values with the same keys. Parallelization and execution of programs are automated on the run time system which manages partitioning of data, scheduling, managing communication, and also recovery from failure.

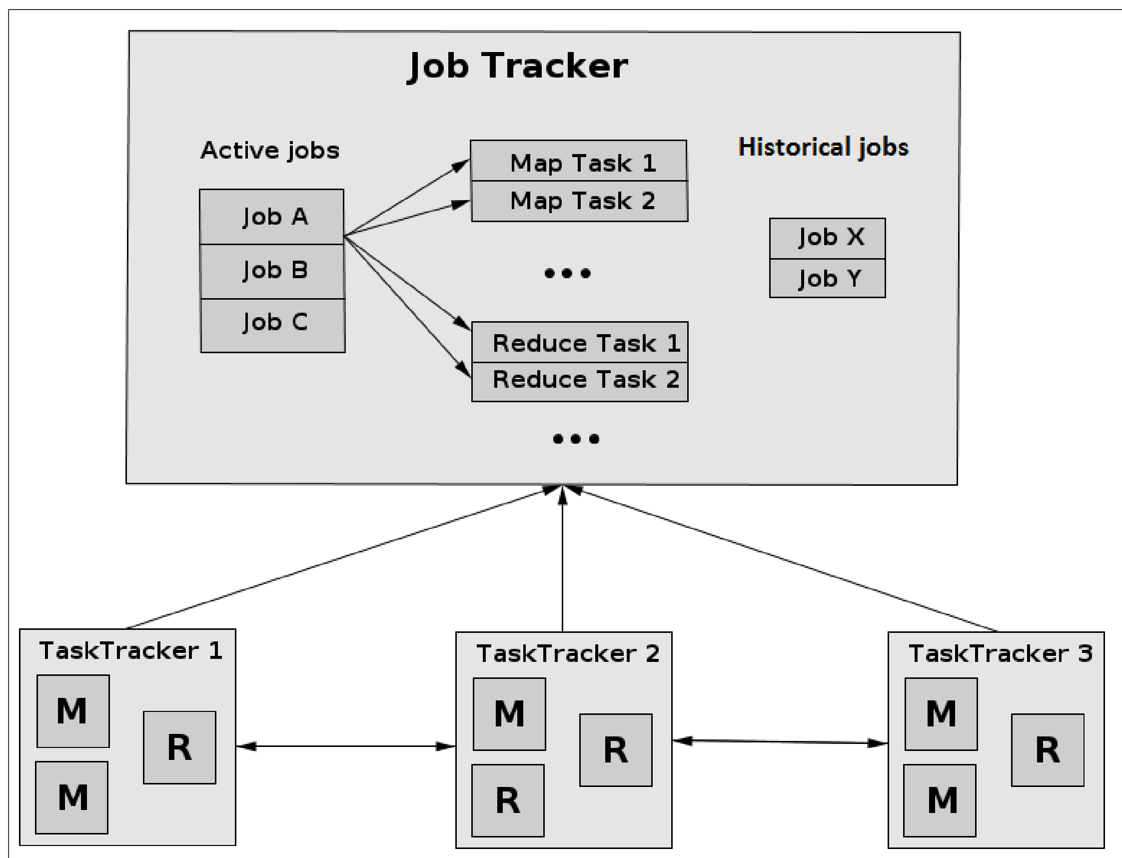

Figure 2.4: Map-Reduce Task and Job Tracking

Hadoop cluster architecture consists of nodes called *TaskTrackers* which are managed by *JobTrackers*. *JobTrackers* coordinates with *TaskTrackers* processes by accepting Map-Reduce requests from clients, and then scheduling *map* and *reduce* tasks on the *TaskTrackers*. Map functions reads the input from HDFS and outputs it on local disk, from where reduce tasks reads it, and write their outputs back to HDFS. In Figure 2.4, *TaskTrackers* transmits heart beat signals at regular intervals to *JobTrackers*, indicating when they can take a new *map* and *reduce* tasks for execution. *JobTrackers* uses a *Scheduler* to assign tasks to *TaskTrackers*, and sends this information to *TaskTrackers* along with the heart beat response.

Some of the main advantages of Hadoop are as follows: (i) Its ability to write Map-Reduce programs in high level language like, Java. (ii) Its ability to process massive data in parallel. (iii) Its ability to be deployed on large clusters of cheap commodity hardware, as opposed to expensive parallel-processing hardware. (iv) Its ability to be accessed as a on-demand service, ex. Amazon's EC2 cluster computing services. More details related to Hadoop's physical architecture, performance, portability, and available algorithms is given in [84].

CHAPTER 3
SOCIAL NETWORKS DOMAIN : SHARED MEMORY

In Chapter 1 of this thesis, we developed a BLLP community detection algorithm which was capable of tackling small to medium sized graph-structured data. But this algorithm, when used on large real world networks, is computationally expensive which makes it impractical to use. Hence we require parallel programming models, which we described in Chapter 2, to design scalable algorithms to tackle this large volume of data. In this chapter, we develop a shared memory based multi-core multi-level community detection algorithm, to tackle large volumes of graph-structured data.

3.1 Introduction

One of the most relevant and widely studied structural properties of networks is their community structure or clustering. Detecting communities is of great importance in social networks, where systems are often represented as graphs. Community detection in a network also extracts the structural properties of the network ([26]) and the various interactions in the network ([8]). There is no universally accepted definition for community detection. Hence most of the recent work in this area does not have a community structure defined in its literature, but has a quality function defined to quantify how well the network is divided into communities. So the community detection problem focus on optimizing this quality function ([60]). One of the quality functions often used is *modularity* ([61]). Different community detection

algorithms have been introduced in the past few years, which look at the problem from different perspectives. Most of these algorithms including our BLLP algorithm, however, have expensive computational time which makes them impractical to use for large networks found in the real world.

The remainder of this chapter is organized as follows: In Section 3.2, we describe the related work. We state our contributions in Section 3.3. In Section 3.4, we describe various stages of the MCML algorithm along with its parallel implementation. In Section 3.5, we describe computational results of applying MCML algorithm on two benchmark datasets (Karate club and Dolphin club), followed by large datasets like Facebook forum and Amazon product network, along with its comparative study. We finally end this chapter with some discussion in Section 3.6.

3.2 Related Work

Community detection is an interesting problem in the domain of *graph partitioning*. Interest in community detection problem started with the new *partitioning* approach by ([26]),([61]); where the edges in the network with the maximum betweenness are removed iteratively, thus splitting the network hierarchically into communities. Similar algorithms were proposed later on, where attributes like 'local quantity' i.e. number of loops of a fixed length containing the given edge ([75]) and a complex notion of 'information centrality' ([25]), is used to decide removal of edges. *Hierarchical clustering* is another major technique used for community detection, where based on the similarity between the nodes, an agglomerative technique

iteratively groups vertices into communities. There are different existing methods to choose the communities to be merged at each iteration. ([60])([91]) developed an algorithm which starts with all the nodes as individual communities and iteratively merges them to optimize the 'modularity' function. Many other algorithms in the literature of community detection, like one proposed by ([19])([28]), rely heavily on *modularity maximization*. *Label propagation* is another well known technique used for community detection, which finds communities by iteratively spreading labels across the network. Raghavan et al.([76]) proposed an algorithm where each node picks the label in its 1-neighborhood that has the maximum frequency. These labels are permitted to spread synchronously and asynchronously across the network until near stability is attained in the network. This method has some limitations, where large communities dominate the smaller one's in the network, this phenomenon is called 'epidemic spread'. This limitation was resolved by ([65]). Liu et al.([48]) used affinity propagation, which is a similar approach to label propagation, for finding communities/clusters in images. The algorithm we propose uses label propagation ideas and also prevents 'epidemic spread' in the network, thus avoiding extremely large communities that dominate the entire network. Some community detection algorithms use *random walks* as a tool. The idea is that, due to the higher density of internal edges, the probability of staying inside the community is greater than going outside. This approach is used in Walktrap ([69]) and Infomap ([82]) algorithms. A thorough review on community detection algorithms for networks is given in ([24]).

Community detection algorithms is a well studied research area, but achiev-

ing strong scalability along with detecting high quality communities is sill an open problem. One of the recent parallel algorithms developed to detect disjoint community structures based on maximizing weighted network partitioning is given in ([69]). Recently, ([88]) proposed a scalable parallel algorithm for community detection, based on label propagation, which is optimized for GPGPU architectures. This algorithm just works on local information which drives the high scalability of this algorithm. ([71]) proposed a scalable community detection algorithm, which partitions the network by maximizing the Weighted Community Clustering (WCC), a recently proposed community detection metric based on triangle analysis ([70]). Some other works which focus on developing parallel implementation for existing community detection heuristics is given in ([83]). We develop a shared memory based community detection algorithm, which achieves a good balance between *scalability* and *quality* of the communities discovered.

3.3 Contribution

We summarize our main contribution to this problem as follows:

1. We develop an MCML community detection algorithm which achieves a good balance between scalability and quality of the communities detected, compared to other algorithms in the current state of the art.

2. We show that the quality of the results obtained by the MCML algorithm for benchmark datasets with ground truth is highly accurate.

3. We show that applying MCML to datasets without ground truth detects com-

munities roughly as meaningful as other well known algorithms in the current state of the art and in some cases even better (Facebook forum). The comparison is done using modularity as the metric.

3.4 MCML Algorithm

In this section, we present the MCML algorithm involving a preprocessing stage, where each edge is assigned a strength based on the topology of the graph. Then based on the strength requirement of the communities, weak edges are removed and coarser graph instances are recursively created by identifying and removing communities, using the node with highest centrality each time. We recursively apply this step until every node is assigned to a community.

3.4.1 Preprocessing: Edge Strength Assignment

The MCML algorithm finds communities in a graph $G(V, E)$ where V represents the nodes/vertices and E represents the edges between the nodes, by assigning strength to the edges initially. It is desirable to assign an edge strength value that most accurately represents the topological structure of the graph in the MCML algorithm. Since we do not have any prior knowledge of the community structure, we assign a strength value to each edge based on the significance of that edge to the other nodes in the graph, and to the nodes at the end points of that edge. For each edge $e(i, j)$ (where i and j are nodes) in the fine graph G, the topological edge strength value $\alpha(i, j)$ assigned to it is the ratio of number of triangles that edge $e(i, j)$ participates in to the total number of triangles containing node i.

If the strength value of an edge $e(i,j)$ is greater than other edges in the 1-neighborhood of i then, node i and node j are more likely to be in the same community. Whereas on the contrary, if edge $e(i,j)$ has lower strength value than most other edges in the 1-neighborhood of i, then node i and node j are less likely to be in the same community. Mathematically,

$$\alpha(i,j) = \frac{t_{(i,j)}}{\sum_{(i,k)}(t_{(i,k)})} \quad ; \quad k \in N_i \tag{3.1}$$

where N_i is the 1-neighborhood of i, and $t_{(i,j)}$ is the total number of triangles whose sides contain edge (i,j).

The MCML algorithm also works well with weighted graphs, where the edges are assigned weights w_{input} as an input. To get the total weight of an edge, we simply have to take product of the topological edge strength value, with its input weight.

$$\alpha_{total}(i,j) = \alpha(i,j) \times w_{input}(i,j) \tag{3.2}$$

After this we normalize the edge strength for all the edges, such that they range in between 0 and 1.

3.4.2 Remove Weak Edges

The procedure of removing the weak edges from the fine graph G is based on the required strength β of the communities. Edges with $\alpha(i,j) < \beta$ are removed. This simply means that, for low values of β, fewer edges will be removed from the fine graph G, as compared to when β has higher values. After deleting these edges we might label some nodes as *non-community nodes* i.e., the degree of these nodes

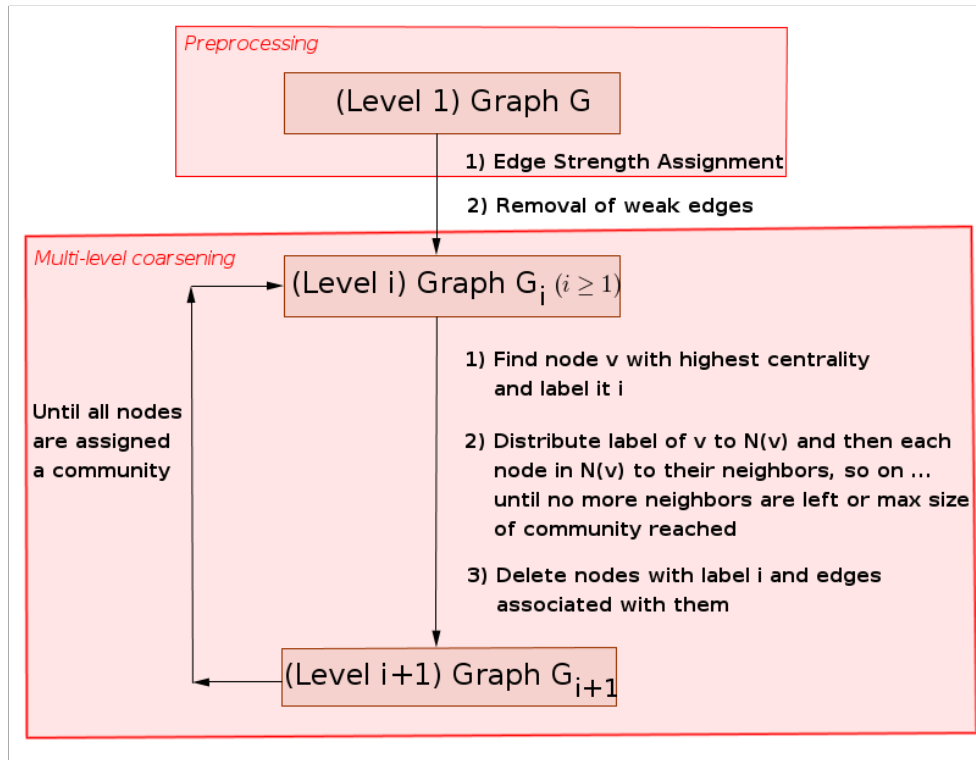

Figure 3.1: MCML Algorithm : General Schema

is zero. For higher values of β we get a higher number of non-community nodes and more stronger, smaller and significant communities are extracted. Whereas for lower values of β we get smaller number of non-community nodes, and higher number of nodes are assigned a community.

3.4.3 Multi-level Coarsening

Let G_i $(i \geq 1)$, be the graph obtained by removing weak edges (i.e., $\alpha(j, k) < \beta$) from G. We apply the following coarsening step recursively to extract meaningful community structure.

Multilevel coarsening : We select a node v from G_i, having highest centrality and

label it i. Now we distribute this label i assigned to v, to all the neighbors of v, denoted by $N(v)$. We continue distributing labels to the neighbors of all the nodes with label i. We do this until there no more neighbors are left to send the label, or the maximum required size of the community is reached. Then we obtain a coarse graph G_{i+1} by removing all the nodes with label i from G_i, along with their associated edges. We continue this process recursively until all the nodes are assigned a community. The communities having number of nodes less than the *minimum number* required in a community, labels all the nodes in that community as non-community nodes. The general schema for the algorithm is shown in Figure 3.1.

3.4.4 Parallel Implementation

Parallel shared-memory based, multi-core implementation, for each stage of our MCML algorithm is described in this subsection.

Parallel Preprocessing : In the preprocessing stage, we designate a master thread, which divides the graph roughly into k equal parts, where k is the number of cores/threads available. We perform a k partition on the input graph. We can also use an existing k way graph partitioning library like KaHIP, METIS, PMETIS, etc. to divide the graph into k parts. The master thread then assigns each of these k parts to k threads individually (including itself), as shown in Figure 3.2. Then each thread computes the edge strengths in the part of the graph assigned to them, using Equation 3.1. The inter-partition edges, which are the dashed edges in Figure 3.2, are excluded in this computation. Once all the threads have completed their edge strength

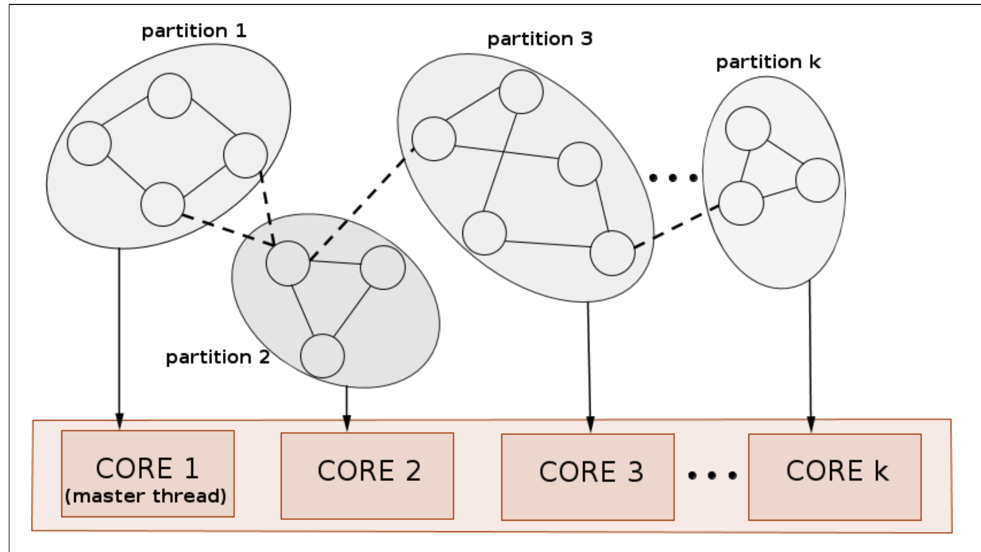

Figure 3.2: Parallel Preprocessing

assignment computations, the master thread merges the k parts of the graph back together and computes the edge strengths of the previously excluded dashed edges.

Parallel Weak Edge Removal : In weak edge removal stage of the algorithm, we again partition the preprocessed graph in to k parts and assign each part to each of the k cores/threads individually, in the similar way we did in the preprocessing stage. Each core removes the edges having strength less than threshold value (β) from the corresponding part of the graph they process. Note that, here we do not have to worry about the inter-partition edges (dashed edges in Figure 3.2) because, if they have strength less than the threshold they will be removed, else will be restored, by both the threads they are assigned to. Once all the threads have completed their weak edge removal process, the master thread merges all k parts of the graph back together.

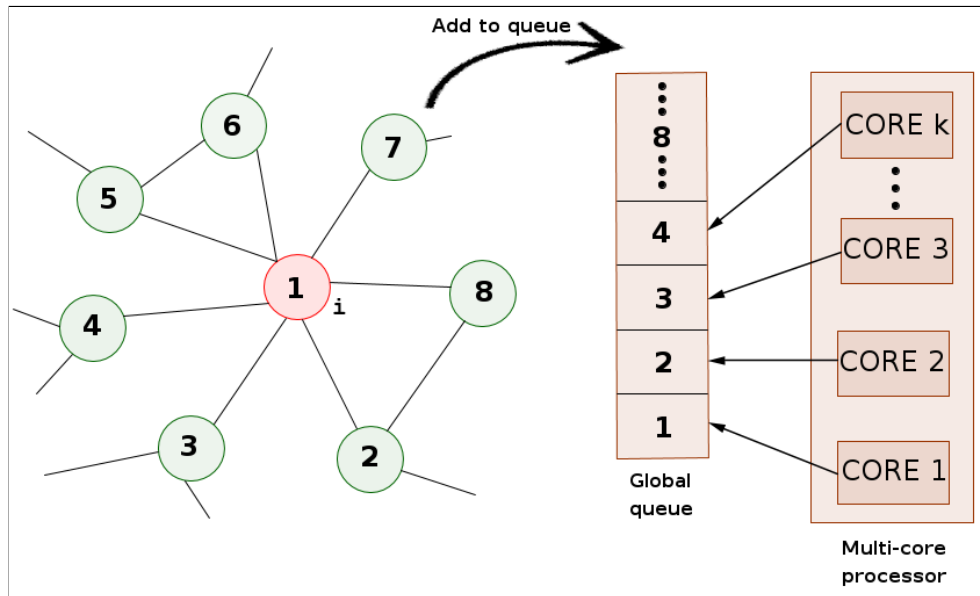

Figure 3.3: Parallel Multilevel Coarsening

Parallel Multilevel Coarsening : In the multilevel coarsening stage of the algorithm, we designate a master thread, which first finds the node with highest degree centrality in the graph and labels it i ($i \geq 1$). We then create a global queue, such that all the k cores point to the rear-end of this queue, as shown in Figure 3.3. The highest centrality node is then pushed into this global queue. The master thread is then assigned to this node, based on our construction of the queue. Then the master thread distributes label i to 1-neighborhood of this node and also add the new nodes it discovers in the 1-neighborhood to the queue. Similarly in the consequent rounds, the threads are assigned nodes from the queue as shown in Figure 3.3 and each thread distributes label i to a 1-neighborhood (nodes that have not yet received the label i) of the node assigned to it, along with adding the newly discovered nodes

to the queue. So at a given time, there are k nodes that are assigned to k cores in a cyclic fashion, which can simultaneously propagate its labels. In Figure 3.3, initially the master thread i.e. core 1 finds node 1 (red node) which is the highest centrality node and adds it to the queue after assigning label i to it. Node 1 is then assigned to core 1, which distributes label i of node 1 to the nodes in its 1-neighborhood (green nodes) which have not yet been labeled. Along with label distribution, it also add these nodes to the queue. Nodes $2, 3, 4, 5, 6, 7$, and 8 are added to the queue. In the second round, nodes $2, 3, 4$, and 5 are assigned to cores $1, 2, 3$, and 4 respectively in a cyclic fashion from the rear-end of the queue. Each of the cores then follow the same steps that was followed by core 1 in the initial round.

We place appropriate *barriers* and *write locks* to the queue in order avoid *race conditions* between threads. This process continues in cyclic fashion, until the queue is empty (disjoint component found) or maximum desired size of the community is reached. After finding a community, the k threads remove the community with label i from the graph (using a trivial *parallel for loop*) and the same process is iteratively applied on the remaining graph with label $i+1$. We continue this until the algorithm terminates, i.e. all nodes are assigned to a community. Note that we did not perform a graph partition in this stage to avoid nodes of the same community to be assigned to multiple threads. The pseudo code for the MCML algorithm is given in Algorithm 3.1.

Algorithm 3.1 : MCML Algorithm

Require: Graph $G(V, E)$, β, *max size*, *min size*

```
1: return Community of each node
2: for all thread T do
3:    Assign nodes and edges to each thread T
4:    for all Edge e(i, j) assigned to thread T do
5:       Find strength α(i, j) using Equation 3.1
6:    end for
7:    for all Edge e(i, j) assigned to thread T do
8:       if (α(i, j) < β) then
9:          Delete e(i, j)
10:      end if
11:   end for
12:   while (All nodes are assigned to a community) do
13:      for all Node assigned to thread T do
14:         Find node v with highest centrality, label it i
15:      end for
16:      while (No neighbors left or max sized reached) do
17:         Assign nodes and edges to each thread T
18:         Distribute label to the neighbors of the nodes, with label i
19:      end while
20:      for all Node assigned to thread T do
21:         Delete node with label i and associated edges
22:      end for
23:   end while
24: end for
25: return community label for each node
```

3.5 Computational Results

In this section, we describe computational results of applying MCML algorithm on two benchmark datasets (Karate club and Dolphin club), followed by large datasets like Facebook forum and Amazon product network, along with its comparative study.

3.5.1 Benchmark Datasets

We use the benchmark datasets, Karate club ([97]) and Dolphin club ([55]) to determine the *quality* of the results obtained by applying MCML algorithm. Since these two datasets have ground truth communities, we measure the quality of the results based on the *accuracy* metric i.e., the number of nodes correctly assigned by MCML algorithm to the community they actually belong to in real life. We run MCML algorithm for various values of β (0, 0.1, 0.4, 0.6, 1.0). We also have a plot showing the number of nodes marked as non-community nodes, versus different values of β. Comparison of various community detection algorithms on Karate and Dolphin club benchmark datasets is shown in Table 3.1. The comparisons are made on the basis of the number of communities detected, number of correct matches and incorrect matches of the community nodes with its ground truth communities.

Karate Club : This is a social network of friendships between 34 members of a karate club. It contains 156 edges and the dataset is unweighted and undirected. The real life known partition of this graph is into 2 groups. The group breaks down into 1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 17, 18, 22 and 9, 10, 14, 15, 16, 19, 20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34. In Figure 3.4(a), where $\beta = 0.0$ and maximum size

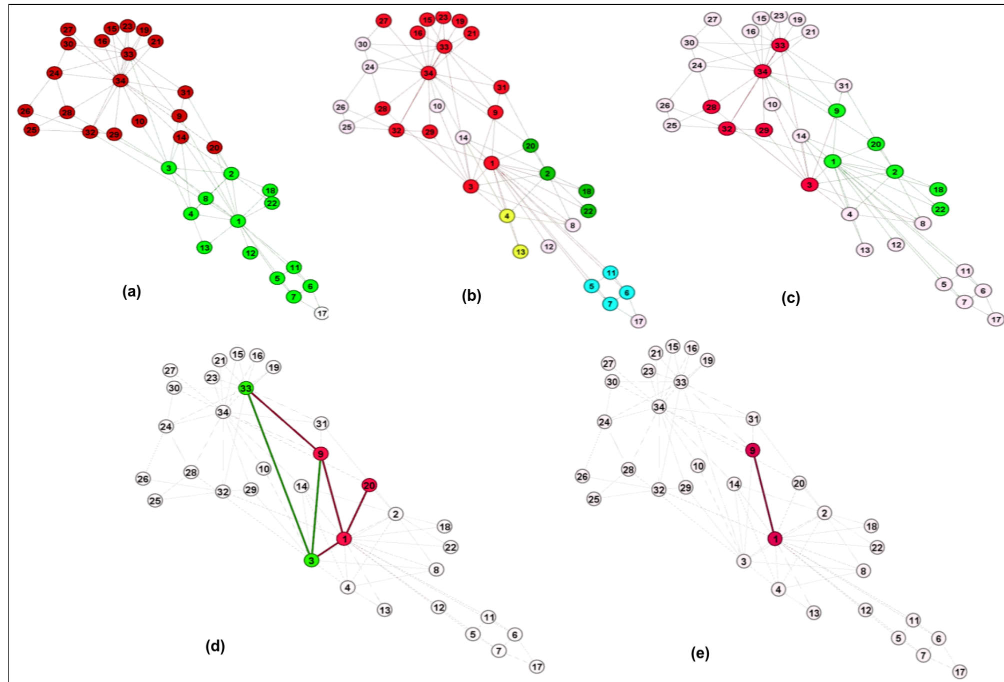

Figure 3.4: Karate Club : (a) $\beta = 0.0$ (b) $\beta = 0.1$ (c) $\beta = 0.4$ (d) $\beta = 0.6$ (e) $\beta = 1.0$ (Note: Edges highlighted in (d) and (e) have stronger connections than other edges in the graph)

= 34, when compared to the ground truth communities of this network, all the nodes are correctly grouped, except node number 17, which becomes the non-community node. All the nodes in white are non-community nodes. The nodes with different colors belong to different communities. The accuracy plot for groupings made by MCML algorithm is shown in Figure 3.6. In Figure 3.4(e), where $\beta = 1.0$, we extract the strongest link in the club, which is between node 1 and 9. In Figure 3.5(a), we can see that, as the value of β increases the number of non-community nodes increases and the more stronger and smaller communities are extracted. The edges extracted

represent stronger connections compared to edges associated with non-community nodes.

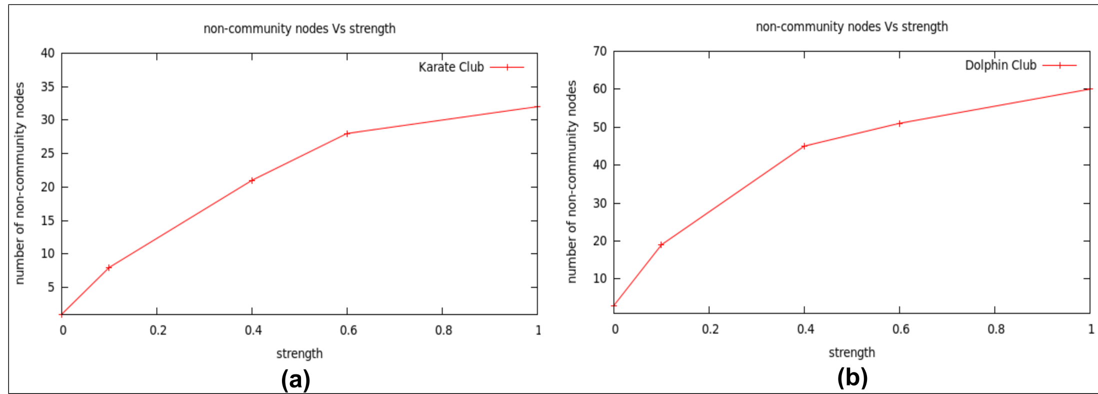

Figure 3.5: Non-community nodes vs strength : (a) Karate club (b) Dolphin club

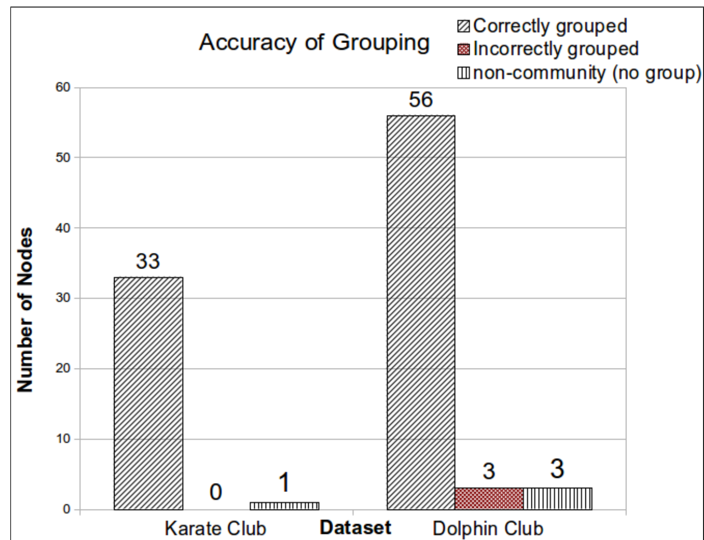

Figure 3.6: Accuracy Plot : Karate club and Dolphin club

Algorithm	**Karate club (34 nodes)**			**Dolphin club (62 nodes)**		
—	# comm.	correct	incorrect	# comm.	correct	incorrect
Jancura et al.([33])	3	24	10	3	46	16
Raghvan et al.([76])	2	27	7	4	45	17
Soman et al.([88])	2	29	5	2	49	13
MCML	**2**	**33**	**1**	**3**	**56**	**6**

Table 3.1: Comparing various community detection algorithms for Karate and Dolphin club benchmark datasets

Dolphin Club : This is an undirected social network of frequent associations between 62 dolphins (nodes) in a community living off Doubtful Sound, New Zealand. Dolphin club is an unweighted network containing 159 edges. In Figure 3.7(a), where $\beta = 0.0$ and maximum size = 62, when compared to the ground truth communities of this network, 56 nodes are correctly grouped, 3 node become non-community nodes and 3 nodes are incorrectly grouped. These 3 incorrectly grouped nodes form a new community, since they have much stronger connection amongst each other, than with community with the red label. All the nodes in white are non-community nodes. Accuracy plot is shown in Figure 3.6. In Figure 3.7(e), where $\beta = 1.0$ and we extract the strongest link in the club, which is between node 51 and 46. In Figure 3.5(b), we can see that as the value of β increases the number of non-community nodes increases.

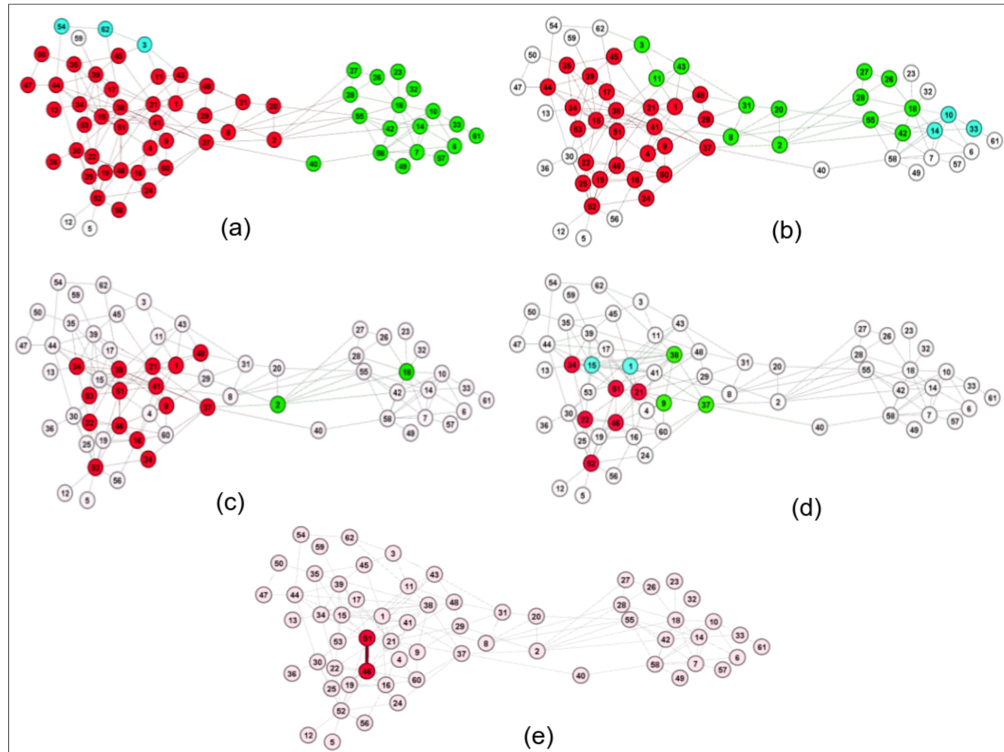

Figure 3.7: Dolphin club : (a) $\beta = 0.0$ (b) $\beta = 0.1$ (c) $\beta = 0.4$ (d) $\beta = 0.6$ (e) $\beta = 1.0$ (Note: Edges highlighted in (d) and (e) have stronger connections than other edges in the graph)

On the basis of the above experiments on the 2 benchmark datasets, we can conclude that the, higher the value of β the stronger and smaller is the extracted community.

3.5.2 Facebook Forum Dataset

Facebook Forum dataset is obtained from Facebook online social network. The main focus in this network is on user's activity in the forum. The forum represents a 2-mode network between primary nodes which are 899 *users*, and secondary nodes which are 522 *topics* in the forum. It is a weighted network where the weights represent

the number of messages a user posted on a particular topic. We use the preprocessed version of this dataset for our experiments, where 2-mode network is transformed into a 1-mode network maintaining the primary nodes which are users, and containing $142,761$ edges. This dataset is available from `http://toreopsahl.com/datasets/#online_social_network`.

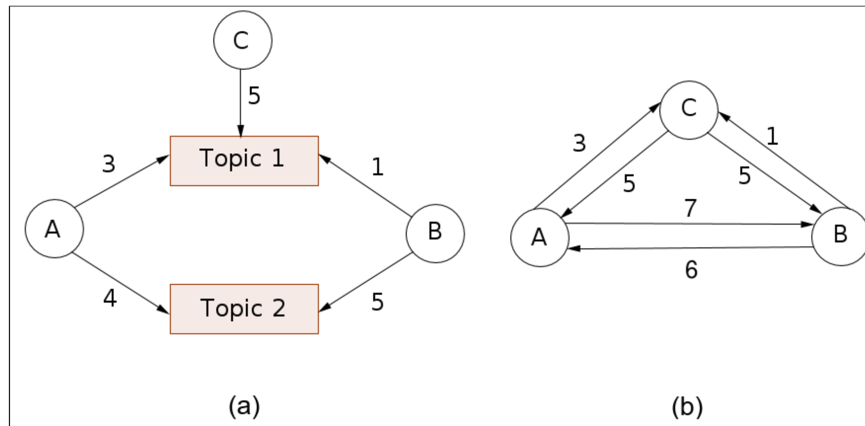

Figure 3.8: (a) 2-mode weighted network (b) preprocessed 1-mode weighted network

Figure 3.8(1), represents the format of original dataset. User A posts 3 messages on topic 1 and 4 messages on topic 2. User B posts 1 message on topic 1 and 5 messages on topic 2. When this dataset is preprocessed to eliminate the secondary nodes, we have 2 directional links between A and B, i.e., A to B which has weight 7 and B to A having weight 6, as shown in Figure 3.8(2). The 1-mode projection of a weighted 2-mode network is based on the weights the two nodes have, directed

towards common nodes. The two nodes interact with the common node, and Figure 3.8 shows how to project it onto a directed weighted 1-mode network. This data set does not have ground truth communities, so we use *modularity* to determine the quality of the communities found. The quality comparison based on modularity and computational time for Facebook Forum data set is given in Table 3.2. The modularity achieved for communities detected by MCML ($\beta = 0.15$) is 0.3566, which is the best so far in the present state of the art. So even though we do not achieve the best running times as compared to other well known algorithms like ([10]), we manage to maintain a good balance between the quality of the results and running times.

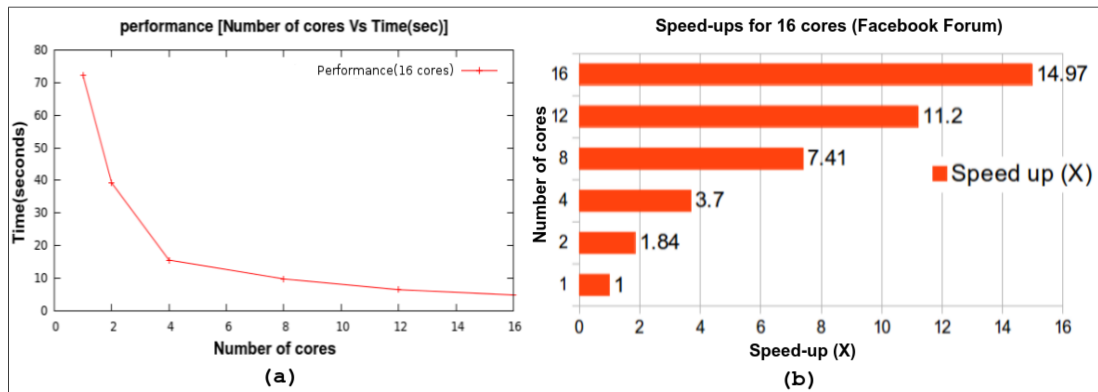

Figure 3.9: Facebook forum : (a) Computational time Vs number of cores (b) Speed-ups

Finding communities in this interesting data set implies, finding groups of users sharing similar interests. This information can be used by social networking sites to provide friend suggestions to users, or suggestions to join a particular community

forum having common interests. The performance of MCML algorithm, on this data set is shown in Figure 3.9, where we achieve speed-ups up to 14.97 times using 16 cores. This is close to a k fold improvement using k core processor where ($k \geq 1$).

Algorithm	**Facebook Forum**		**Amazon**	
—	Modularity	Time (secs)	Modularity	Time (sec)
Newman & Girvan([61])	0.0488	—	—	—
Pons & Latapy([69])	0.2031	—	0.451	—
Rosvall & Bergstorm([82])	0.1372	—	0.470	—
Raghvan et al.([76])	0.1733	47	0.210	$> 10{,}000$
Jancura et al.([33])	0.3458	3	—	—
Yang & Leskovec([95])	—	—	0.125	1890
Prat-Perez et al.([71])	—	—	0.295	15
Lancichinetti et al.([45])	—	—	0.510	4800
MCML	**0.3566**	**4.83**	**0.494**	**2389**

Table 3.2: Comparing various community detection algorithms for Facebook Forum and Amazon datasets based on modularity and computational time using 16 cores. (The blank values are not available in the literature of this research area. To get the computational time we include all the three stages of the algorithm.)

3.5.3 Amazon Product Dataset

This dataset represents a graph of products, where each node is a product and there is an edge between two products if they have been co-purchased frequently. This dataset has $334,863$ nodes and ≈ 1 million edges. This data set has a $151,037$ ground truth communities, in which top 5000 communities are the most significant. We use this dataset (`http://snap.stanford.edu/data/index.html`) in our experiments, to show that MCML algorithm gives fairly good performance and speed-up when applied to this dataset, and also, we do not degrade the quality of the results while achieving this.

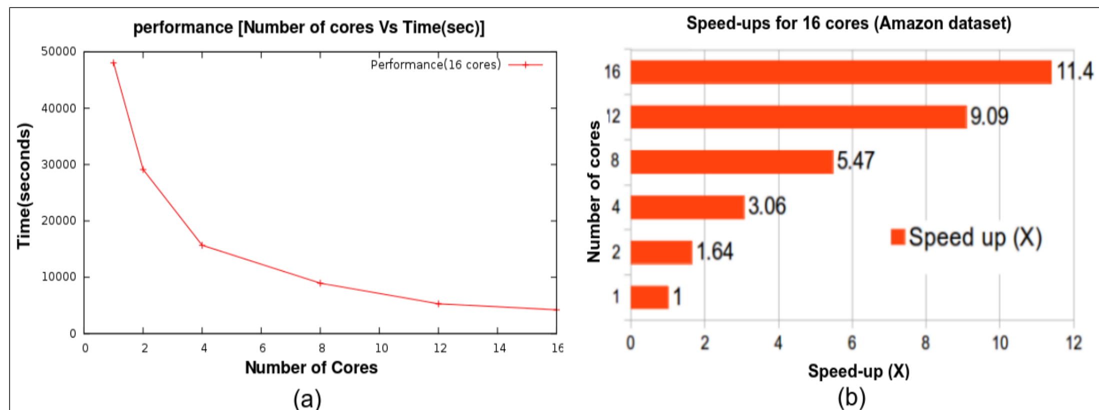

Figure 3.10: Amazon : (a) Computational time Vs number of cores (b) Speed-ups

For $\beta = 0.1$ and $maximumsize = 80$ which is the largest community in ground truth community data, it takes ≈ 7.34 hrs to extract communities in this dataset using

1 core and $\approx$ 39.82 minutes using 16 cores (i.e. speed-up of 11.4 times). In Figure 3.10, we show the time taken to find communities in this dataset for $1, 2, 4, 8, 16$ cores and corresponding speed-up's respectively. The quality comparison based on modularity and computational time for Amazon data set is given in Table 3.2. The modularity achieved for communities detected by MCML ($\beta = 0.1$) is 0.494, which is better than most of the other algorithms in the present state of art. Even though we do not achieve exceptional running times as compared to other well known algorithms like ([71]) and ([95]), we manage to maintain a good balance between the quality and running times.

3.6 Discussion

This research [64] focuses on developing a multi-core multi-level (MCML) community detection algorithm, which achieves a good balance between running times and quality of the communities discovered, which is a well known challenging problem in this area. We have shown that, the quality of the results obtained by the MCML algorithm for benchmark datasets with ground truth is highly accurate. We also compare MCML with other well known algorithms for datasets without ground truth, using the modularity metric for quality analysis, and conclude that MCML can detect communities roughly as meaningful as other known algorithms and in some cases even better (Facebook forum).

Partitioning the network into sub-networks to achieve the highest level of parallelism requires more cores. As the social networks become larger and larger the ability

to process them using shared memory architecture on one single machine becomes infeasible due to both memory and time constraints. Hence distributed memory algorithms exploiting multiple cores of multiple machines is essential to overcome this. In the next chapter, we present a novel hybrid (shared + distributed memory) parallel algorithm to efficiently detect high quality communities in massive social networks.

CHAPTER 4
SOCIAL NETWORKS DOMAIN : DISTRIBUTED MEMORY

4.1 Introduction

One of the most relevant and widely studied structural properties of networks is their community structure. A community in a network is a set of nodes that are densely connected with each other and sparsely connected to the other nodes in the network. Community detection in a network extracts the structural properties of the network ([26]) and the various interactions in the network ([8]). Detecting communities in social networks is of great importance because social networks consists of patterns which can be viewed as independent components, with each component having distinct features and can be detected based on network structure. For example, community detection in social networks can help to target users for marketing purposes, providing recommendations to users to connect with other users, join communities or forums, market basket analysis, etc.

The increasing size of social networks like Facebook, Twitter, LinkedIn, etc. has made community detection more difficult, with data size which can reach up to billions of vertices and edges. For example, Facebook has $\approx 1.1B$ users, LinkedIn has $\approx 500M$ users, etc. As a result the ability to process this large graph-structured data in memory of a single machine is infeasible due to time and memory constraints. Most of the research in community detection has been focused on sequential algorithms on SMP machines and a thorough review of the same is presented in ([24]). Where as

some fast scalable community detection algorithms ([10], [64], [76]) which have been developed can only tackle network sizes which can be stored in the RAM of one machine. All of these algorithms adopt sequential, parallel shared-memory and non-distributed architecture. Processing networks with hundreds of millions of vertices and billions of edges require several hundred gigabytes of RAM. To address this challenge, parallel distributed community detection algorithms are necessary. *To avoid any confusion, we use the term cluster only for computer cluster, a part of the computer cluster will be denoted as machine or node, the objects in a network will be denoted as vertex and groups of vertices will be denoted as communities.*

In this chapter, we modify and extend our multi-level multi-core (MCML), shared-memory based community detection algorithm ([64] we described in Chapter 3, to distributed memory parallel framework using Message Passing Interface (MPI). This hybrid (shared + distributed memory) algorithm can process massive social networks to extract high quality communities efficiently. The main challenges we encountered were (1) the initial partitioning of the network and assigning each of these parts to different nodes in the parallel computers in such a way that, when community detection algorithm is applied on each individual node, it should not incur high communication overhead, (2) each node in the parallel computers should intelligently reduce the size of the network partition assigned to it such that, after merging, the entire network should fit in memory of one machine and quality of the communities detected is not compromised.

In this work, we integrate an existing network partitioning algorithm in our

hybrid algorithm's flow such that, it will partition the original network into chunks to be distributed across the network of parallel machines, incurring minimum communication overhead between them. In order to minimize the probability of distributing vertices belonging to the same community across different machines, we use network partitioning algorithm which tries to minimize the inter-partition edges ([36]). After network partitioning and distribution, we intelligently reduce the size of every network partition on each machine in such a way that, when merging all the partitions back in the master node, the entire network can fit into the memory of a single master node to which we apply our MCML algorithm to extract high quality communities. All our simulations are done using MPI and OpenMP implementation on the HPC Neon cluster at The University of Iowa.

The remainder of this chapter is organized as follows: In Section 4.2, we describe the related work. We state our contributions in Section 4.3. In Section 4.4, we describe our hybrid community detection algorithm. In Section 4.5, we discuss and present our datasets used and results, followed by some discussion in Section 4.6.

4.2 Related Work

NETWORK PARTITIONING : It aims to divide the network into k-parts in such a way that edge cuts are minimized and each partition roughly has same number of vertices. Most of the network partitioning problems are *NP-Hard* ([24]). One group of techniques in graph partitioning relies on optimizing an objective function which is defined as a ratio of number of intra-partition edges to number of inter-partition

edges. Another group of partitioning uses multi-level partitioner ([36], [35]) whose implementation is in METIS and PMETIS library respectively. There exists many other better partitioning algorithms which scales better than METIS ([39], [56]) but we plan to utilize parallel PMETIS to perform our initial graph partitioning, due to its low communication overhead, ease of use and wide availability. The parallel implementation was implemented using GNU C++ and MPI.

COMMUNITY DETECTION : It is an interesting problem in the domain of *graph partitioning*. Interest in community detection problem started with the new *partitioning* approach by ([26], [61]); where the edges in the network with the maximum betweenness are removed iteratively, thus splitting the network hierarchically into communities. Similar algorithms were proposed later on, where attributes like 'local quantity' i.e. number of loops of a fixed length containing the given edge ([75]) and a complex notion of 'information centrality' ([25]), is used to decide removal of edges. *Hierarchical clustering* is another major technique used for community detection, where based on the similarity between the nodes, an agglomerative technique iteratively groups vertices into communities. There are different existing methods to choose the communities to be merged at each iteration. Algorithms described in ([60]) and ([91]) starts with all the nodes as individual community and iteratively merge them to optimize the 'modularity' function. Many other algorithms in the literature of community detection, like ones proposed by ([19]) and ([28]) rely heavily on *modularity maximization*. *Label propagation* is another well known technique used for community detection, which finds communities by iteratively spreading labels across

the network. Raghavan et al. ([76]) proposed an algorithm, where each node picks the label in its 1-neighborhood that has the maximum frequency. These labels are permitted to spread synchronously and asynchronously across the network until near stability is attained in the network. This method has some limitations, where large communities dominate the smaller one's in the network, this phenomenon is called 'epidemic spread'. This limitation was resolved by ([65]). Liu et al.([48]) used affinity propagation, which is a similar approach to label propagation, for finding communities/clusters in images. Some community detection algorithms use *random walks* as a tool. The idea is that, due to the higher density of internal edges, the probability of a random walk staying inside the community is greater than going outside. This approach is used in Walktrap ([69]) and Infomap ([82]) algorithms. A thorough review on community detection algorithms for networks is given in ([24]).

Parallel Community Detection : Community detection algorithms is a well studied research area, but achieving strong scalability along with detecting high quality communities is an open problem. Most of the past research on community detection has focused on single threaded algorithms. There is a rich and vast literature of such algorithms and the ones based on modularity maximization being the most prominent amongst them ([61]). The *Louvain* method which is based on modularity maximization ([10]) is the most widely used community detection algorithm which can scale to networks with millions of vertices. However, the quality of results obtained deteriorates as the size of the network increases ([44]). It is observed that modularity maximization based algorithms are unable to detect small and well-defined

communities in large networks ([59], [74]). One of the recent parallel algorithms developed to detect disjoint community structures based on maximizing weighted network partitioning is given in ([69]). A scalable community detection algorithm, which partitions the network by maximizing the Weighted Community Clustering (WCC), is proposed in ([71]) which uses community detection metric based on triangle analysis ([70]). Some other works which focused on developing parallel implementation for existing community detection heuristics is given in ([83]). Recently, ([88]) proposed a scalable parallel algorithm for community detection, based on label propagation, which is optimized for GPGPU architectures. This algorithm just works on local information which drives the high scalability of this algorithm.

Recent works mentioned above on exploitation of parallelism for community detection has the form of multi-core algorithms for SMP machines i.e. shared memory architecture. In ([53]), a parallel multi-core Louvain algorithm is proposed which exhibits the above mentioned pitfalls. In ([7]) a parallel version of Infomap is presented which relaxes the concurrency assumption of the original method ([82]), achieving parallel efficiency of 70%.

There is minimal literature work on distributed algorithms for community detection. In ([93]), a distributed memory parallel algorithm extending the Louvain method is proposed. Here, the most costly iteration of the algorithm is made embarrassingly parallel without any noticeable loss in final modularity. This approach was validated using an MPI implementation on a High Performance Computing (HPC) cluster. However, the original pitfalls of Louvain's method mentioned above and in

([44]) prevails. *Hadoop Map-Reduce Model* can be used speed up algorithms which can break down into embarrassingly parallel tasks. In ([58]), the proposed algorithm is a distributed memory parallel version of the Girvan-Newman algorithm ([61]). This version adopts the Map-Reduce framework where it breaks down the algorithm into four embarrassingly parallel tasks: (1) calculating all-pair shortest paths in the network, (2) calculating the edge betweenness for every pair of nodes in the network, (3) k-edges are selected based on edge betweenness and removed, (4) network update for next iteration. The performance results showed that elapsed time decreased almost linearly with the number of reducers. Most simple community detection algorithms which can be broken down into the embarrassingly parallel independent tasks, does not yield high quality communities on real world networks.

We propose to extend our MCML shared memory parallel algorithm ([64]), to distributed memory parallel framework using the MPI implementation on The University of Iowa's Neon HPC cluster, to detect communities in massive networks with high accuracy and attain scalability.

4.3 Contribution

We summarize our main contribution to this problem as follows:

1. We propose a hybrid (shared + distributed memory) community detection algorithm, a modification and extension of our shared memory based MCML algorithm ([64]), which utilizes multiple cores of multiple machines and scales to hundreds of millions of vertices and edges without compromising quality of

the detected communities.

2. We showcase our algorithms' efficiency by using synthetic graphs ranging from $100K$ up to $16M$ vertices.

3. We further test for efficiency and quality on real world networks like (a) section of Twitter-2010 network having $\approx 41M$ vertices and $\approx 1.4B$ edges (b) UK-2007-05 (.uk web domain) having $\approx 1.2B$ vertices and $\approx 3.2B$ edges.

4.4 Hybrid Algorithm

In this section, we describe our hybrid community detection algorithm which utilizes multiple cores of multiple machines and scales to hundreds of millions of vertices and edges without compromising quality of the detected communities. We use our shared memory based MCML community detection algorithm described in Chapter 3 and ([64]) as a subroutine in our hybrid algorithm.

We take advantage of the initial network partitioning when designing parallel distributed community detection algorithms, in order to speed up the processing time by minimizing the communication between processors. This reduces the possibility of vertices in the same community to spread across multiple partitions. We modify our parallel shared memory MCML algorithm presented in Section 3.4 of Chapter 3, to enable it to adapt the distributed MPI framework for processing massive networks. Our hybrid algorithm follows a multilevel algorithmic framework which includes the following steps (also explained with an example in Figure 4.1):

1. **Level 1 - Network Partitioning :** Using network partitioning we aim to

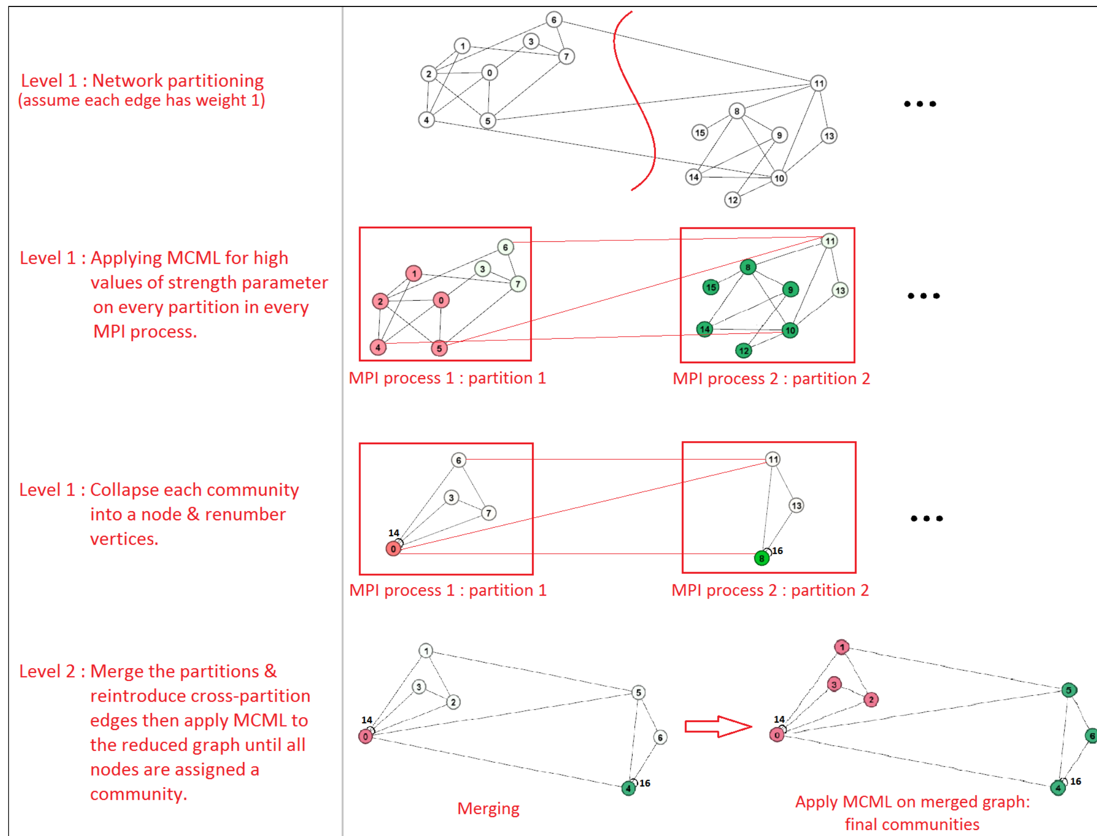

Figure 4.1: Example : Hybrid Algorithm

split the original network k-ways such that, the edge cuts between partitions are minimized and there is a balance in the number of vertices in each partition. Network partitioning is a $\mathcal{NP}$-*Hard* problem ([24]). Most of the existing network partitioning techniques use network processing tools such as Apache Giraph ([5]) which is based on hashing or vertex ordering i.e. random graph partitioning. We use parallel METIS partitioning algorithm (PMETIS) ([35]) due to its low communication overhead, ease of use and wide availability. We use the k-way partitioning library which divides the network based on minimum edge cuts.

The parallel implementation was developed using GNU C++ and MPI.

2. **Level 1 - Apply MCML on Partitions :** Each MPI processor is assigned a partition based on the process id, and MCML is applied locally on each processor. Interprocessor communications is allowed using message passing interface across cross-partition edges. MCML is applied for higher value of the strength parameter $\beta \geq 0.8$, where $0 \leq \beta \leq 1$, such that all nodes are not assigned to a community, but nodes with stronger affiliations are put in the same community. After finding these strong communities in each of those partitions we collapse each community into a single node such that, all the intra cluster edges will be represented as a self-loop on that node. Note that we do not collapse nodes residing in the partition on a different machine. This is the most crucial step where we expect to reduce the size of each partition considerably.

3. **Level 1 - Renumber Vertices and Merge Partitions :** Next step is to merge all the partitions together at the master MPI process. Since all the individual vertices have local numbering, we are required to renumber all the vertices across all partitions in a continuous fashion. We use the following method to renumber vertices before the merging step. Using the *all gather* operation in MPI, each process collects the total number of vertices every other process has. Each MPI process p_i, now has a list of total number of vertices in every other partition $\{N_0, .., N_{i-1}, N_{i+1}, .., N_{\mathcal{P}-1}\}$ where, $\mathcal{P}$ is the total number of MPI processes. It then renumbers its vertices in a way that the ones associated to its

partitions start from n_{start_i} which is based on the values of all processes p_j with $j < i$ as follows :

$$n_{start_i} = \sum_{0}^{i-1} N_j$$

Once the renumbering is performed, each MPI process sends its partitions to the master MPI process where the merging takes place.

4. **Level 2 - MCML :** The merged network represents the level 2 of the original network where the size of the network is reduced significantly. We then apply MCML algorithm again on this level until all the vertices are assigned a community. This step can be performed on a single machine i.e., master MPI process, since the size of the graph is reduced significantly and can completely loaded in to the memory.

4.5 Computational Results

In this section, we describe computational results of applying our hybrid algorithms on synthetic datasets and also massive portions of two real world social networks datasets (Twitter-2010 and UK-2007-05 (.uk web domain)). The performance of our hybrid algorithm is evaluated by executing series of experiments on the High Performance Neon Cluster at University of Iowa. We use 8 heterogeneous standard machines each having $64GB$ RAM and 16 Xeon Phi cores. All the experiments were executed as a single batch command comprising of at most 8 compute machines having 16 cores each. Each experiment is executed three times and average of the results from these runs are reported to preserve accuracy and consistency.

No. of vertices	No. of edges
100,000	422,015
500,000	1,652,471
1,000,000	3,559,759
2,000,000	6,995,154
4,000,000	14,598,778
8,000,000	32,115,764
16,000,000	63,221,980

Table 4.1: Random graph datasets

4.5.1 Datasets

We generate random graphs for our empirical studies to have control over the graph sizes and study the scalability of our algorithm over different graph sizes. We generate these random undirected and unweighted graphs using the same benchmarking package used by Fortunato in ([24]). Graph generation using this package also offers fine control over the average and maximum degree distribution, etc. Many similar studies use this package for their empirical studies ([93]). The properties of the seven graphs we generated are shown in the Table 4.1

For our empirical studies we also use massive portions of two real world social networks described in Table 4.2:

(a) **Twitter-2010 :** Twitter is a website, owned and operated by Twitter Inc.,

Datasets	**(a) Twitter-2010**	**(b) UK-2007-05**
No. of vertices	$41,652,230$	$105,896,555$
No. of edges	$1,468,365,182$	$3,301,876,564$
Edge-list file size	52.3 GB	110 GB
Average degree	35.253	35.7

Table 4.2: Real world social network datasets

which offers a social networking and microblogging service, enabling its users to send and read messages called tweets. Tweets are text-based posts of up to 140 characters displayed on the user's profile page. This is a crawl done in ([43]). Every node represents a user and there is an edge from node x to node y if x is a follower of y, i.e. edges follow the direction of tweet transmission. This dataset is publicly available from `http://law.di.unimi.it/datasets.php`.

(b) **UK-2007-05 :** This web based graph is crawled by Boldi et.al. [11]. The web-graphs of the 12 snapshots from each of the 12 months of .uk domain have been merged into a single graph. Each node represents a URL and there is an edge between URL x and URL y if the web page of URL x contains URL y. This dataset is publicly available from `http://law.di.unimi.it/datasets.php`.

4.5.2 Evaluation

Our empirical studies focuses mainly on analyzing the scalability of our hybrid algorithm and the quality of the results obtained. We scale the problem by increasing the graph size and the number of processor cores. We run the performance analysis which includes the step 2, 3 and 4 of the hybrid algorithm described in Section 4.4.

In Figure 4.2, we see the variation in the total runtime while scaling up the number of processor cores for different graph sizes. We can observe that our algorithm exhibits high scalability for all possible permutations of graph size and processor cores. The graph with $16M$ vertices could only be tested when 16 or more processor cores are used, due to memory constraints of one machine. In Figure 4.3, we see that our algorithm achieves $\approx 6X$ speedups for synthetic graphs upto $8M$ vertices. We also see that speedups flatten and start declining for most of the graphs after scaling them past 64 processor cores. This is mainly due to Amdahl's law and increase in communication overhead.

In Figure 4.4, we observe that the gap between runtime of parallel implementation with varying processor cores increases as the graph size increases. This shows the high scalability of our hybrid algorithm for large graphs. But we see a decline in this runtime improvement as we scale up to 64-128 processor cores. This is due to the similar reason explained above.

Our hybrid algorithms main goal is to achieve high scalability along with maintaining accuracy of the communities detected. In Figure 4.5, we observe the percentage error using the difference in the modularity between sequential run and

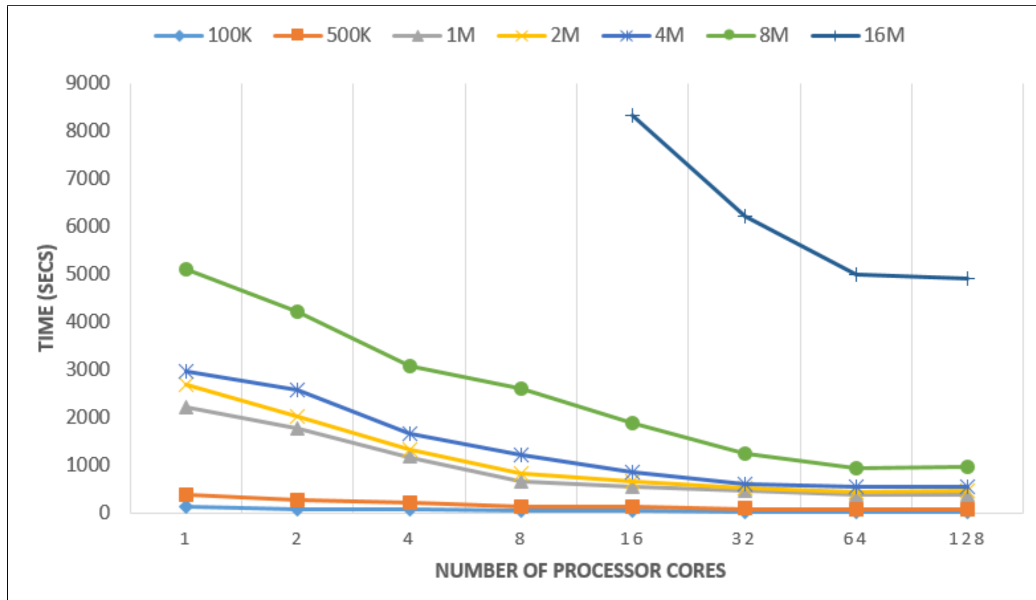

Figure 4.2: Run-time while scaling up the number of processor cores over varying graph sizes

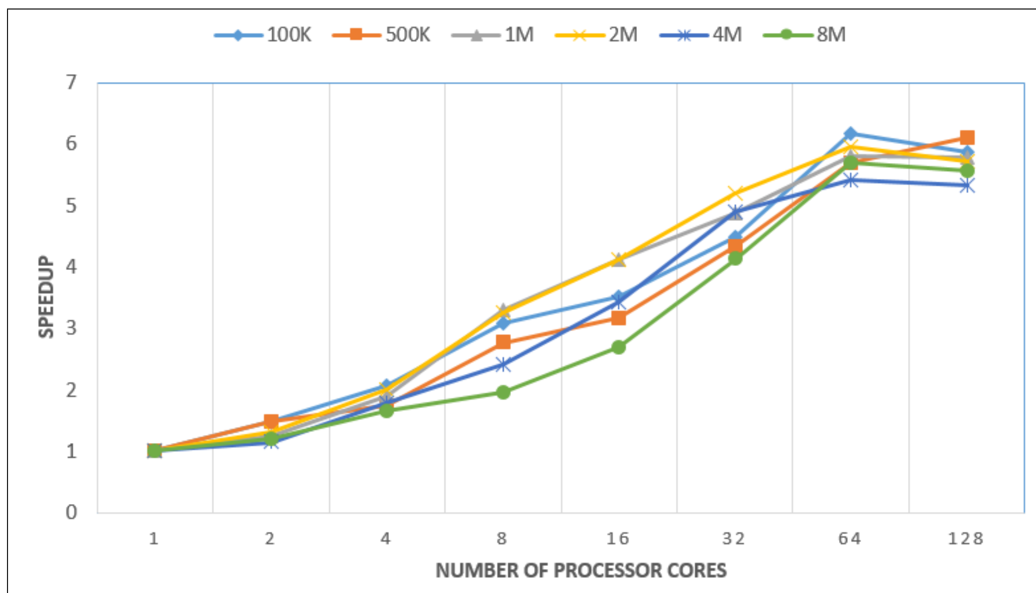

Figure 4.3: Speedups compared to sequential hybrid algorithm while scaling up the number of processor cores over varying graph sizes

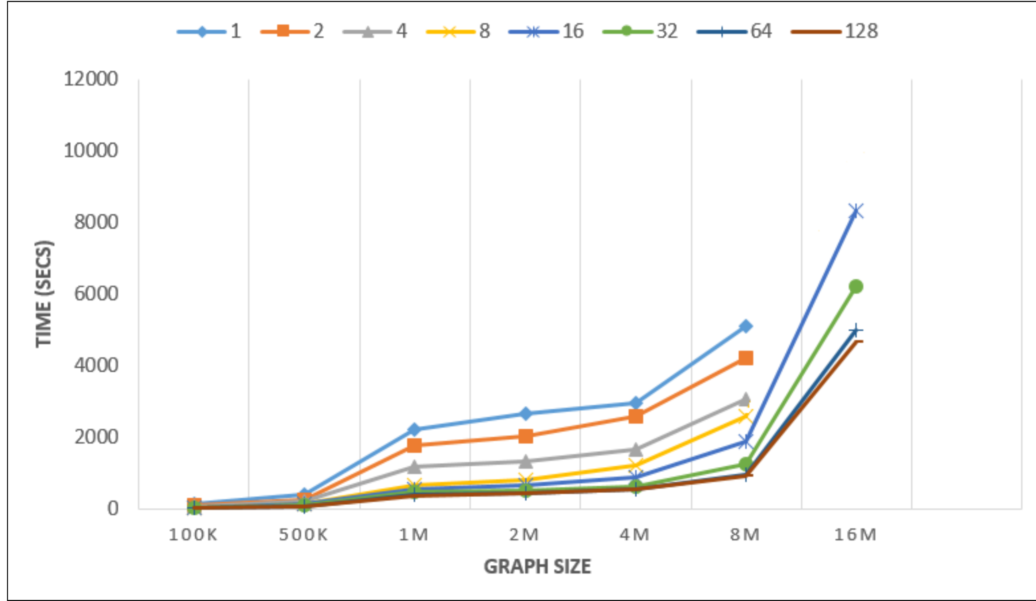

Figure 4.4: Run-time while scaling up the graph sizes over varying processor cores

parallel run of our hybrid algorithm. This error is calculated using the formula:

$$\%Error = abs\left(\frac{mod_{par} - mod_{seq}}{mod_{seq}}\right) \times 100 \tag{4.1}$$

Where mod_{seq} and mod_{par} represents the final modularity obtained by sequential run and parallel run of our hybrid algorithm respectively. We observe that the error percentage decreases as the size of graph increases.

This is an expected phenomenon, since PMETIS partitions the graph into multiple subgraphs by minimizing the number of cross partitioning edges between them i.e. minimum size edge cuts. For small graphs, PMETIS is constrained because of the number of partitions and hence will have to partition the graph with higher cross-partition edges. This leads to higher probability of partitioning the communities

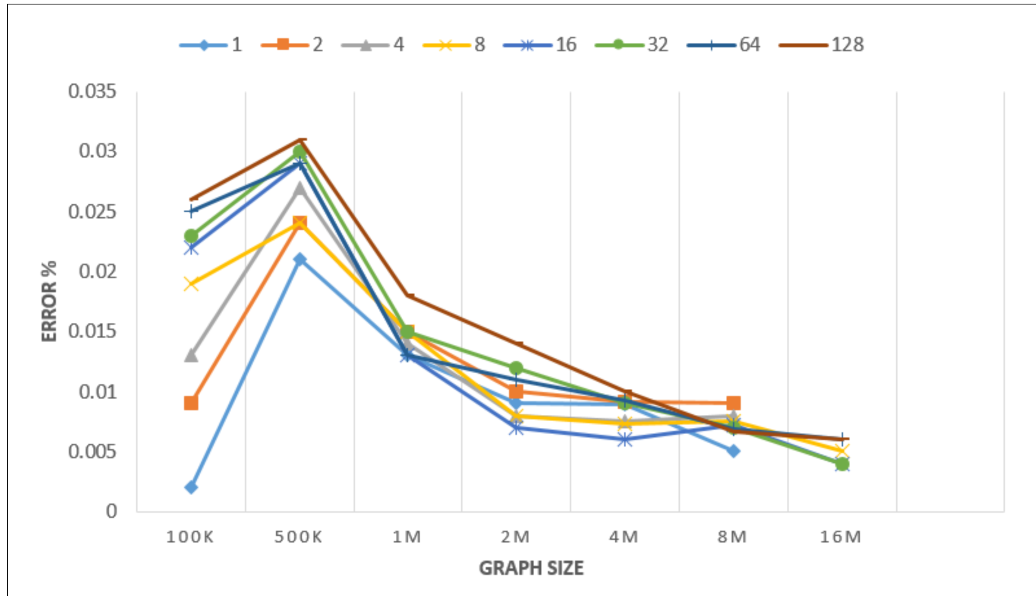

Figure 4.5: Change in error percentage of final modularity compared to that achieved by sequential execution of hybrid algorithm

across multiple subgraphs. Whereas, for larger graphs this probability decreases since PMETIS is not constrained as much by the number of partitions and can effectively reduce the number of cross-partition edges between subgraphs. This phenomenon is also observed in similar studies, like the one in ([93]). It is important to note that our method does not ignore the cross partition edges completely, since labels are allowed to transfer in the form of messages across cross-partition edges. But we do not collapse the nodes on the boundary i.e. associated with these cross-partitioning edges, which is done in step 2 of our hybrid algorithm.

We also test our hybrid algorithm on real world networks described in Subsection 4.5.2. In Figure 4.6, we see the variation in the total runtime while scaling

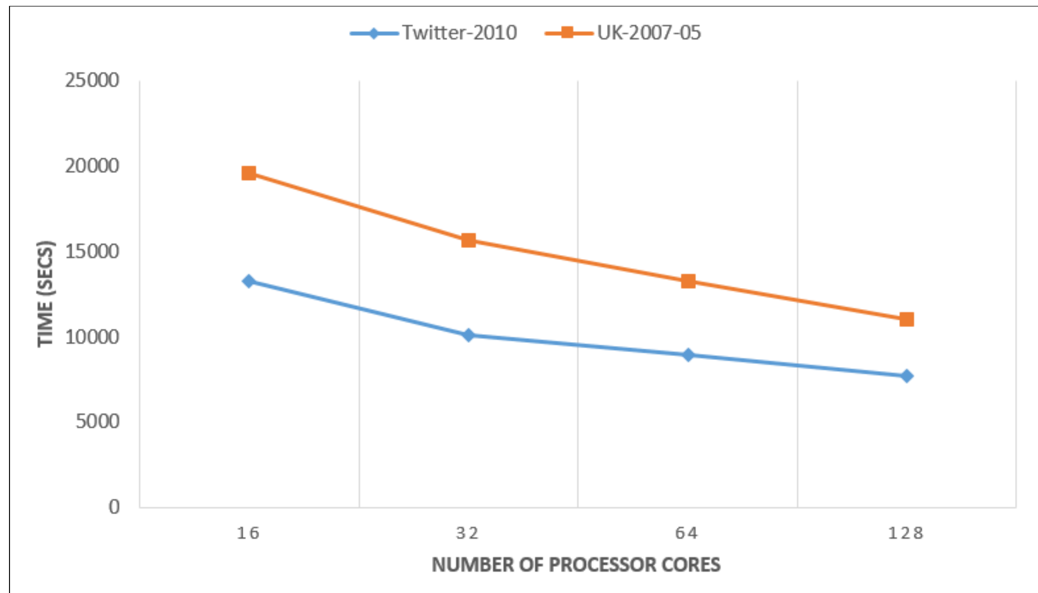

Figure 4.6: Run-time while scaling up the number of processor cores

up the number of processor cores for different graph sizes. 16 processor core run of our hybrid algorithm acts as the *"base run"* for our real world data sets due to memory constraints of a single machine. We can observe that our algorithm scales better than base run for both the datasets, as the number of processor core increases. In Figure 4.7, we see that our algorithm achieves $\approx 1.8X$ speedup for 128 processor cores, compared to the base run.

In Figure 4.8, we observe the percentage error using the difference in the modularity between base run and higher processor cores hybrid algorithm. This error is calculated using the formula mentioned in Equation 4.1. It is evident that we maintain good quality of the results along with achieving high scalability for large real world social networks, scaling up to hundreds of millions of vertices and billions of edges.

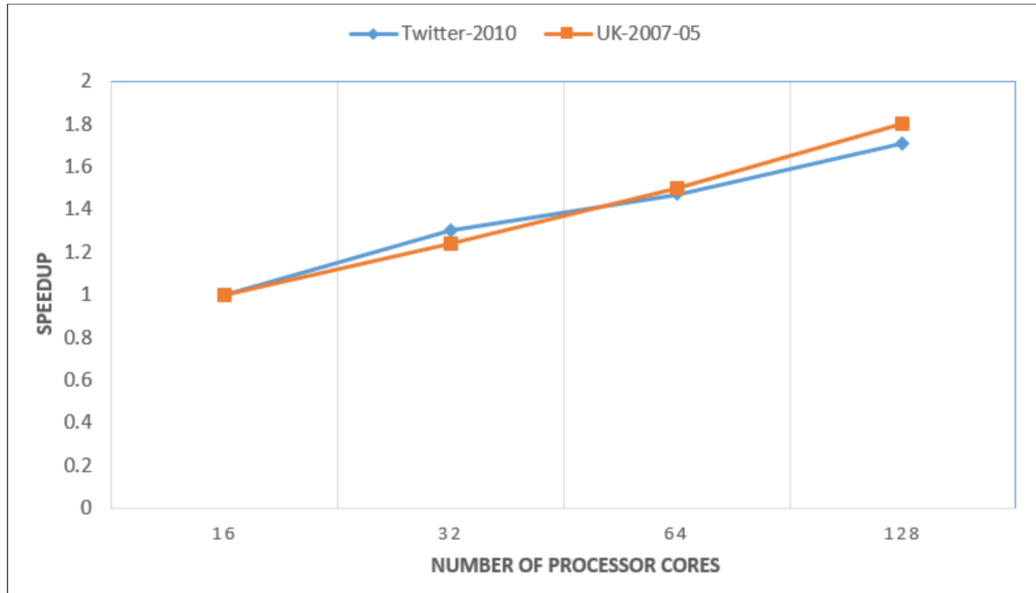

Figure 4.7: Speedups compared to base run of hybrid algorithm while scaling up the number of processor cores up to 128

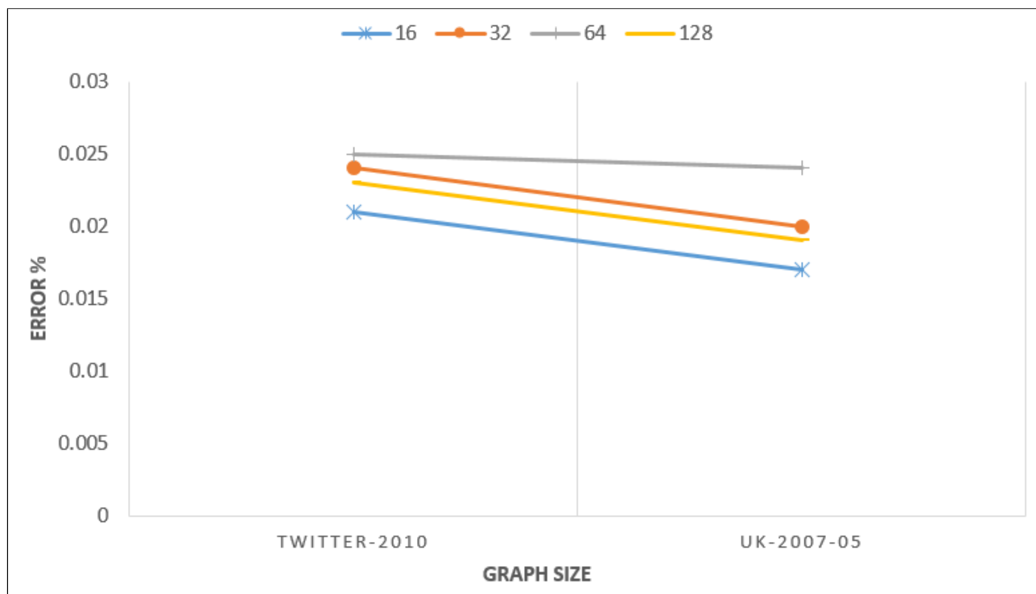

Figure 4.8: Change in error percentage of final modularity compared to that achieved by base run of hybrid algorithm while scaling up the number of processor cores up to 128

4.6 Discussion

Detecting communities in large networks, while achieving a good balance between scalability and quality of the results is an important open problem, especially due to the massive growth of social networks. This work combines our existing MCML algorithm [64] as a subroutine in our hybrid community detection algorithm presented in this paper. We also combine an existing graph partitioning technique (i.e. PMETIS) which minimizes cross-partition edges, as a preprocessing step to our algorithm. Our simulation results on a MPI setup with 8 compute nodes having 16 cores each shows that, upto $\approx 6X$ speedup is achieved for synthetic graphs upto $8M$ vertices in detecting communities without compromising the quality of results. Our hybrid algorithm can scale for large section of real world social networks like Twitter-2010 and UK-2007-05 having $\approx 41M$ and $\approx 105M$ vertices respectively and maintain good quality of the results when compared to other existing similar works. Our research described in this chapter will appear in [85].

In all the previous chapters including this chapter, we designed algorithms to process and analyze small to massive data which can be represented as networks i.e. graph structured data. Now we want to move to other types of massive datasets which cannot be represented in the form of a network and we would focus on LiDAR data which lies in the spatial domain.

CHAPTER 5
SPATIAL DOMAIN : DATA REDUCTION

5.1 Introduction

Airborne Light Detection and Ranging (LiDAR) is the most effective means for high density and high accuracy terrain data acquisition. Three dimensional spatial imaging with LiDAR technology is a powerful remote sensing methodology that can be used to produce detailed maps of objects, surfaces, and terrains across widely varying scales ([27]). Improved scanning technologies have made it easier to generate massive high density LiDAR point clouds and therefore more accurate, compact terrain models and other three dimensional representations ([87]). LiDAR topographic data provide highly accurate digital terrain information, which is used in applications like updating and creating flood insurance rate maps, forest and tree studies, coastal change mapping, soil and landscape classification, 3D urban modeling, river bank management, agricultural crop studies, etc. However, the generation of such improved models from high density, and enormous volume of data imposes great challenges with respect to data storage, processing, and manipulation.

Over the last 15 years, LiDAR data for generating reliable and accurate Digital Elevation Models (DEMs) is widely used in geospatial science communities ([31]). The accuracy of the generated DEM is directly proportional to the density of the sample terrain LiDAR data used. Hence, strategies to process large volumes of dense LiDAR data without compromising the accuracy, are essential. In this chapter, we

describe a novel algorithmic technique to reduce LiDAR data to achieve an optimal equivalence between the density and volume of data, which facilitates accurate and efficient generation of DEMs. Our data reduction technique reduces LiDAR points based on the topography, mainly slope-map of the terrain under consideration. To the best of our knowledge, this is the first ever landscape driven data reduction technique. We also use parallel programming to exploit multi-core architecture of CPUs, thus making our algorithm highly scalable and time-efficient.

The remainder of this chapter is organized as follows: In Section 5.2, we describe the related work. We state our contributions in Section 5.3. In Section 5.4, we describe our study area and LiDAR data reduction algorithm along with its parallel implementation. In Section 5.5, we discuss and present our experimental results, followed by some discussion in Section 5.6.

5.2 Related Work

The enhancement in data collection technologies has enabled generation of massive amounts of data, which poses computing issues when disseminating, processing, and storing data. Data is valuable only if it can convey valuable information. All the points in the entire LiDAR point cloud do not provide equally valuable information about the terrain under consideration ([16]). In order for a DEM to be useful, it should be of a desired size, so that it can be manipulated whenever required within the technology used to render it. This is one of the main challenges in massive geo-spatial data processing - reduce dataset to attain an optimal balance between

size of the dataset required, and desired resolution. Anderson et al.([4]) described the effects of LiDAR data density on generated DEMs for a range of resolutions. They further showed that LiDAR data can be reduced substantially yet still be able to generate accurate DEMs for elevation predictions. Liu et al. ([49]) explored effects of LiDAR data density on accuracy of generated DEMs, and studied the extent to which LiDAR data can be reduced and still achieve DEMs with required accuracy. Oryspayev et al.([66]) introduced a method of vertex decimation i.e. selective removal of points from the LiDAR point cloud that does not convey enough information. Hegeman et al.([29]) proposed a method in which each point was considered for deletion based on z-variance of the point cloud in the small local region. A variance threshold was initially set up as an input parameter; local regions having z-variance less than threshold, undergo removal of most of their central points. They concluded that, for certain regimes this point decimation technique performs significantly better than random decimation.

5.3 Contribution

We develop a novel landscape driven LiDAR data reduction algorithm, which preserves optimal balance between the density and volume of data, to generate accurate DEMs efficiently. We use parallel programming to exploit multi-core architecture of CPU's, thus achieving high scalability and efficiency. We detail various experiments we conducted, to determine the quality and performance of our technique.

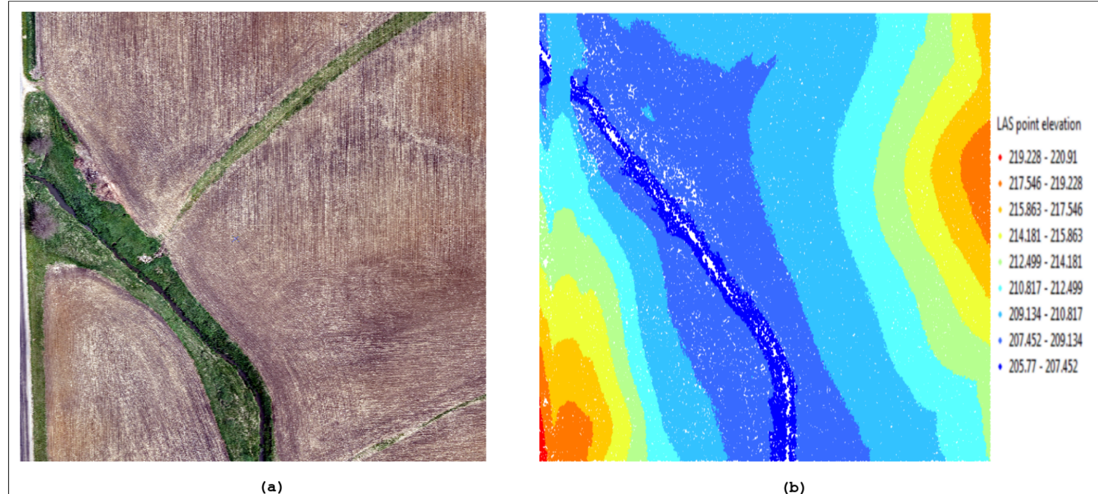

Figure 5.1: (a) Imagery of the study area (b) 2-D LiDAR point cloud of the study area which is colored based on variations in elevation

5.4 Data Reduction Algorithm

5.4.1 Preprocessing

The test LiDAR dataset we use for our experiments represents a 300×380 sq. m. tile, of a terrain in Iowa. The imagery of the study area is shown in Figure 5.1(a). This LiDAR dataset is interesting because, it has a good mix of flat land, land with moderate inclination and steep slopes. Thus it resembles a real-life terrain, rather than computationally ideal terrains consisting of just flat land. The slope angle for the entire terrain varies between 0.03° to 63.18°. This LiDAR dataset has been collected using Airborne LiDAR by Aerial Services, Inc. (ASI). The dataset is comprised of 6.94 million points, with a point density of approximately 120 points per sq.m. In the experiments, the input files were formatted in ASPRS LAS File Format, version 1.0. The size of this dataset was 1.5 GB. Also, for visualization purposes, we converted the

input data with extension .LAS to .xyz using *LASTools*. The LiDAR point cloud for the above data is generated using *ArcGIS*, ArcMap v10.3 as shown in Figure 5.1(b). The spatial resolution of the LiDAR data was estimated to be 1 inch vertically and 1 inch horizontally.

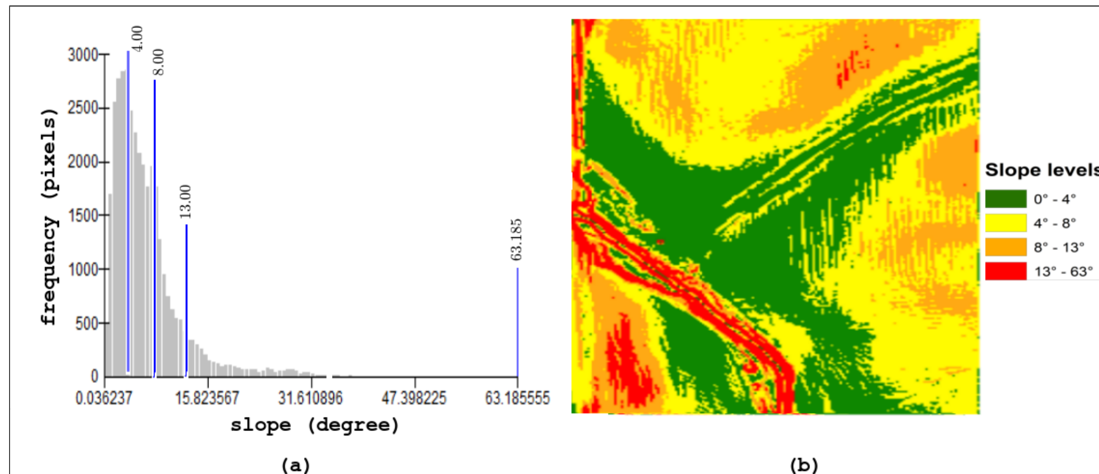

Figure 5.2: (a) Statistical analysis of elevation data for the terrain (b) Slope-map consisting of four slope ranges

Initially, we make use the elevation data available for the study area to generate a slope-map. For this work, we generate this elevation data using LiDAR data we collected for the entire region. Elevation data is also available freely in *National Elevation Dataset (NED)* `http://ned.usgs.gov/`, which is the primary elevation data product of the United States Geological Survey (USGS), and serves as the elevation layer of “The National Map”. The NED provides elevation information for earth science studies and mapping applications in the United States. We conducted

a statistical analysis of the elevation data for our dataset, using the *Natural Breaks (Jenks)* method in ArcGIS. This analysis showed that the slope angle for our terrain ranges from 0.03° to 63.18°, with the mean value of 6.52°, and standard deviation value of 5.73°. Based on this statistical analysis showed in Figure 5.2(a), the slope-map having four regions, with different slope ranges were generated as shown in Figure 5.2(b). The ranges were $Slope_{green} = [0.03°, 4°]$, $Slope_{yellow} = [4.00°, 8.00°]$, $Slope_{orange} = [8.00°, 13.00°]$ and $Slope_{red} = [13.00°, 63.18°]$. $Slope_{green}$ represent the flattest regions in the terrain, whereas $Slope_{red}$ represent the the most uneven and rough regions.

After the creation of the slope-map layer, we overlay the LiDAR points we collected for the same terrain on it, while preserving spatial geo-referencing. Based on which slope range the LiDAR point lies in, we append a new parameter i.e., *Slope* for each LiDAR point. After processing, our LiDAR dataset contains x, y, z, and *Slope* values, which we use as an input to our data reduction algorithm. This entire process is done using *ArcGIS* and *LASTools*.

5.4.2 Algorithm

In our LiDAR data reduction algorithm, initially we overlay a grid of 1 sq.m. cells on the LiDAR data, by preserving the spatial geo-referencing. Select the first cell in the grid, and choose a LiDAR point randomly from all the LiDAR points that lie in that grid cell. We keep on selecting different random LiDAR points from the selected cell, until we get a LiDAR point that lies in $Slope_{green}$ region, or until all the

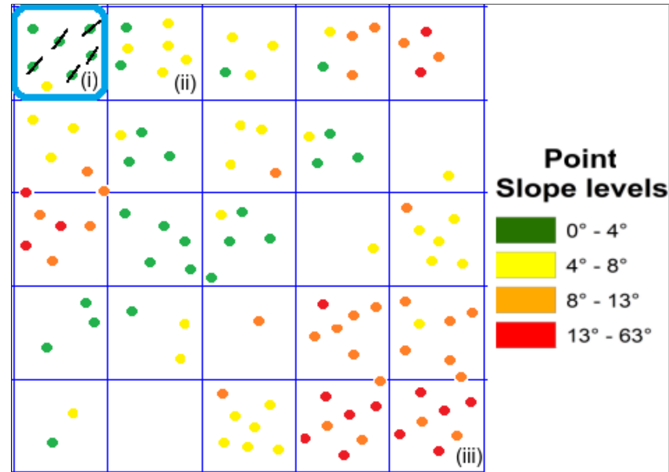

Figure 5.3: Data reduction algorithm : Grid overlayed on processed LiDAR data

LiDAR points are checked. If we find a LiDAR point lying in $Slope_{green}$ region, we check whether β% (user input) of the LiDAR points in that cell belongs to $Slope_{green}$ region or not. If it does, then we remove all the points in that cell which belong to $Slope_{green}$ region, except the chosen point (ex. cell (i) in Figure 5.3, with $\beta = 90\%$). If we do not find LiDAR point lying in $Slope_{green}$ region, we check for LiDAR point lying on $Slope_{yellow}$ region (ex. cell (ii) in Figure 5.3), and do the same as above. If we do not find any LiDAR point lying in regions $Slope_{green}$ or $Slope_{yellow}$, or there is no removal of points in the cell; (ex. there won't be any data removal in cell (iii)), we simply proceed to the next cell in the grid. We repeat the above steps until we have processed all the cells in the grid, starting left-right, top-bottom fashion. The motivation behind this LiDAR data reduction algorithm is the fact that, we do not need too many LiDAR points to represent a flat terrain as compared to an irregular

terrain.

Algorithm 5.1 : LiDAR data reduction algorithm

```
1: Input Grid of 1 sq.m. on geo-referenced LiDAR data
2: Return Reduced LiDAR data
3: Initialize: i = 0, cell[k], p = 0, β ;
4: while (cell[i]) do
5:    Select point p ∈ cell[i] randomly
6:    while (All points p ∈ cell[i] is checked) do
7:       if (p ∈ Slope_green and β% of total points in cell[i] ∈ Slope_green) then
8:          Remove all points lying in Slope_green except p;
9:          break;
10:      else if (p ∈ Slope_yellow and β% of points in cell[i] ∈ Slope_yellow) then
11:         Remove all points lying in Slope_yellow except p;
12:         break;
13:      end if
14:      Select different p ∈ cell[i] randomly;
15:   end while
16:   i + +;
17: end while
```

5.4.3 Parallel Implementation

Algorithm 5.1, when implemented sequentially, is extremely inefficient in terms of running time. It takes approximately 6 hours to process a LiDAR point cloud of

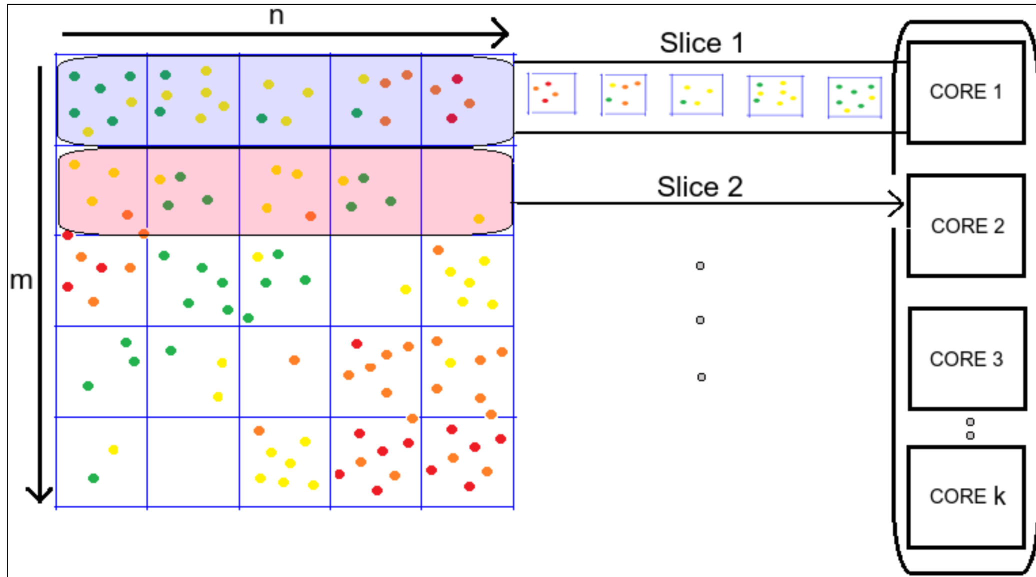

Figure 5.4: Parallel implementation of data reduction algorithm

6.94 million points sequentially. In this section, we present a parallel implementation of the LiDAR data reduction algorithm, which is much faster than the sequential version and also highly scalable. We distribute the LiDAR data over multiple cores of the CPUs, and process them in parallel using both, block and cyclic assignments.

Consider m rows $\times$ n columns grid ($m, n \in \mathbb{N}$), which is overlay on the LiDAR data, shown in Figure 5.4 . For a k core processor ($m \geq k$), we dedicate a master core which slices the first k of m rows, and assign each one of them to each processor individually (block assignment of slices). Each of these k slice has n cells, which are processed one at a time by each core to which the corresponding slice is assigned (cyclic assignment of cells). Once every cell in all the k slices are processed, the next k of m rows are sliced (cyclic assignment of slices) and distributed to the CPU cores, similarly as above. The results of the scalability tests using $1, 2, 4, 8$ and 16 CPU

cores and speed-ups are shown in Figure 5.6.

5.5 Computational Results

With the dense LiDAR data we have, consisting of 6.94 million points, a high-accuracy (1 meter vertical and horizontal accuracy) and high-resolution DEM which covers area 300 × 380 sq. m. of an irregular terrain in Iowa, was generated using IDW interpolation method (shown in Figure 5.5). We test our algorithm for different input values of $\beta\%$ in order to find the optimal balance between accuracy and density of LiDAR data.

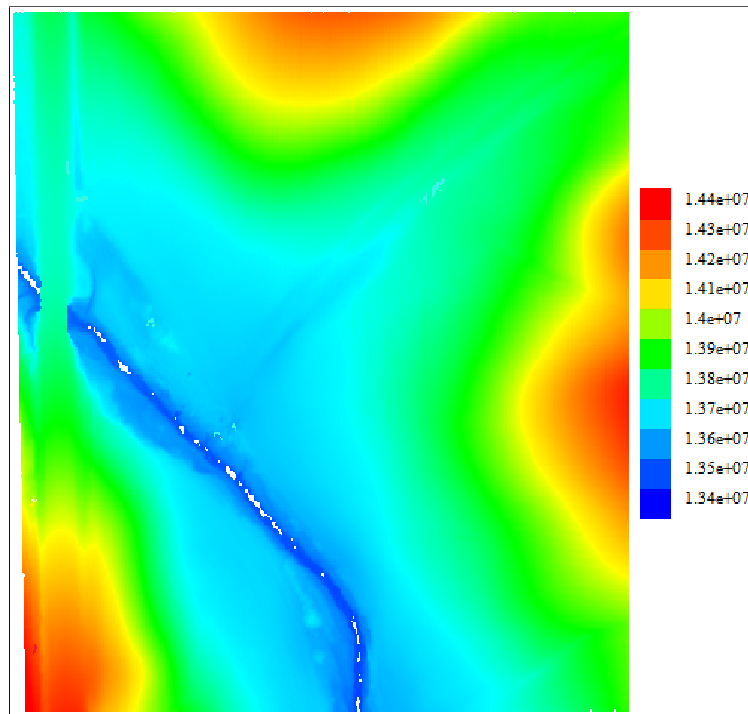

Figure 5.5: DEM generated for the original dataset having 6.94 million LiDAR points

In Figure 5.6(a), we have shown the speed-ups and scalability test of our LiDAR data reduction algorithm for $1, 2, 4, 8$ and 16 CPU cores. Using a 16 core Xeon Phi processor we speed-up the running time of algorithm by 15.81 times i.e. from $\approx$ 6 hours to $\approx$ 20 minutes, for the dataset under consideration. The scalability we achieve is closer to a k fold improvement for a k core processor ($k \geq 1$).

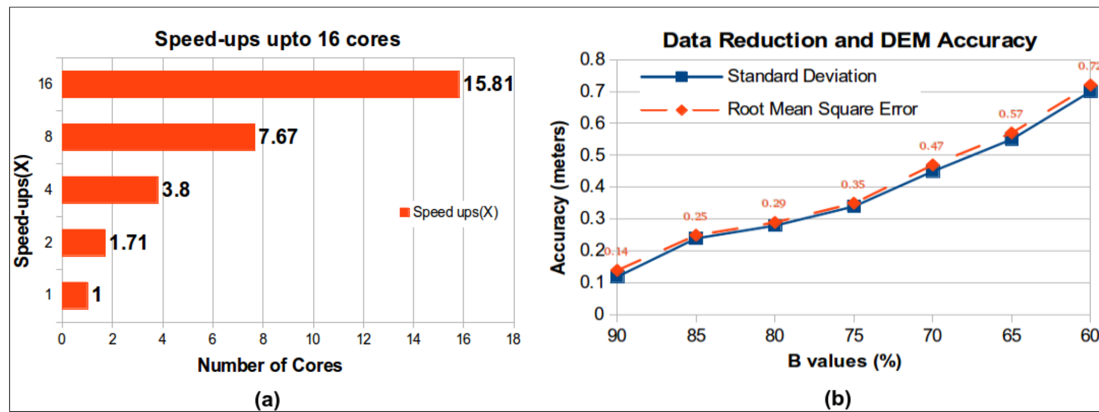

Figure 5.6: (a) Parallel speed-ups for LiDAR data reduction algorithm (b) Data reduction and DEM accuracy

We observed that, compared to the DEM generated from the original complete LiDAR dataset, there is no significant decrease in accuracy for the DEM generated from the 52% reduced dataset obtained by applying our algorithm for $\beta = 90\%$ to the original LiDAR dataset as input. The root mean square error (RMSE) and standard deviation supporting the same is shown in Figure 5.6(b). In comparison to the original DEM, the error introduced in the generated DEM, when $\beta = 90\%$, is

only 0.14 meters. In Figure 5.7(a), we have shown the 2-D and 3-D DEM generated from 66% reduced dataset obtained by applying our algorithm for $\beta = 80\%$, and in Figure 5.7(b), DEM generated from 52% reduced dataset obtained for $\beta = 90\%$. From Figure 5.7(a) and Figure 5.6(b), we can see that, there is significant loss of accuracy in the DEM generated from reduced dataset obtained for $\beta = 80\%$ and $\beta \leq 85\%$ respectively. For $\beta = 80\%$ the data density is reduced by 66%, but there is an error of 0.29 meters which reduces the accuracy of the DEM considerably. Whereas for values of $\beta > 90\%$ the percentage of data reduction is not significant.

The processing time for the DEM generation is directly proportional to the size of the LiDAR data used for its generation ([50]). It takes half the time to generate DEM from 52% reduced dataset for $\beta = 90\%$ compared to the original LiDAR dataset. The smaller the value of β, the lower the density of the reduced LiDAR data, and lesser is the accuracy of DEMs generated. We need to decide the value of β based on the type of terrain for example, based on our study, $\beta = 90\%$ is optimal for terrains having a good mix of flat land, land with moderate inclination, and steep slopes. It reduces the data density to half the size of the original dataset, and also preserves high accuracy of the DEMs generated. For terrains dominated by flat lands, higher β values may lead to optimal reduction of LiDAR data, whereas for terrains dominated by moderate, steep slopes, and rough regions, lower values of β may be the correct choice. We thus demonstrate that our LiDAR data reduction algorithm can significantly improve the processing time, and the file size of DEM generations.

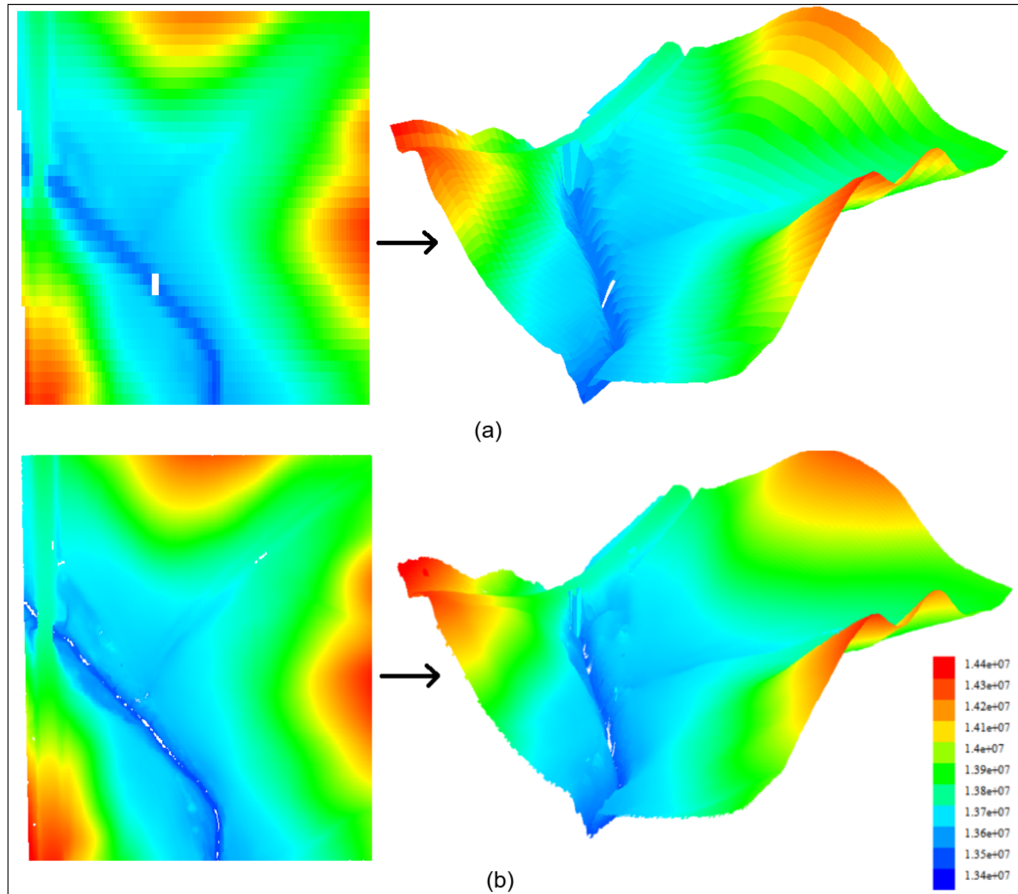

Figure 5.7: (a) $\beta = 80\%$, reduced dataset comprising of 2.2 million points (b) $\beta = 90\%$, reduced dataset containing 3.1 million points

5.6 Discussion

It is observed that not all LiDAR data contribute effectively to the accurate generation of DEMs. It is important to identify points representing the specific features of the terrain which contain more significant information, compared to other points ([49, 16]). While designing our LiDAR data reduction algorithm, we take features like slope of the terrain into consideration, to remove less important points and keep critical points. Of all the features of a terrain, we choose slope to be the most

important feature. Terrain slopes highlight changes in the terrain surfaces which provides elevation information of a point, and they also showcase information about their surroundings. Significant changes in slopes of the terrain indicates points with more critical information, compared to other points ([47]). Thus using our LiDAR data reduction algorithm, which includes slope of the terrain as the main data removal factor, reduces the number of data points required for DEM generation, while maintaining high accuracy. Results show that our parallel implementation of this algorithm is highly scalable and efficient in terms of processing times. Our research described in this chapter also appears in [86].

Many algorithms for DEM generation have been described in several studies, but taking into account the specific characteristics of LiDAR data and the application, it is important to select appropriate interpolation algorithms, modeling techniques, and resolution of the DEM. One of our applications involve real-time mapping of terrain on which the vehicle equipped with LiDAR sensor moves. Using traditional sequential spatial interpolation algorithms along with our data reduction algorithm is not sufficient to achieve the goals of real-time terrain mapping. So it makes it essential to design a multi-core parallel spatial interpolation algorithm which can be combined with our data reduction algorithm, to address the problem of real-time terrain mapping. In the next chapter, we describe our multi-core spatial interpolation algorithm to address this issue.

CHAPTER 6
SPATIAL DOMAIN : INTERPOLATION

6.1 Introduction

Spatial data interpolation is a crucial technique in Geographical Information System (GIS), which computes unknown terrain height values of points, based on the known elevation values of points in the neighborhood ([47]). A natural terrain surface is a continuous surface comprising of infinite points ([22]). We use point sampling techniques to approximate the accuracy of the generated DEMs to the required resolutions. The most commonly used DEMs are the grid DEM, the contour line DEM, and triangular irregular network (TIN) DEM. A grid DEM can be represented as a matrix, having related data points which capture information of the terrain's topography. Every grid cell has a value which denotes the elevation for the entire cell ([80]). Each of the grid cells get this elevation value by interpolating (approximation procedure) adjacent sampling points. Burrough et al. ([14]) defined interpolation as, a process of interpreting values at points found in unsampled regions, on the basis of values at points within the confined area of study. Interpolation techniques in grid DEMs is used to determine the terrain height value of a point based on the known elevation values of points in the neighborhood ([47]). Spatial interpolation methods are defined on the basis of geometric and geo-statistical properties. Spatial interpolation can be classified in various classes like local, global, deterministic, probabilistic, exact and approximate. Local interpolation techniques just process a part

of the dataset as opposed to the global techniques, which processes the entire dataset. Exact interpolation techniques like Kriging, (Inverse Distance Weighted) IDW and some spline methods, generates DEMs which takes into consideration all the points in the dataset. Probabilistic interpolation methods use geo-statistics to generate DEMs. These methods include Kriging, and Fourier analysis. The quality of the generated DEMs is evaluated based on the difference between the "true" and the interpolated value at points in entire or selected locations ([12]).

Practically applying spatial interpolation is a computationally expensive task and it requires powerful computing resources. Spatial interpolation is applied more to massive data analysis, which requires more processing time. We develop parallel shared memory spatial interpolation technique, which exploits multiple cores of CPUs. We conduct comparative studies of the DEM generated by our algorithm, to the ones generated by traditional sequential approaches, using validation technique, and also evaluate the comparison using statistical approaches like Root Mean Square Error (RMSE). We also conduct comparisons of our spatial interpolation with various interpolation algorithms, and DEM resolutions, to check where our algorithm lies in terms of performance and quality, in the comprehensive guidelines of this area.

The remainder of this chapter is organized as follows: In Section 6.2, we describe the related work. We state our contributions in Section 6.3. In Section 6.4, we describe our LiDAR data interpolation algorithm along with its parallel implementation. In Section 6.5, we discuss and present our experimental results, followed by some discussion in Section 6.6.

6.2 Related Work

There are many spatial interpolation techniques to generate DEMs like, grid DEM generation technique called Inverse Distance Weighted (IDW), Triangular Irregular Networks (TIN), geo-statistical methods like Kriging, local polynomial, etc. ([22]) showed that generating DEMs using grid DEM techniques has more efficient storage and manipulation scope. DEMs generated using grids introduces errors, since the terrain is represented in a discrete fashion. The size of the grid used for generating DEM is directly proportional to the approximation ratio of the terrain surface representation. Since LiDAR data is dense, such limitations of grid DEM method can be eliminated. Kraus et al.([41]) studied complex models to generate DEMs resulting from hybrid techniques. But in practice, all the DEMs generated from LiDAR are done using grids techniques ([52]). Due to the availability of large variety of interpolation techniques, questions on which is the most appropriate technique for different terrains needs to be answered. The authors of ([22, 99, 51]), conducted empirical studies to answer these questions, and evaluate the affects of various interpolation techniques on DEM quality. There does not exist any one interpolation technique which is optimal for all terrain surface data ([23]). IDW interpolation technique is proved to exhibit better performance when the sampled data has high density. Since LiDAR data has high density IDW is a preferred choice to generate DEMs ([68, 1]). In this chapter, we describe our parallel algorithm for spatial interpolation, which aims towards generating high quality DEMs in less computational time, compared to traditional approaches.

6.3 Contribution

We develop a modified IDW spatial interpolation technique, which generates better or similar quality DEMs in less computational time, compared to traditional IDW. We design a novel parallel implementation which exploits multi-core architecture of the CPUs to achieve high scalability and efficiency.

6.4 Spatial Interpolation

Spatial data interpolation is a crucial technique in Geographical Information System (GIS), which computes unknown terrain height values of points, based on the known elevation values of points in the neighborhood ([47]). Processing massive spatial data is a computationally expensive and complex process, and traditional sequential algorithms cannot meet the demand for faster processing speeds along with maintaining accuracy. In this section, we describe a parallel spatial interpolation algorithm which is a modification to *QuickGrid* ([18]) and *traditional IDW*. The modification takes place in the algorithm, as well as its implementation, where it exploits the multi-cores of the CPU to increase the computational speed.

6.4.1 Algorithm

We initialize the algorithm by overlaying a grid of k sq.m. cells ($k > 0$) on the LiDAR data, while preserving spatial geo-referencing. We used $k = 0.0254$ sq. m., for our simulations. Each LiDAR point has x, y and z coordinate, and we want interpolate for z-values to be assigned to each grid cell. Steps for our spatial interpolation algorithm are as follows:

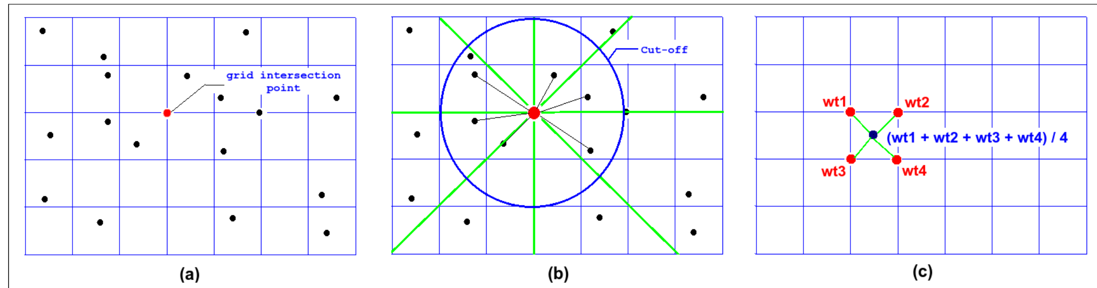

Figure 6.1: (a) Grid of k sq.m. ($k > 0$) overlaid on the LiDAR data (b) Selecting a cut-off radii and assigning weights to grid intersections (c) Assigning weights to each grid cell

1. Parse through the intersection points of each cell in the grid (shown in Figure 6.1(a)) in left-right, top-bottom fashion.

2. For each intersection point, we compute and assign a weight as follows:

   - Initially select a cut-off radius for the circle whose center is the intersection point. (It is recommended to choose small cutoff radii for very dense datasets, compared to less dense datasets.)

   - Then divide the circle into eight equal sectors, and choose the closest LiDAR points in each sector, if there exist any (shown in Figure 6.1(b)). Set the grid intersection point to the average of these chosen LiDAR points weighted by 1/(distance from grid intersection)^2.

3. After assigning weights to all the grid intersection points, we assign each grid cell the average weight of its four surrounding grid intersection points (shown in Figure 6.1(c)).

6.4.2 Parallel Implementation

The sequential implementation of the algorithm described in Subsection 6.4.1 is highly inefficient in terms of processing time. Parsing over every grid cell and grid intersection points individually, multiple times, can be computationally expensive and time consuming task. In this subsection, we design a parallel implementation for the above algorithm consisting of two phases: the *split phase*, where we distribute the data over multiple cores of a CPU to process it simultaneously, and the *merge phase*, where we merge the processed data back together.

Initially, in the *split phase*, we dedicate a master core which divides the grid (along with the LiDAR data) into k equal parts, where k is the total number of CPU cores available for processing. It distributes and assigns each of the k parts of the grid to each core individually (block assignment), shown in Figure 6.2. Each core then simultaneously execute the algorithm mentioned in Section 4, for the part of the grid data that is assigned to it.

Once all the cores have finished their computations, the master core initializes the *merge phase*, where the common grid intersection points between two grid parts are averaged and merged (shown in Figure 6.2), so as to get the original grid with all grid intersection points computed. Then the master core divides the grid into k equal parts and assigns each of the k parts individually to each core, which computes and assigns each grid cell the average weight of its surrounding four grid intersection points. Scalability tests using 1,2,4,8 and 16 CPU cores and speed-ups are shown in Section 4.3.

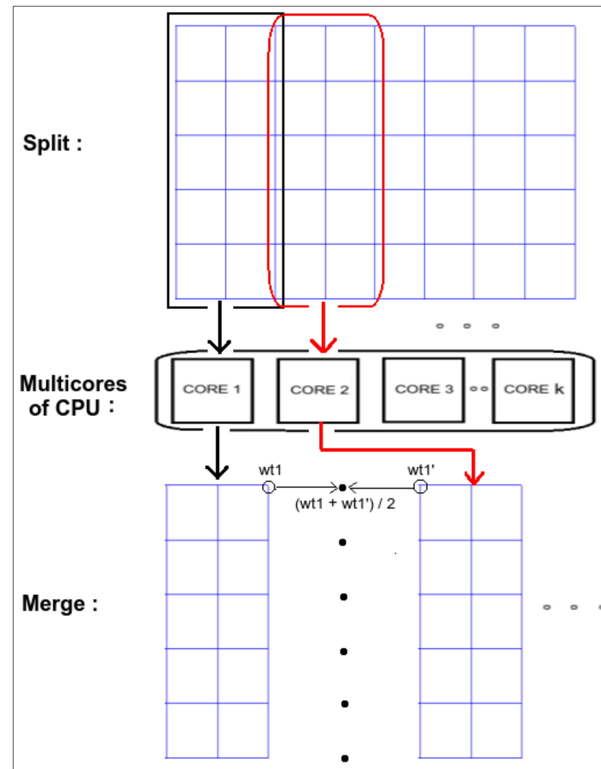

Figure 6.2: Parallel split and merge phases of our spatial interpolation algorithm

6.5 Computational Results

In this section, we test our spatial interpolation algorithm with fixed cut-off 1.5 meters, on two different test terrains shown in Figure 5.5, which is our 'dataset 1' and Figure 6.3, which is our 'dataset 2'. Dataset 2 is 200 × 500 meters, relatively flatter with shallow valleys, and less rough with slope angle ranging from 0.07° to 43.8° when compared to dataset 1, which is a mix of deeper valleys, steep slopes and few flat lands. dataset 2 contains 2.1 million LiDAR points. All the LiDAR data density reduction is done using the algorithm described in Chapter 5, with $\beta = 90\%$.

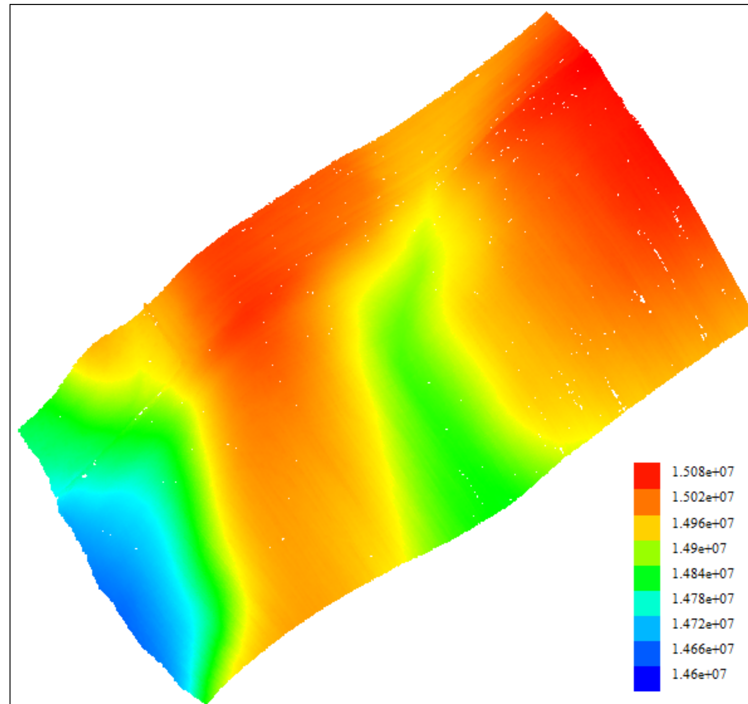

Figure 6.3: (a) Imagery of study area for dataset 2 (b) Elevation map of dataset 2 showing terrain with less roughness, shallow valleys and flat regions

By applying our LiDAR data reduction algorithm to dataset 1, we reduce the data density to 52%, and when applied to dataset 2 for $\beta = 80\%$ we reduce the data density to 71%. We then generate DEMs with this reduced LiDAR data, as well as complete LiDAR dataset using traditional IDW and modified IDW.

To obtain high accuracy in our statistical analysis, we compare the elevation values of each of the LiDAR points to the corresponding elevation value of the DEMs generated, rather than checking for few control points. This method is known as *validation*. RMSEs were calculated (shown in Figure 6.4(b)), to study the performance

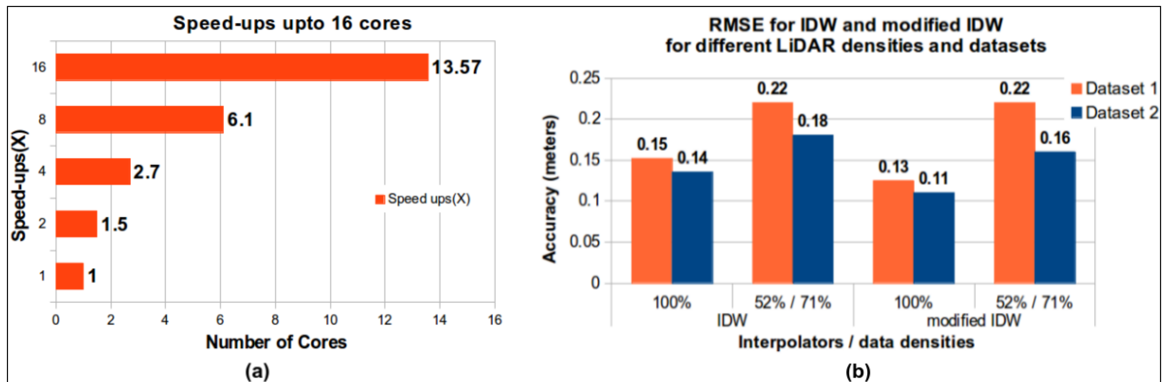

Figure 6.4: (a) Parallel speed-ups for our spatial algorithm (b) RMSE for IDW and modified IDW at different LiDAR density levels for two datasets using validation method

of our spatial interpolation algorithm compared to traditional IDW algorithm, for different LiDAR data density. From our empirical study shown in Figure 6.4(b), we can conclude the following :

- RMSEs for both the interpolation algorithms increase with decrease in LiDAR data density for both the datasets.
- RMSEs for the more complex terrain, i.e. dataset 1, is higher than dataset 2, which is relatively flat and has shallow valleys and less roughness.
- Quality of results obtained by modified IDW is at least as good as traditional IDW for reduced density, complex terrain (Ex. 52% reduced LiDAR dataset 1, gives an RMSE of 0.22 meters for both the algorithms).
- Quality of results obtained by modified IDW is better than traditional IDW for

dense, complex and relatively flatter terrains.

- Time to generate DEMs using modified IDW, is much less than that used by sequential traditional IDW.

As shown in Figure 6.4(a), using a 16 core Xeon Phi processor we speed-up the running time of algorithm by 13.5X i.e. 190 seconds using single core versus 14 seconds.

6.6 Discussion

In this chapter we present our modified IDW spatial interpolation algorithm which achieves results which are at least as good as traditional IDW for reduced density, complex terrain, and better than traditional IDW for dense, complex and relatively flatter terrains. It also achieves good scalability, and takes much less time to generate DEMs compared to traditional IDW. Our parallel IDW algorithm is integrated with our data reduction algorithm and used as a core subroutine in streaming based applications.

CHAPTER 7
EXPERIMENTAL ENVIRONMENT

7.1 Chapter 1 : Biological Domain

We implemented the BLLP algorithm using C++ and graph libraries. All the simulations are done on Processor-Intel Core i7 3770 3.4GHz and Turbo Boost enabled Memory-16GB DDR3-1600 RAM 500G 3GB/s 7200 RPM; Linux machines. All the plots are done using Gephi and Gnuplot.

7.2 Chapter 3 : Social Networks Domain - Shared Memory

We implemented the MCML algorithm using C++ and boost graph libraries. The simulations for the benchmark datasets and the Facebook forum dataset are done on Processor-Intel Core $i73770$, $3.4GHz$ and Turbo Boost enabled Memory-$16GB$ $DDR3-1600$ RAM; Linux machines. These machines have 4 cores with hyper-threading enabled. The simulations for the Amazon dataset is done on a system running on CentOS 6.3, a Linux operating system based on Red Hat Linux, with $512GB$ Nodes, 32 GB RAM, $2.9GHz$, and 16 Xeon Phi cores. All the results obtained are average of 5 runs. We use OpenMP directives for implementing parallel MCML algorithm. All the plots are done using Gephi and Gnuplot.

7.3 Chapter 4 : Social Networks Domain - Distributed Memory

The performance of our hybrid algorithm is evaluated by executing series of experiments on the High Performance Neon Cluster at University of Iowa. We im-

plemented the hybrid algorithm using C++ and boost graph libraries. We also use parallel implementation of PMETIS which was developed in GNU C++ MPI. We use 8 heterogeneous standard machines each having 64GB RAM, 16 Xeon Phi cores and 2.6 GHz processor. All the experiments were executed as a single batch command comprising of at most 8 compute machines having 16 cores each. Each experiment is executed 3 times and average of the results from these runs are reported to preserve accuracy and consistency.

7.4 Chapter 5 & 6 : Spatial Domain - Data Reduction & Interpolation

We implemented the LiDAR data reduction algorithm and modified IDW using C++. We use OpenMP/p-threads directives for implementing parallel versions of the above algorithms. The simulations for the LiDAR dataset is done on a system running on CentOS 6.3, a Linux operating system based on Red Hat Linux, with 512GB Nodes, 32 GB RAM, 2.9GHz, and 16 Xeon Phi cores. The LiDAR point cloud, slope-map and slope statistics are generated and visualized using *ArcGIS, ArcMap* v10.3. IDW algorithm is used to generate DEMs for reduced LiDAR data (not our new spatial interpolation algorithm). We use *QuickGrid* tool to visualize DEM's generated by our algorithms. All the plots are done using *Gnuplot* and *LibreDraw*. All the results obtained are average of five runs.

REFERENCES

[1] Tarig A Ali. On the selection of an interpolation method for creating a terrain model (tm) from lidar data. In *Proceedings of the American Congress on Surveying and Mapping (ACSM) Conference*. Citeseer, 2004.

[2] Md Altaf-Ul-Amin, Yoko Shinbo, Kenji Mihara, Ken Kurokawa, and Shigehiko Kanaya. Development and implementation of an algorithm for detection of protein complexes in large interaction networks. *BMC bioinformatics*, 7(1):207, 2006.

[3] Grama Ananth, Anshul Gupts, George Karypis, and Vipin Kumar. Introduction to parallel computing, 2003.

[4] ES Anderson, JA Thompson, and RE Austin. Lidar density and linear interpolator effects on elevation estimates. *International Journal of Remote Sensing*, 26(18):3889–3900, 2005.

[5] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[6] Gary D Bader and Christopher WV Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC bioinformatics*, 4(1):2, 2003.

[7] Seung-Hee Bae, Dan Halperin, Jevin West, Martin Rosvall, and Brandon Howe. Scalable flow-based community detection for large-scale network analysis. In *Data Mining Workshops (ICDMW), 2013 IEEE 13th International Conference on*, pages 303–310. IEEE, 2013.

[8] Albert-Laszlo Barabasi and Zoltan N Oltvai. Network biology: understanding the cell's functional organization. *Nature Reviews Genetics*, 5(2):101–113, 2004.

[9] Chapman Barbara, Jost Gabriele, and P Ruud. Using openmp: portable shared memory parallel programming, 2007.

[10] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[11] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[12] Graeme F Bonham-Carter. *Geographic information systems for geoscientists: modelling with GIS*, volume 13. Elsevier, 2014.

[13] Stefan Bornholdt and Heinz Georg Schuster. *Handbook of graphs and networks: from the genome to the internet*. John Wiley & Sons, 2006.

[14] Peter A Burrough and Rachael A McDonnell. *Principles of geographical information Systems*, volume 19988. Oxford University Press, 2011.

[15] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[16] Yue-Hong Chou, Pin-Shuo Liu, and Raymond J Dezzani. Terrain complexity and reduction of topographic data. *Journal of Geographical Systems*, 1(2):179–198, 1999.

[17] R Cohen, S Havlin, Daniel ben Avraham, and S Bornholdt. Handbook of graphs and networks. *Willey-VCH, New York*, 2002.

[18] J Coulthard. Quikgrid: 3-d rendering of a surface represented by scattered data points, 2007.

[19] Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti. Generalized louvain method for community detection in large networks. In *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on*, pages 88–93. IEEE, 2011.

[20] Inderjit Dhillon, Yuqiang Guan, and Brian Kulis. A fast kernel-based multilevel algorithm for graph clustering. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 629–634. ACM, 2005.

[21] Chris Ding, Xiaofeng He, Hui Xiong, and Hanchuan Peng. Transitive closure and metric inequality of weighted graphs: detecting protein interaction modules using cliques. *International journal of data mining and bioinformatics*, 1(2):162–177, 2006.

[22] Naser El-Sheimy, Caterina Valeo, and Ayman Habib. *Digital terrain modeling: acquisition, manipulation, and applications*. Artech House, 2005.

[23] Peter F Fisher and Nicholas J Tate. Causes and consequences of error in digital elevation models. *Progress in Physical Geography*, 30(4):467–489, 2006.

[24] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[25] Santo Fortunato, Vito Latora, and Massimo Marchiori. Method to find community structures based on information centrality. *Physical review E*, 70(5):056104, 2004.

[26] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[27] Ayman Habib, Mwafag Ghanma, Michel Morgan, and Rami Al-Ruzouq. Photogrammetric and lidar data registration using linear features. *Photogrammetric Engineering & Remote Sensing*, 71(6):699–707, 2005.

[28] Takako Hashimoto, Basabi Chakraborty, and Yukari Shirota. Social media analysis–determining the number of topic clusters from buzz marketing site. *International Journal of Computational Science and Engineering 6*, 7(1):65–72, 2012.

[29] James W Hegeman, Vivek B Sardeshmukh, Ramanathan Sugumaran, and Marc P Armstrong. Distributed lidar data processing in a high-memory cloud-computing environment. *Annals of GIS*, 20(4):255–264, 2014.

[30] Yuen Ho, Albrecht Gruhler, Adrian Heilbut, Gary D Bader, Lynda Moore, Sally-Lin Adams, Anna Millar, Paul Taylor, Keiryn Bennett, Kelly Boutilier, et al. Systematic identification of protein complexes in saccharomyces cerevisiae by mass spectrometry. *Nature*, 415(6868):180–183, 2002.

[31] Michael E Hodgson and Patrick Bresnahan. Accuracy of airborne lidar-derived elevation. *Photogrammetric Engineering & Remote Sensing*, 70(3):331–339, 2004.

[32] Timothy R Hughes, Matthew J Marton, Allan R Jones, Christopher J Roberts, Roland Stoughton, Christopher D Armour, Holly A Bennett, Ernest Coffey, Hongyue Dai, Yudong D He, et al. Functional discovery via a compendium of expression profiles. *Cell*, 102(1):109–126, 2000.

[33] Pavol Jancura, Dimitrios Mavroeidis, and Elena Marchiori. Deen: A simple and fast algorithm for network community detection. In *Computational Intelligence Methods for Bioinformatics and Biostatistics*, pages 150–163. Springer, 2012.

[34] George Karypis and Vipin Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[35] George Karypis and Vipin Kumar. Parallel multilevel graph partitioning. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 314–319. IEEE, 1996.

[36] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[37] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In *Network and Parallel Computing*, pages 266–275. Springer, 2008.

[38] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[39] Shad Kirmani and Padma Raghavan. Scalable parallel graph partitioning. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 51. ACM, 2013.

[40] Kishore Kothapalli, Sriram V Pemmaraju, and Vivek Sardeshmukh. On the analysis of a label propagation algorithm for community detection. In *Distributed Computing and Networking*, pages 255–269. Springer, 2013.

[41] Karl Kraus and Johannes Otepka. Dtm modelling and visualization–the scop approach. 2005.

[42] Nevan J Krogan, Gerard Cagney, Haiyuan Yu, Gouqing Zhong, Xinghua Guo, Alexandr Ignatchenko, Joyce Li, Shuye Pu, Nira Datta, Aaron P Tikuisis, et al. Global landscape of protein complexes in the yeast saccharomyces cerevisiae. *Nature*, 440(7084):637–643, 2006.

[43] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[44] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: a comparative analysis. *Physical review E*, 80(5):056117, 2009.

[45] Andrea Lancichinetti, Filippo Radicchi, José J Ramasco, Santo Fortunato, et al. Finding statistically significant communities in networks. *PloS one*, 6(4):e18961, 2011.

[46] Min Li, Jianxin Wang, and Jian Er Chen. A fast agglomerate algorithm for mining functional modules in protein interaction networks. In *BioMedical Engineering and Informatics, 2008. BMEI 2008. International Conference on*, volume 1, pages 3–7. IEEE, 2008.

[47] Zhilin Li, Christopher Zhu, and Chris Gold. *Digital terrain modeling: principles and methodology*. CRC press, 2004.

[48] Jen-Chang Liu, Yin-Chen Liang, and Shih-Wei Lin. Selection of canonical images of travel attractions using image clustering and aesthetics analysis. *International Journal of Computational Science and Engineering*, 8(4):324–335, 2013.

[49] Xiaoye Liu and Zhenyu Zhang. Lidar data reduction for efficient and high quality dem generation. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 37:173–178, 2008.

[50] Xiaoye Liu, Zhenyu Zhang, James Peterson, and Shobhit Chandra. The effect of lidar data density on dem accuracy. In *Proceedings of the International Congress on Modelling and Simulation (MODSIM07)*, pages 1363–1369. Modelling and Simulation Society of Australia and New Zealand Inc., 2007.

[51] Xiaoye Liu, Zhenyu Zhang, and Jim Peterson. Evaluation of the performance of dem interpolation algorithms for lidar data. In *Proceedings of the Surveying and Spatial Sciences Institute Biennial International Conference (SSC 2009)*, pages 771–779. Surveying and Spatial Sciences Institute, 2009.

[52] Xiaoye Liu, Zhenyu Zhang, Jim Peterson, and Shobhit Chandra. Lidar-derived high quality ground control information and dem for image orthorectification. *GeoInformatica*, 11(1):37–53, 2007.

[53] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.

[54] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[55] David Lusseau, Karsten Schneider, Oliver J Boisseau, Patti Haase, Elisabeth Slooten, and Steve M Dawson. The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, 54(4):396–405, 2003.

[56] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1055–1064. IEEE, 2015.

[57] Burkhard Monien, Robert Preis, and Ralf Diekmann. Quality matching and local improvement for multilevel graph-partitioning. *Parallel Computing*, 26(12):1609–1634, 2000.

[58] Seunghyeon Moon, Jae-Gil Lee, and Minseo Kang. Scalable community detection from networks by computing edge betweenness on mapreduce. In *Big Data and Smart Computing (BIGCOMP), 2014 International Conference on*, pages 145–148. IEEE, 2014.

[59] Raj Rao Nadakuditi and Mark EJ Newman. Graph spectra and the detectability of community structure in networks. *Physical review letters*, 108(18):188701, 2012.

[60] Mark EJ Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066133, 2004.

[61] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[62] Suely Oliveira and Sang-Cheol Seok. Spectral document clustering algorithms with different data structures. In *CSC*, pages 223–229. Citeseer, 2005.

[63] Suely Oliveira and Sang-Cheol Seok. A matrix-based multilevel approach to identify functional protein modules. *International journal of bioinformatics research and applications*, 4(1):11–27, 2008.

[64] Suely Oliveira and Rahil Sharma. High quality multi-core multi-level community detection algorithm. *International Journal of Computational Science and Engineering*, http://www.inderscience.com/info/ingeneral/forthcoming.php?jcode=ijcse, 2016.

[65] Suely Oliveira and Rahil Sharma. Identification and prediction of functional protein modules using a bi-level community detection algorithm. *International Journal of Bioinformatics, Research and Application*, 12(2):129–148, 2016.

[66] Dossay Oryspayev, Ramanathan Sugumaran, John DeGroote, and Paul Gray. Lidar data reduction using vertex decimation and processing with gpgpu and multicore cpu technology. *Computers & Geosciences*, 43:118–125, 2012.

[67] Peter S Pacheco. *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[68] Tomaž Podobnikar. Suitable dem for required application. In *Proceedings of the 4th International Symposium on Digital Earth*, 2005.

[69] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. In *Computer and Information Sciences-ISCIS 2005*, pages 284–293. Springer, 2005.

[70] Arnau Prat-Pérez, David Dominguez-Sal, Josep M Brunat, and Josep-Lluis Larriba-Pey. Shaping communities out of triangles. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1677–1681. ACM, 2012.

[71] Arnau Prat-Pérez, David Dominguez-Sal, and Josep-LLuis Larriba-Pey. High quality, scalable and parallel community detection for large real graphs. In *Proceedings of the 23rd international conference on World wide web*, pages 225–236. ACM, 2014.

[72] N Pržulj, Derek G Corneil, and Igor Jurisica. Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[73] Nataša Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[74] Filippo Radicchi. A paradox in community detection. *EPL (Europhysics Letters)*, 106(3):38001, 2014.

[75] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 101(9):2658–2663, 2004.

[76] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[77] Mohammad S Rahman and Alioune Ngom. A fast agglomerative community detection method for protein complex discovery in protein interaction networks. In *Pattern Recognition in Bioinformatics*, pages 1–12. Springer, 2013.

[78] Anand Rajaraman, Jeffrey D Ullman, Jeffrey David Ullman, and Jeffrey David Ullman. *Mining of massive datasets*, volume 77. Cambridge University Press Cambridge, 2012.

[79] Emad Ramadan, Christopher Osgood, and Alex Pothen. The architecture of a proteomic network in the yeast. In *Computational Life Sciences*, pages 265–276. Springer, 2005.

[80] J Raul Ramirez. A new approach to relief representation. *Surveying and Land Information Science*, 66(1):19–25, 2006.

[81] R Real. Tables of significant values of jaccard's index of similarity. *Miscel· lània Zoològica*, 22(1):29–40, 1999.

[82] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[83] Inna Rytsareva, Timothy Chapman, and Ananth Kalyanaraman. Parallel algorithms for clustering biological graphs on distributed and shared memory architectures. *International Journal of High Performance Computing and Networking*, 7(4):241–257, 2014.

[84] Jeffrey Shafer, Scott Rixner, and Alan L Cox. The hadoop distributed filesystem: Balancing portability and performance. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 122–133. IEEE, 2010.

[85] Rahil Sharma and Suely Oliveira. Community detection algorithm for massive social networks using hybrid architecture. *Submitted (In review)*, 2016.

[86] Rahil Sharma, Zewei Xu, Ramanathan Sugumaran, and Suely Oliveira. Parallel landscape driven data reduction & spatial interpolation algorithm for big lidar data. *ISPRS International Journal of Geo-Information*, 5(6):97, 2016.

[87] G Sithole and G Vosselman. The full report: Isprs comparision of filters [ol], 2003.

[88] Jyothish Soman and Ankur Narang. Fast community detection algorithm with gpus and multicore architectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 568–579. IEEE, 2011.

[89] Victor Spirin and Leonid A Mirny. Protein complexes and functional modules in molecular networks. *Proceedings of the National Academy of Sciences*, 100(21):12123–12128, 2003.

[90] Peter Uetz, Loic Giot, Gerard Cagney, Traci A Mansfield, Richard S Judson, James R Knight, Daniel Lockshon, Vaibhav Narayan, Maithreyan Srinivasan, Pascale Pochart, et al. A comprehensive analysis of protein–protein interactions in saccharomyces cerevisiae. *Nature*, 403(6770):623–627, 2000.

[91] Vinícius da F Vieira, Carolina R Xavier, Nelson FF Ebecken, and Alexandre G Evsukoff. Modularity based hierarchical community detection in networks. In *Computational Science and Its Applications–ICCSA 2014*, pages 146–160. Springer, 2014.

[92] Christian Von Mering, Roland Krause, Berend Snel, Michael Cornell, Stephen G Oliver, Stanley Fields, and Peer Bork. Comparative assessment of large-scale data sets of protein–protein interactions. *Nature*, 417(6887):399–403, 2002.

[93] Charith Wickramaarachchi, Marc Frincu, Patrick Small, and Viktor K Prasanna. Fast parallel algorithm for unfolding of communities in large graphs. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.

[94] Hui Xiong, Xiaofeng He, Chris HQ Ding, Ya Zhang, Vipin Kumar, and Stephen R Holbrook. Identification of functional modules in protein complexes via hyper-clique pattern discovery. In *Pacific symposium on biocomputing*, volume 10, pages 221–232. World Scientific, 2005.

[95] Jaewon Yang and Jure Leskovec. Overlapping community detection at scale: a nonnegative matrix factorization approach. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 587–596. ACM, 2013.

[96] Soon-Hyung Yook, Zoltán N Oltvai, and Albert-László Barabási. Functional and topological characterization of protein interaction networks. *Proteomics*, 4(4):928–942, 2004.

[97] Wayne W Zachary. An information flow model for conflict and fission in small groups. *Journal of anthropological research*, pages 452–473, 1977.

[98] Lan V Zhang, Sharyl L Wong, Oliver D King, and Frederick P Roth. Predicting co-complexed protein pairs using genomic and proteomic data integration. *BMC bioinformatics*, 5(1):38, 2004.

[99] Dale Zimmerman, Claire Pavlik, Amy Ruggles, and Marc P Armstrong. An experimental comparison of ordinary and universal kriging and inverse distance weighting. *Mathematical Geology*, 31(4):375–390, 1999.