Parallel Mining and Analysis of Triangles and Communities in Big Networks

S M Arifuzzaman

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Madhav V. Marathe, Chair
Md-Abdul M. Khan, Co-chair
Lenwood S. Heath
Ali Pinar
Anil Kumar S. Vullikanti

July 22, 2016
Blacksburg, Virginia

Keywords: Network Mining, Parallel Algorithm, Triangle Counting, Community Detection, Big Data
Copyright 2016, S M Arifuzzaman

Parallel Mining and Analysis of Triangles and Communities in Big Networks

S M Arifuzzaman

(ABSTRACT)

A network (graph) is a powerful abstraction for interactions among entities in a system. Examples include various social, biological, collaboration, citation, and co-purchase networks. Real-world networks are often characterized by an abundance of triangles and the existence of well-structured communities. Thus, counting triangles and detecting communities in networks have become important algorithmic problems in network mining and analysis. In the era of big data, the network data emerged from numerous scientific disciplines are very large. Online social networks such as Twitter and Facebook have millions to billions of users. Such massive networks often do not fit in the main memory of a single machine, and the existing sequential methods might take a prohibitively large runtime. This motivates the need for scalable parallel algorithms for mining and analysis.

We design MPI-based distributed-memory parallel algorithms for counting triangles and detecting communities in big networks and present related analysis. The dissertation consists of four parts. In Part I, we devise parallel algorithms for counting and enumerating triangles. The first algorithm employs an overlapping partitioning scheme and novel load-balancing schemes leading to a fast algorithm. We also design a space-efficient algorithm using non-overlapping partitioning and an efficient communication scheme. This space efficiency allows the algorithm to work on even larger networks. We then present our third parallel algorithm based on dynamic load balancing. All these algorithms work on big networks, scale to a large number of processors, and demonstrate very good speedups. An important property, very related to triangles, of many real-world networks is high transitivity, which states that two nodes having common neighbors tend to become neighbors themselves. In Part II, we characterize networks by quantifying the number of common neighbors and demonstrate its relationship to community structure of networks. In Part III, we design parallel algorithms for detecting communities in big networks. We propose efficient load balancing and communication approaches, which lead to fast and scalable algorithms. Finally, in Part IV, we present scalable parallel algorithms for a useful graph preprocessing problem– converting edge list to adjacency list. We present non-trivial parallelization with efficient HPC-based techniques leading to fast and space-efficient algorithms.

Dedication

Dedicated to my parents Abdur Rahman Sheikh and Aleya Begum, for their endless love, encouragement, and support.

Acknowledgments

I have enjoyed an exciting, humbling, and enriching journey throughout my years as a Ph.D. student at Virginia Tech. I am blessed to have had many wonderful individuals in my academic and personal circles, who have offered me guidance, support, and encouragement, and helped me complete this dissertation. I would like to sincerely thank them all.

First and foremost, I would like to express my deepest appreciation to my advisors Drs. Madhav Marathe and Maleq Khan, for their continuous guidance, effective encouragement, and valuable support throughout my graduate study. I am grateful to Dr. Madhav Marathe for taking time from his busy schedule to provide me with constructive suggestions about research problems, presentation skill, and dissertation document. I also thank him for his advice regarding my academic progress and career aspiration. He always made himself available to discuss any problems and help sort them out. I am also grateful to him for offering me the opportunity to work in his lab when I joined the Ph.D. program at Virginia Tech.

I would like to specially thank Dr. Maleq Khan for his enthusiastic and active participation in developing the ideas in this dissertation. He has supported me in every aspect of my Ph.D. study, from advising for coursework to helping me build an effective research habit. He has always been accessible whenever I needed his advice. He has spent a substantial amount of time, selflessly, to guide this dissertation. I also thank him for his invaluable guidance in writing our research papers, which helped me improve my writing skill. In addition to guiding my research, Dr. Maleq Khan cared for my personal and professional growth with great thoughtfulness. I have been very fortunate to have an excellent advisor and mentor like him.

I would like to thank other members of my dissertation committee, Drs. Lenwood Heath, Ali Pinar, and Anil Vullikanti, for their valuable feedback to improve the dissertation. They have always been accessible to discuss any problems and offer suggestions. I would like to thank Dr. Lenwood Heath for carefully reviewing the draft of this dissertation, which helped greatly in improving the presentation of the dissertation. He has also provided valuable advice regarding the future directions of my research. I am also indebted to Dr. Ali Pinar for mentoring me during my summer internship at Sandia National Laboratories and actively guiding me for a part of this dissertation. I have benefited a lot from his scholarly comments and instructions. I am immensely inspired by his knowledge and wisdom in the subject area of this research. I am grateful to Dr. Anil Vullikanti for offering me useful insights about alternate technical approaches for several problems. He has always provided me encouraging remarks, thoughtful follow-up comments, and valuable suggestions for future work.

I am specially thankful to Drs. Madhav Marathe, Maleq Khan, and Anil Vullikanti for their advice and assistance during my job search. I would not have been able to secure an academic position without the kind support from them. I appreciate their time for writing recommendation letters and offering me practical guidance. I am specially grateful to Dr. Maleq Khan for his feedback on my application documents and presentation slides.

I also thank the anonymous referees of the papers [7, 8, 9, 10] containing the results of this dissertation published in various conferences. Their suggestions and detailed comments helped greatly in improving the presentation of the results.

My heartfelt gratitude goes to my wonderful family for their love, blessing, and support throughout my life, and I cannot thank them enough. I would like to particularly mention my parents, who will be the happiest persons in the world at the completion of this dissertation. It is my father, a teacher by profession, who guided me in my early stages of education and has always inspired me to dream big. It is my mother from whom I have learned to practice patience and resilience in tough times. Thank you, *Abbu* and *Ammu*.

I would like to thank my labmates in the Network Dynamics and Simulation Science Lab. I benefited greatly from the discussions and interactions with my labmates. I am also grateful to the Bangladeshi community at Virginia Tech for their inspiration and friendship. I am equally thankful to the Virginia Tech (and Blacksburg) community for making my stay here a pleasant one. I also thank all my friends at home and abroad, who always believe in me and celebrate my success with a big smile.

Contents

Chapter 1 Introduction **1**

1.1 Network Mining and Analysis 1

1.2 Research Problems 2

1.3 Organization and Contributions 3

I Counting and Listing Triangles in Big Networks **6**

Chapter 2 Introduction to Counting Triangles **7**

2.1 Related Work 7

2.2 Our Contributions 8

2.3 Preliminaries 9

Chapter 3 PATRIC: An Efficient Parallel Algorithm for Counting Triangles in Massive Networks **12**

3.1 Introduction 12

3.2 Sequential Algorithms 13

3.2.1 Optimal Node Ordering 16

3.3 The Parallel Algorithm 20

3.3.1 Overview of the Algorithm 20

3.3.2 Partitioning the Network 20

3.3.3 Counting Triangles 21

3.3.4 Load Balancing in PATRIC 22

3.3.5 Performance Analysis 27

3.4 A Sparsification-based Parallel Approximation Algorithm 30

3.5 Conclusion 33

Chapter 4 A Space-efficient Parallel Algorithm for Counting the Exact Number of Triangles in Massive Networks **34**
4.1 Introduction 34
4.2 A space-efficient Parallel Algorithm for Counting Triangles 36
4.2.1 Overview of the Algorithm. 36
4.2.2 An Efficient Communication Approach 36
4.2.3 Pseudocode for Counting Triangles. 38
4.2.4 Partitioning and Load Balancing 38
4.2.5 Correctness of the Algorithm 41
4.2.6 Analysis of the Number of Messages 41
4.3 Experimental Evaluation 42
4.4 Sparsification-based Parallel Approximation Algorithms 45
4.5 Conclusion 46
Chapter 5 A Fast Parallel Algorithm for Counting Triangles in Networks using Dynamic Load Balancing **47**
5.1 Introduction 47
5.2 Comparison with Related Parallel Algorithms 48
5.3 A Fast Parallel Algorithm with Dynamic Load Balancing 50
5.3.1 Overview of the Algorithm 50
5.3.2 An Efficient Dynamic Load Balancing Scheme 51
5.3.3 Counting Triangles 52
5.3.4 Correctness of the Algorithm 52
5.3.5 Performance 53
5.4 Conclusion 58
Chapter 6 Applications of Our Algorithms for Counting Triangles **59**
6.1 Listing Triangles in Graphs 59
6.2 Computing Clustering Coefficient of Nodes 60
6.3 Other Applications for Counting Triangles 61

II Characterizing Networks Based on Common Neighbor Statistics 64

Chapter 7 How Much Common Neighbors Can Reveal about Networks 65

7.1 Introduction 65

7.2 Preliminaries 67

7.3 Computing Jaccard Index and Transition Plots 68

7.3.1 Computing Jaccard Index 68

7.3.2 Transition Plots 69

7.3.3 Transition Plots for Variety of Networks 70

7.3.4 An Alternative Justification of the Threshold 71

7.4 Other Implications of Threshold Behavior 71

7.4.1 Contrasting Bi-partitions 72

7.4.2 Random Network Models and the Threshold Behavior 72

7.5 Common Neighbors and Communities 77

7.5.1 Common Neighbor Distribution in Networks 77

7.5.2 Clustering Coefficients, Community Size and Degree Distribution 77

7.6 Characterizing Networks Based on Jaccard Statistics 80

7.6.1 Predicting Classes from Jaccard Statistics 80

7.6.2 Regression Analysis on Community Sizes and Jaccard Statistics 82

7.7 Conclusion 83

III Community Detection in Big Networks 85

Chapter 8 PASCL: Parallel Algorithms for Scalable Community Detection in Large Networks 86

8.1 Introduction 86

8.1.1 Background of Community Detection 87

8.1.2 Challenges with Massive Networks 88

8.2 Related Work on Parallel Algorithms 88

8.3 Fast and Scalable Parallel Algorithms for Community Detection 89

8.3.1 Sequential Louvain Algorithm 89

8.3.2 Overview of Our Parallel Algorithm 90

8.3.3 Partitioning . . . 91
8.3.4 Local Computing of Community Labels . . . 91
8.3.5 Renumbering Community Labels . . . 92
8.3.6 Constructing Supergraph . . . 93
8.4 Label Propagation Algorithm . . . 94
8.5 Evaluation of Our Parallel Algorithms . . . 96
8.5.1 Load Balancing and Scalability . . . 96
8.5.2 Trading off the Quality and Speed of our Community Detection Algorithms 97
8.5.3 Parallel Sparsification Algorithm . . . 98
8.5.4 Comparison with Other Algorithms . . . 99
8.6 Conclusion . . . 99

IV Converting Edge List to Adjacency List 100

Chapter 9 Fast Parallel Conversion of Edge List to Adjacency List for Large-Scale Graphs 101

9.1 Introduction . . . 101
9.2 Preliminaries and Background . . . 103
9.2.1 Basic Definitions . . . 103
9.2.2 A Sequential Algorithm . . . 103
9.3 The Parallel Algorithm . . . 104
9.3.1 Overview of the Algorithm . . . 104
9.3.2 (Phase 1) Local Processing . . . 105
9.3.3 (Phase 2) Merging Local Adjacency Lists . . . 105
9.3.4 Partitioning and Load Balancing . . . 108
9.4 Performance Analysis . . . 111
9.4.1 Load Distribution . . . 112
9.4.2 Strong Scaling . . . 112
9.4.3 Comparison between Message-based and External-memory Merging . . . 113
9.4.4 Weak Scaling . . . 113
9.5 Conclusion . . . 113

Chapter 10 General Conclusion **115**

Bibliography **116**

List of Figures

3.1 Algorithm NodeIterator++, where $\prec$ is the degree based ordering of the nodes defined in Equation 3.1. 14

3.2 Algorithm NodeIteratorN, a modification of NodeIterator++. 14

3.3 Comparison of runtime of sequential triangle counting (NodeIteratorN) with four distinct orderings of nodes. For each network, we compute the percentage of runtime with respect to the maximum runtime given by any of these orderings. In all cases, the degree based ordering gives the least runtime. Note that we compute the average runtime from 25 independent runs for the random ordering. 17

3.4 The main steps of our fast parallel algorithm. 20

3.5 Memory usage with optimized and non-optimized data storing. 21

3.6 Algorithm executed by processor P_i to count triangles in $G_i(V_i, E_i)$. 22

3.7 A network with a skewed degree distribution: $d_{v_0} = n - 1$, $d_{v_{i \neq 0}} = 3$. 23

3.8 Speedup with equal number of core nodes in all processors. 23

3.9 Computing load of individual processors (equal number of core nodes). 24

3.10 Load balancing cost for LiveJournal network with different schemes. 24

3.11 Load distribution among processors for LiveJournal, Miami and Twitter networks by different schemes. 27

3.12 Speedup gained from different load balancing schemes for LiveJournal, Miami and Twitter networks. 28

3.13 Weak scaling on PA($P/10 \times 1M, 50$) networks. 28

3.14 Improved scalability with increased network size. 30

3.15 Two triangles (v, u, w) and (v', u, w) with an overlapping edge. 31

3.16 Counting the number of triangles in a network with our parallel sparsification method. 31

4.1 The procedure executed by P_i after receiving message $\langle data, X \rangle$ from some P_j. . . 38

4.2 An algorithm for counting triangles using surrogate approach. Each processor P_i executes Line 1-22. After that, they are synchronized, and the aggregation is performed (Line 24-25). 39

4.3 Runtime reported by various algorithms for counting triangles in Twitter network. 43

4.4 Speedup factors of our algorithm with both direct and surrogate approaches. 43

4.5 Improved scalability of our algorithm with increasing network size. 44

4.6 Comparison of the cost function $f(v)$ estimated for our algorithm with non-overlapping partitioning and the best function $g(v)$ in Chapter 3. 44

4.7 Weak scaling of our algorithm, experiment performed on PA($t/10 * 1M, 50$) networks, t = number of processors used. 45

5.1 A procedure executed by processor P_i to count triangles corresponding to the task $\langle v, t \rangle$. 53

5.2 An algorithm for counting triangles with dynamic load balancing. 54

5.3 Speedup factors of our algorithm on Miami, LiveJournal and web-BerkStan networks with both $f(v) = 1$ and $f(v) = d_v$ cost functions. 55

5.4 Runtime required by processors (rankwise) with both static tasks and dynamic adjustment of task granularity. 55

5.5 Our algorithm with dynamic load balancing shows improved scalability with increasing network size. Further, this algorithm achieves higher speedups than PATRIC (in Chapter 3). 56

5.6 Weak scaling of our algorithm. We perform this experiment on PA($t/10 * 1M, 50$) networks, t = number of processors used. 56

5.7 Comparison of speedup factors of our algorithm with [8] and [9] on Miami and LiveJournal networks. 57

6.1 Listing triangles after performing the set intersection operation for counting triangles. 59

6.2 Tracking local counts by processor P_i. Each triangle (v, u, w) is detected by the triangle listing algorithm shown in Figure 6.1. 61

6.3 Aggregating local counts for $v \in V_i^c$ by P_i. 61

6.4 Strong scaling of clustering coefficient algorithm with both *AOP* and *ANOP* on LiveJournal and Twitter networks. 62

6.5 Weak scaling of the algorithms for computing clustering coefficient (CC) and counting triangles (TC). 62

7.1 Algorithm for computing all-pair Jaccard indices with wedge enumeration. Pairs with a Jaccard index of 0 are omitted. 68

7.2 Transition curve for Jaccard indices for Astrophysics collaboration network. . . . 69

7.3 Transition curve for Jaccard indices on social-network-like graphs. 70

7.4 Transition curve for Jaccard indices on non social-network-like graphs. 70

7.5 Change of the prediction performance in terms of F_1 scores by varying the threshold of Jaccard indices on networks with social structures. 72

7.6 Degree distribution of two contrasting partitions– partitions with weak and strong edges, respectively, with strength determined by Jaccard index threshold=0.1. . . . 73

7.7 Degree distribution of two contrasting partitions– partitions with weak and strong edges, respectively, with strength determined by Jaccard index threshold=0.1. . . . 74

7.8 Jaccard transition curve of AstroPhysics Network. 75

7.9 Jaccard transition curve of the BTER graph constructed from the same degree distribution and degree-wise CC of AstroPhysics Network. 75

7.10 Jaccard transition curve of ER graph Gnp(1k, 10k). 75

7.11 Edge probability $p(k) = 1 - (1 - c)^k$ with varying c. 75

7.12 Edge probability $p(k) = 1/(1 + e^{-k})$, a sigmoid function. 75

7.13 Edge probability $p(k) = 1/(1 + e^{-k})$ for positive k. 75

7.14 Jaccard transition curves for networks with 1000 nodes and 10000 edges generated with $p(k) = 1 - (1 - c)^k$ and varying c, where c is the input average clustering coefficient (CC-in). 76

7.15 Average CC of the generated networks (CC-out) as compared to the input value (CC-in) of c in the function $1 - (1 - c)^k$. 76

7.16 Average CC-out in the generated network with varying the multiple a in $p(k) = 1 - (1 - c)^{ak}$ and CC-in=0.5. 76

7.17 Average CC-out with varying number of edges in the generated network with $p(k) = 1 - (1 - c)^k$ and CC-in=0.5. Larger graph with same setting has larger average CC. 76

7.18 Jaccard transition curve for the network generated with $P(k) = 1/(1 + e^{-4k})$ (sigmoid function). 76

7.19 Average CC-out with varying the constant a in the sigmoid function $P(k) = 1/(1+e^{-ak})$. 76

7.20 Wedge distribution (equivalently, common neighbors distribution) curves for networks with communities. 77

7.21 Wedge distribution curves for a network with partial community structure (in a) and for networks without communities (in b and c). 78

7.22 Jaccard transition curve for the CL network generated from the degree distribution of AstroPhysics network. 80

7.23 Wedge distribution for the CL network generated from the degree distribution of AstroPhysics network.. 80

7.24 The predicted versus actual plot (left) and the residual by predicted plot (right) of the regression analysis on a set of LFR networks. These networks have 10000 nodes, an average degree of 40, community sizes varying from 50 to 500, and mixing parameter 0.2. 83

7.25 Mixing parameter versus the accuracy with our regression model with LFR networks.. 83

7.26 Regression diagnostic plots for our analysis on real-world networks: the predicted versus actual plot (left) and the residual by predicted plot (right). 84

8.1 Pseudocode of the sequential Louvain algorithm. $C[v]$ is the community label of node v. The quantity $\triangle mod(v, C[v] \rightarrow C[u])$ denotes the difference in modularity when node v is moved from $C[v]$ to a neighboring community $C[u]$. 90

8.2 Pseudocode for our parallel Louvain algorithm. 95

8.3 Laod distribution for Miami network with equal number of nodes and edges per processors. 96

8.4 Laod distribution for LiveJournal network with equal number of nodes and edges per processors. 96

8.5 Speedups of our parallel Louvain algorithm on Miami and LiveJournal networks. . 97

8.6 Global sparsification of a network in parallel. 98

8.7 Local sparsification of a network in parallel. 99

9.1 The edge list and adjacency list representations of an example graph with 5 nodes and 6 edges. 103

9.2 Sequential algorithm for converting edge list to adjacency list. 104

9.3 Algorithm for performing Phase 1 computation. 105

9.4 Parallel merging with the binary tree scheme ($P = 7$). Numbers in the circle denote rank of the processors. 106

9.5 Parallel algorithm for merging local adjacency lists to construct final adjacency lists N_v. A message, denoted by $< v, N_v^i >$, refers to local adjacency lists of v in processor i. 107

9.6 Load distribution among processors for LiveJournal, Miami and Twitter before applying the load balancing scheme. 108

9.7 Parallel algorithm executed by each processor i for computing $f(v) = d_v$. 109

9.8 Load distribution among processors for LiveJournal, Miami and Twitter networks by different schemes. 111

9.9 Strong scaling of our algorithm on LiveJournal, Miami and Twitter networks with and without load balancing scheme. Computation of speedup factors includes the cost for load balancing. 112

9.10 Weak scaling of our parallel algorithm. For this experiment we use networks PA($x/10 \times 1M, 20$) for x processors. 114

List of Tables

2.1 Datasets used in the experimental evaluation of our algorithms. 10

3.1 Running time for the two sequential algorithms for counting triangles, NodeIterator++ and NodeIteratorN. 16

3.2 Cost functions $f(.)$ for our load balancing schemes. 25

3.3 Runtime performance of PATRIC using 200 processors and the algorithm in [72]. . 29

3.4 Accuracy of our parallel sparsification algorithm and DOULION [76] with $q = 0.1$. Our parallel algorithm was run with 100 processors. Variance, max error and average error are calculated from 25 independent runs for each of the algorithms. . 32

3.5 Comparison of our parallel sparsification algorithm and DOULION [76] on LiveJournal network with 100 processors. 32

4.1 Memory usage of our algorithms (size of the largest partition) with both overlapping and non-overlapping partitioning. Number of partitions used is 100. 35

4.2 Number of messages exchanged in Direct and Surrogate approaches. 42

4.3 Runtime performance of our algorithms *AOP* and *ANOP*. We used 200 processors for this experiment. We showed both direct and surrogate approaches for *ANOP*. . 43

4.4 Accuracy of our parallel sparsification algorithm and DOULION [76] with $q = 0.1$. Our parallel algorithm was run with 100 processors. Variance, max error and average error are calculated from 25 independent runs for each of the algorithms. The best values for each attribute are marked as bold. 45

4.5 Comparison of accuracy between our parallel sparsification algorithms and DOULION on one realistic synthetic and three real-world networks with 100 processors. The best values for each q are marked as bold. 46

5.1 Memory required for storing networks along with their average and maximum degree statistics. 49

5.2 Trade-off between space and runtime efficiency of algorithms in [8, 9] and this chapter. 50

5.3 Runtime performance of our algorithm and algorithm [8]. 56

6.1 Comparison of the number of triangles ($\triangle$) and normalized triangle count (NTC) in various networks. We used both artificially generated and real-world networks. . 63

7.1 Datasets used in our experiments. 67

7.2 Jaccard indices that achieve the maximum F_1 scores for several Facebook networks. 72

7.3 Accuracies for predicting edges based on the optimum Jaccard index J_{tr} achieved from the training data in Table 7.2. 73

7.4 Comparison of m, the number of triangles $\triangle$, maximum degree d_{max}, and average degree d_{avg} in the network induced by weak edges $G_{<t=0.1}$ and the Chung-Lu network G_{cl} constructed with the same degree distribution as $G_{<t=0.1}$. The weak edges are the edges with Jaccard indices < 0.1. 73

7.5 Comparison of m, the number of triangles $\triangle$, maximum degree d_{max}, and average degree d_{avg} in the network $G_{<t=0.1}$ induced by weak edges and the network $G_{>t=0.1}$ induced by strong edges. The weak and strong edges are determined based on the Jaccard index < 0.1. 74

7.6 Class assignments according to the largest community in the networks. 81

7.7 Class assignments according to the modularity values obtained for the networks. . . 81

8.1 Comparison of modularity and runtime between parallel LPA and Louvain Algorithm. 97

8.2 Modularity and runtime with various sparsification method on different networks. . 99

9.1 Number of messages received in practice compared to the theoretical bounds. This results report $\max_i M_i$ with $P = 50$. 110

9.2 Comparison of external-memory (EXT) and message-based (MSG) merging (using 50 processors). 113

Chapter 1

Introduction

Data from diverse fields are modeled as networks (graphs) nowadays because of their convenience in representing underlying relations and structures [23]. Some significant examples are the Web [18], various social networks such as Facebook and Twitter [41], collaboration and co-authorship networks [50], infrastructure networks such as road networks [69], and many forms of biological networks [32]. Networks are studied across many fields of science including physics [25], biology [20, 36], finance [6], economics, and social science [11, 47, 78]. One rich aspect of such study is network mining and analysis. The goal is to find structures or patterns in networks and reveal properties that govern the construction and evolution of these networks. Thus, mining and analyzing networks help researchers and practitioners to understand and improve the corresponding systems.

With the unprecedented advancement of computing and data technology, we are deluged with massive data from diverse areas such as business and finance [6], computational biology [20], and social science [11]. In the era of big data, the network data emerged from those areas are very large. The Web has over 1 trillion web pages. Most social networks, such as Twitter and Facebook, have millions to billions of users [22]. The emergence of such big data poses non-trivial challenges for network analysts. These networks often do not fit in the main memory of a single machine. Further, the existing sequential algorithms might take a prohibitively large runtime to process such networks. To analyze the large quantities of data represented by massive networks, space-efficient and scalable parallel algorithms [52, 72] are necessary.

1.1 Network Mining and Analysis

There are several prominent areas of interest pertaining to network mining. First, find structures and properties associated with real-world networks [15, 25, 33, 39, 47, 49, 65, 75, 77]: these graphs are often characterized by an abundance of triangles and the existence of well-structured communities. Second, discover interesting or frequent subgraphs [21, 48]: some research work along this line is directed to identifying candidate subgraphs in a computationally efficient way. Third, find general statistics of networks [13, 18, 23, 51]: researchers have been interested in degree distribution, diameter, or eigenvalues with a focus on designing efficient algorithms. Last,

mine time-evolving networks [43, 45]: time-evolving networks arise in multiple application areas. These networks characterize information flow in communication networks [35], phases of pathway switching in gene interaction networks [74], or a changing collaboration network of a field over years [50]. One prominent direction is to explore how various properties or metrics of such networks change over time [45]. In this dissertation, we focus on the first of the above areas: mining and analyzing static real-world networks, mostly social networks. We are mainly interested in triangles and communities from an algorithmic and analytic perspective. To deal with the challenges emerged from big data, we design parallel algorithms and high performance computing techniques, which are scalable and space-efficient.

1.2 Research Problems

We address the following research problems pertaining to mining and analysis of big real-world networks. Efficient solutions to these problems are crucial to understanding interesting properties and revealing useful insights about such networks.

Counting and Listing Triangles in Big Networks. Counting triangles [15, 19, 38, 53] in a network is an important algorithmic problem arising in the study of complex networks. An efficient solution to the triangle counting problem can also lead to efficient solutions for many other graph theoretic problems, e.g., computation of clustering coefficient, transitivity, and triangular connectivity [22, 23, 48] in networks. The existence of triangles and the resulting high clustering coefficient in a social network reflect the common theory of social science that people who have common friends tend to be friends themselves [47]. Further, triangle counting has important applications in network science and database. Recently, it has been used to detect spamming activity and assess content quality in social networks [15], to uncover the thematic structure of the web [30], and query planning optimization in databases [12]. In this dissertation, we address the problem of counting triangles and computing clustering coefficients and present efficient parallel algorithms and related analysis.

Characterizing Networks Based on Common Neighbor Statistics. Characterizing real-world social and information networks based on graph theoretic metrics or properties has been of growing interest [39, 43, 44, 49]. Among the most explored metrics are degree distribution, number of triangles, and clustering coefficients. An important property related to triangles, of many real-world networks, is high transitivity [49], which states that two nodes (vertices) having common neighbors tend to become neighbors themselves [47]. However, there is no quantifiable analysis in this regard. Specifically, we do not know how much the number of common neighbors can tell about those nodes becoming neighbors. Further, there has been an interest in learning how common neighbor statistics relate to community structure of networks [33]. In this dissertation, we characterize a network based on a quantification of common neighbors of pairs of nodes. We also demonstrate how common neighbor statistics relate to community structures of networks.

Community Detection in Big Networks. Complex systems are organized in clusters or communities [16, 24, 49], each having a distinct role or function. In the corresponding network representation, each functional unit appears as a dense set of nodes having higher connection inside the set

than outside. Finding communities may reveal the organizational information of a complex system. For instance, a community is often interpreted as a social clique or group in contact and friendship networks, a functional unit in biological networks, or a scientific discipline in citation networks [46]. Thus detecting communities in social and information networks has become an interesting and fundamental problem in network science [52, 77]. In this dissertation, we deal with the problem of scalable community detection in big networks and design efficient parallel algorithms and perform useful analysis pertaining to the speed and quality of such detection.

Network Data Preprocessing Problem– Converting Edge List to Adjacency List. In most cases, network data are represented as lists of edges (*edge list*). However, many graph algorithms work efficiently when information of the adjacent nodes (*adjacency list*) for each node is readily available. For example, computing shortest path, breadth-first search, and depth-first search are executed by exploring the neighbors (adjacent nodes) of a node. Although the conversion from edge list to adjacency list is not complicated for small networks, such conversion becomes challenging for the emerging large-scale networks consisting of billions of nodes and edges. In this dissertation, we present scalable parallel algorithms for this network preprocessing problem by designing efficient high performance computing techniques.

1.3 Organization and Contributions

The dissertation is organized into ten chapters. We present an introduction to this dissertation in **Chapter 1** (the ongoing chapter). Our main technical contributions for the aforementioned four research problems are organized into four parts, each containing one or more chapters.

In **Part I**, we devise distributed memory parallel algorithms for counting and listing triangles. We also present pertinent theoretical analysis and demonstrate applications for our parallel algorithms. Chapter 2 to 6 constitute Part I of this dissertation.

In **Chapter 2**, we provide an introduction to the research efforts for solving the problem of counting triangles. We also introduce notations, datasets, computational model, and experimental setup for Part I.

In **Chapter 3**, we present our first parallel algorithm for counting triangles in massive networks. The algorithm employs an overlapping partitioning scheme and does not require any inter-processor communication, leading to a fast algorithm. We present and analyze several schemes for balancing load among processors for the triangle counting problem. These schemes achieve very good load balancing. We also show how our parallel algorithm can adapt an existing edge sparsification technique to approximate the number of triangles with very high accuracy. Moreover, we show that a simple modification of a state-of-the-art sequential algorithm for counting triangles improves both running time and space requirement by significant factors. We use this modified sequential algorithm as a basis for our parallel algorithm.

In **Chapter 4**, we present a space-efficient parallel algorithm for counting the *exact* number of triangles in massive networks. The algorithm divides the network into non-overlapping partitions. Our results demonstrate significant space saving over the algorithm with overlapping partitions.

This space efficiency allows the algorithm to deal with larger networks. We present a novel approach that reduces communication cost drastically leading to both a space- and runtime-efficient algorithm. Our adaptation of a parallel partitioning scheme by computing a novel weight function adds further to the efficiency of the algorithm.

In **Chapter 5**, we present another efficient parallel algorithm for counting triangles in large networks. We consider the case where the main memory of each compute node is large enough to contain the entire network. We observe that, for such a case, computation load can be balanced dynamically and present a dynamic load balancing scheme that improves the performance of the algorithm significantly. Our algorithm demonstrates very good speedups and scales to a large number of processors. Our results demonstrate that the algorithm is significantly faster than the related algorithms with static partitioning.

In **Chapter 6**, we provide several applications of our algorithms for counting triangles presented in earlier chapters. Among others, we present how our algorithms can be used to enumerate triangles and compute clustering coefficients of nodes. In a sequential setting, an algorithm for counting triangles can be directly used for computing clustering coefficients of the nodes by simply keeping the counts of triangles for each node individually. However, in a distributed-memory parallel system, combining the counts from all processors for all nodes poses another level of difficulty. We show how our algorithm for triangle counting can be used to compute clustering coefficients in parallel.

In **Part II**, we characterize networks by quantifying the number of common neighbors and demonstrate the relationship with other network properties. In **Chapter 7**, among others, we answer the following questions: how much does the number of common neighbors tell about forming an edge between two nodes? How do common neighbor statistics relate to community structure of networks? Based on the Jaccard indices of edges, we observe that there is an interesting threshold behavior of two nodes connected by an edge in the social and information networks we examined. We present various analyses to reveal how common neighbor statistics (represented by Jaccard indices) relate to global properties or features of networks. Finally, we demonstrate how such statistics relate to community structure of networks.

In **Part III**, we devise distributed memory parallel algorithms for detecting communities in big networks. These algorithms are based on one of the best sequential algorithms in the literature, namely, the *Louvain* algorithm. We present our parallel algorithms in **Chapter 8**. Although these algorithms are based on an efficient sequential method in literature, its parallelization for distributed-memory systems poses non-trivial challenges. We propose efficient load balancing and communication approaches to address those issues. Our parallel algorithms work on large graphs and scale to a large number of processors. Finally, we also demonstrate how our parallel algorithms can be adapted to come up with even faster computations by incorporating edge sparsification techniques.

In **Part IV**, we address the network preprocessing problem of converting edge list to adjacency list. We present efficient MPI-based distributed memory parallel algorithms for this problem in **Chapter 9**. To address the critical load balancing issue, we present a parallel load balancing scheme that improves both time and space efficiency significantly. Our fast parallel algorithm works on massive graphs, achieves good speedups, and scales to a large number of processors.

We present the concluding remarks of this dissertation in **Chapter 10**.

Part I

Counting and Listing Triangles in Big Networks

Chapter 2

Introduction to Counting Triangles

Counting triangles in a Network is a fundamental and important algorithmic problem in network analysis, and its solution can be used in solving many other problems such as the computation of clustering coefficient, transitivity, and triangular connectivity [22, 48]. Existence of triangles and the resulting high clustering coefficient in a social network reflect some common theories of social science, e.g., *homophily*, where people become friends with those similar to themselves, and *triadic closure*, where people who have common friends tend to be friends themselves [47]. Further, triangle counting has important applications in network mining such as detecting spamming activity and assessing content quality in social networks [15], uncovering the thematic structure of the Web [30], query planning optimization in databases [12], and detecting communities or clusters in social and information networks [57].

2.1 Related Work

Counting triangles and related problems such as computing clustering coefficients have a rich history [4, 34, 42, 55, 65, 68, 72, 76]. Despite the fairly large volume of work addressing this problem, only recently has attention been given to the problems associated with big networks. Several techniques can be employed to deal with such graphs: streaming algorithms [15, 38, 73], sparsification based algorithms [67, 76, 80], external-memory algorithms [22], and parallel algorithms [40, 72, 73]. The streaming and sparsification based algorithms are approximation algorithms. Note that approximation algorithms provide an overall (global) estimate of the number of triangles in the graph, which might not be used to count triangles incident on individual nodes (local triangles) with reasonable accuracy. Thus certain local patterns such as local clustering coefficient distribution can not be computed with approximation algorithms. Exact algorithms are necessary to discover such local patterns. External memory algorithms can provide exact solutions, however they can be very I/O intensive leading to a large runtime. Efficient parallel algorithms can solve the problem of a large runtime by distributing computing tasks to multiple processors. Over the last couple of years, several parallel algorithms, both shared memory and distributed memory (MapReduce or MPI) based, have been proposed.

A shared memory parallel algorithm is proposed in [73] for counting triangles in a streaming setting. The algorithm provides approximate counts. The paper reports scalability using only 12 cores. Two other shared memory algorithms have been presented recently in [60, 68]: the reported speedups with the first algorithm vary between 17 and 50 with 64 cores. The second paper reports speedups using only 32 cores, and the obtained speedups are due to both approximation and parallelization. Although these algorithms are useful, shared memory systems with a large number of processors, and at the same time sufficiently large memory per processor, are not widely available. Further, the overhead for locking and synchronization mechanism required for concurrent read and write access to shared data might restrict their scalability. A GPU-based parallel algorithm is proposed recently in [34], which achieves a speedup of only 32 with 2880 streaming processors.

There exist several algorithms based on the MapReduce framework. In [72], two parallel algorithms for exact triangle counting using the MapReduce framework are presented. The first algorithm generates huge volumes of intermediate data, which are all possible 2-paths centered at each node. Shuffling and regrouping these 2-paths require a significantly large amount of time and memory. The second algorithm suffers from redundant counting of triangles. An improvement of the second algorithm is given in a very recent paper [54]. Although this algorithm reduces the redundant counting to some extent, the redundancy is not entirely eliminated. In fact, for p partitions, the algorithm over-counts (p-1 times) triangles whose nodes lie in the same partition. In another recent work [55], Park et al. propose a randomized MapReduce algorithm for triangle enumeration, which gives an approximate count. Another MapReduce based parallelization of a wedge-based sampling technique [67] is proposed in [40], which is also an approximation algorithm.

The MapReduce framework provides several advantages such as fault tolerance, abstraction of parallel computing mechanisms, and ease of developing a quick prototype or program. However, the overhead for doing so results in a larger runtime. On the other hand, MPI-based systems provide the advantages of defining and controlling parallelism from a granular level, implementing application specific optimizations such as load balancing, memory, and message optimization.

2.2 Our Contributions

In the next three chapters, we present MPI-based parallel algorithms that count the exact number of triangles. We also present related analysis and demonstrate the applicability of these algorithms. We also show how these algorithms can be used for listing all triangles in networks and adapted for designing parallel approximation algorithms. The contributions of **Part I** of this dissertation are summarized below.

i. A fast parallel algorithm with overlapping partitioning (Chapter 3): We propose an MPI based parallel algorithm that employs an overlapping partitioning scheme and a novel load balancing scheme. The overlapping partitions eliminate the need for message exchanges leading to a fast algorithm. The algorithm scales almost linearly with the number of processors, and is able to process a network with 1 billion nodes and 10 billion edges in 16 minutes. To the best of our knowledge, this is the first MPI based parallel algorithm in literature for counting triangles in massive networks.

ii. A space efficient parallel algorithm with non-overlapping partitioning (Chapter 4): We present a space-efficient MPI based parallel algorithm which divides the network into non-overlapping subgraphs and achieves a significant space efficiency over the first algorithm. This algorithm requires inter-processor communications to count a certain type of triangles. However, we present a novel approach that reduces communication cost drastically without requiring additional space, which leads to both a space- and runtime-efficient algorithm. Our adaptation of a parallel partitioning scheme by computing a novel cost function offers additional runtime efficiency to the algorithm.

iii. Sequential algorithm and node ordering (Chapter 3): We show, both theoretically and experimentally, how a simple modification of a state-of-the-art sequential algorithm for counting triangles improves its performance. We use this modified algorithm in the development of our parallel algorithms. We also present a proof of the optimal node ordering that minimizes the computational cost of this sequential algorithm.

iv. Parallel approximation using sparsification technique (Chapter 3 and 4): Although we present algorithms for counting the exact number of triangles in massive graphs, our algorithm can be used for approximate counting in conjunction with an edge sparsification technique [76]. We show how this technique can be adapted to our parallel algorithms and that our parallel sparsification improves the accuracy of the approximation over the sequential sparsification [76].

v. A fast parallel algorithm with dynamic load balancing (Chapter 5): We consider the case where the main memory of each compute node is large enough to contain the entire graph. We observe that, for such a case, computation load can be balanced dynamically and present a dynamic load balancing scheme that improves the performance of our algorithm significantly. This algorithm demonstrates good speedups and scales to a large number of processors. Our results demonstrate that the algorithm is significantly faster than the related algorithms with static partitioning.

vi. Parallel computation of clustering coefficients (Chapter 6): Computing clustering coefficients of nodes requires the count of triangles incident on each node of a network. In a distributed-memory parallel system, combining the counts from all processors for all nodes poses another level of difficulty. We show how our algorithm for triangle counting can be used to compute clustering coefficients in parallel. We also present how our parallel algorithms can be used to list or enumerate all triangles in a network.

2.3 Preliminaries

Below are the notations, definitions, datasets, and experimental setup used in this part.

Basic definitions. We denote a network (graph) by $G(V, E)$, where V and E are the sets of nodes (vertices) and edges, respectively, with $m = |E|$ edges and $n = |V|$ nodes labeled as $0, 1, 2, \ldots, n-1$. We assume that the network is undirected. If $(u, v) \in E$, we say u and v are neighbors of each other. The set of all neighbors of $v \in V$ is denoted by $\mathcal{N}_v$, i.e., $\mathcal{N}_v = \{u \in V | (u, v) \in E\}$. The degree of v is $d_v = |\mathcal{N}_v|$.

Table 2.1: Datasets used in the experimental evaluation of our algorithms.

Network	Nodes	Edges	Source
Email-Enron	37K	0.36M	SNAP [69]
web-Google	0.88M	5.1M	SNAP [69]
web-BerkStan	0.69M	6.5M	SNAP [69]
Miami	2.1M	50M	[14]
LiveJournal	4.8M	43M	SNAP [69]
Twitter	42M	2.4B	[1]
Gnp(n, d)	n	$\frac{1}{2}nd$	Erdős-Réyni [17]
PA(n, d)	n	$\frac{1}{2}nd$	Pref. Attachment [13]

A triangle in G is a set of three nodes $u, v, w \in V$ such that there is an edge between each pair of these three nodes, i.e., $(u, v), (v, w), (w, u) \in E$. The number of triangles containing node v is denoted by T_v. Notice that the number of triangles containing node v is the same as the number of edges among the neighbors of v, i.e.,

$$T_v = |\{(u, w) \in E \mid u, w \in \mathcal{N}_v\}|.$$

The clustering coefficient (CC) of a node $v \in V$, denoted by C_v, is the ratio of the number of edges between neighbors of v to the number of all possible edges between neighbors of v. Then, we have

$$C_v = \frac{T_v}{\binom{d_v}{2}} = \frac{2T_v}{d_v(d_v - 1)}.$$

Let p be the number of processors used in the computation, which we denote by $P_0, P_1, \ldots, P_{p-1}$ where each subscript refers to the rank of a processor.

We use K, M and B to denote thousands, millions and billions, respectively; e.g., 1B stands for one billion.

Datasets. We use both real world and artificially generated networks for the experimental evaluation of our algorithms. A summary of all the networks is provided in Table 2.1. Miami [14] is a synthetic, but realistic, social contact network for the city of Miami. Twitter, LiveJournal, Email-Enron, web-BerkStan, and web-Google are real-world networks. Artificial network PA(n, d) is generated using the preferential attachment (PA) model [13] with n nodes and average degree d. Network Gnp(n, d) is generated using the Erdős-Réyni random graph model [17], also known as $G(n, q)$ model, with n nodes and edge probability $q = \frac{d}{n-1}$ so that the expected degree of each node is d. Both real-world and PA(n, d) networks have very skewed degree distributions. Networks having such distributions create difficulty in partitioning and balancing loads and thus give us a chance to measure the performance of our algorithms in some of the worst case scenarios. Note that, in our experiments, we consider edges of the input graph to be undirected, that is, we ignore the original directionality of edges for web-Google, web-BerkStan, Email-Enron, and LiveJournal networks.

Computation Model. We develop parallel algorithms for message passing interface (MPI) based distributed-memory parallel systems, where each processor has its own local memory. The pro-

cessors do not have any shared memory, one processor cannot directly access the local memory of another processor, and the processors communicate via exchanging messages using MPI.

Experimental Setup. We perform our experiments using a high performance computing cluster with 64 computing nodes (QDR InfiniBand interconnect), 16 processors (Sandy Bridge E5-2670, 2.6GHz) per node, memory 4GB/processor, and operating system CentOS Linux 6.

Chapter 3

PATRIC: An Efficient Parallel Algorithm for Counting Triangles in Massive Networks

In this chapter, we present an efficient MPI-based distributed memory parallel algorithm, called PATRIC (PArallel TRIangle Counting), for counting triangles in massive networks. PATRIC scales well to networks with billions of nodes and can compute the exact number of triangles in a network with one billion nodes and 10 billion edges in 16 minutes. Balancing computational loads among processors for a graph problem like counting triangles is a challenging issue. We present and analyze several schemes for balancing load among processors for the triangle counting problem. These schemes achieve good load balancing. We also show how our parallel algorithm can adapt an existing edge sparsification technique to approximate the number of triangles with high accuracy. This modification allows us to count triangles in even larger networks.

3.1 Introduction

We study the problem of counting triangles in massive networks that do not fit in the main memory of a single computing node. We present MPI-based distributed memory parallel algorithms for these problems, which scale well to networks with billions of nodes and edges. Although substantial research has been done on the triangle counting problem, to the best of our knowledge, very few papers have addressed the problems associated with massive networks that do not fit in the main memory and provide an exact solution. A recent paper [72] presents a parallel algorithm for exact triangle counts using the MapReduce framework [27]. Our parallel algorithm improves the performance, both in time and space, over [72] significantly. A detailed comparison with this algorithm is given in Section 3.3. Our contributions in this chapter are as follows.

- We present a parallel algorithm for counting triangles in massive networks. The algorithm scales almost linearly with the number of processors and is able to process a network with 1 billion nodes and 10 billion edges in 16 minutes using 40 processors. We show the performance of our algorithm by using both artificial and real-world networks.

- We show, both theoretically and experimentally, that a simple modification of a current state of the art sequential algorithm for counting triangles improves its performance. We use this modified algorithm in the development of our parallel algorithm.
- We devise and analyze several load balancing schemes to improve the efficiency of our parallel algorithm. With these schemes, we achieve a very good load balancing, even for networks with skewed degree distributions.
- We show how the sparsification technique presented in [76] can be adapted in our parallel algorithm to have a parallel approximation algorithm. This sparsification technique allows our parallel algorithm to work with even larger networks. Moreover, our parallel sparsification improves the accuracy of the approximation over the sequential sparsification of [76].

The rest of the chapter is organized as follows. We discuss sequential algorithms for counting triangles in Section 3.2. We present our parallel algorithm for triangle counting and the load balancing schemes in Section 3.3. The parallelization of the sparsification technique is given in Section 3.4.

3.2 Sequential Algorithms

In this section, we discuss sequential algorithms for counting triangles using adjacency list representation and show that a simple modification to a state-of-the-art algorithm improves both time and space complexity. Although the modification seems quite simple, and others might have used it previously, our theoretical and experimental analyses of this modification are new. To the best of our knowledge, our analysis is the first to show that such simple modification improves the performance significantly. This modification is also used in our parallel algorithms.

A simple but efficient algorithm [65, 72] for counting triangles is: for each node $v \in V$, find the number of edges among its neighbors, i.e., the number of pairs of neighbors that complete a triangle with vertex v. In this method, each triangle (u, v, w) is counted six times – all six permutations of u, v, and w. Many algorithms exist [22, 42, 65, 66, 72], which provide significant improvement over the above method. A very comprehensive survey of the sequential algorithms can be found in [42, 65]. One of the state of the art algorithms, known as NodeIterator++, as identified in two very recent papers [22, 72], is shown in Figure 3.1. Both [22] and [72] use this algorithm as a basis of their external-memory algorithm and parallel algorithm, respectively.

This algorithm uses a total ordering $\prec$ of the nodes to avoid duplicate counts of the same triangle. Any arbitrary ordering of the nodes, e.g., ordering the nodes based on their IDs, makes sure each triangle is counted exactly once, that is, it counts only one among the six possible permutations. However, the algorithm NodeIterator++ incorporates an interesting node ordering based on the degrees of the nodes, with ties broken by node IDs, as defined below:

$$u \prec v \iff d_u < d_v \text{ or } (d_u = d_v \text{ and } u < v). \tag{3.1}$$

```
1: T ← 0      {T stores the count of triangles}
2: for v ∈ V do
3:    for u ∈ 𝒩_v and v ≺ u do
4:       for w ∈ 𝒩_v and u ≺ w do
5:          if (u, w) ∈ E then
6:             T ← T + 1
```

Figure 3.1: Algorithm NodeIterator++, where $\prec$ is the degree based ordering of the nodes defined in Equation 3.1.

```
1: {Preprocessing: Step 2-6}
2: for each edge (u, v) do
3:    if u ≺ v, store v in N_u
4:    else store u in N_v
5: for v ∈ V do
6:    sort N_v in ascending order
7: T ← 0      {T is the count of triangles}
8: for v ∈ V do
9:    for u ∈ N_v do
10:      S ← N_v ∩ N_u
11:      T ← T + |S|
```

Figure 3.2: Algorithm NodeIteratorN, a modification of NodeIterator++.

Definition 1 (effective degree) *While $\mathcal{N}_v$ is the set of all neighbors of $v \in V$, let $N_v = \{u \in V | (u, v) \in E \wedge v \prec u\}$, i.e., N_v is the set of neighbors u of v such that $v \prec u$. We define $\hat{d}_v = |N_v|$ as the effective degree of v.*

This degree based ordering can improve the running time. Let $\hat{d}_v$ be the number of neighbors u of v such that $v \prec u$. We call $\hat{d}_v$ the *effective degree* of v. Assuming N_vs, for all v, are sorted and a binary search is used to check $(u, w) \in E$, a running time $O\left(\sum_v (\hat{d}_v d_v + \hat{d}_v^2 \log d_{\max})\right)$ can be shown, where $d_{\max} = \max_v d_v$. This running time is minimized when $\hat{d}_v$ values of the nodes are as close to each other as possible, although, for any ordering of the nodes, $\sum_v \hat{d}_v = m$ is invariant. Notice that in the degree-based ordering, diversity of the $\hat{d}_v$ values are reduced significantly.

We also observe that for the same reason, degree-based ordering of the nodes helps keep the loads among the processors balanced, to some extent, in a parallel algorithm. We use this degree-based ordering in our parallel algorithm and discuss this issue in detail in Section 3.3.

A simple modification of NodeIterator++ is as follows: perform comparison $u \prec v$ for each edge $(u, v) \in E$ in a preprocessing step rather than doing it while counting the triangles. This preprocessing step reduces the total number of $\prec$ comparisons to $O(m)$ from $\sum_v \hat{d}_v d_v$ and allows us to use an efficient set intersection operation. For each edge (v, u), u is stored in N_v if and only if

$v \prec u$. The modified algorithm NodeIteratorN is presented in Figure 3.2. All triangles containing node v and any $u \in N_v$ can be found by set intersection $N_u \cap N_v$ (Line 10 in Figure 3.2). The correctness of NodeIteratorN is proven in Theorem 1.

Theorem 1 *Algorithm NodeIteratorN counts each triangle in G only once.*

Proof: Consider a triangle (x_1, x_2, x_3) in G, and without the loss of generality, assume that $x_1 \prec x_2 \prec x_3$. By the constructions of N_x in the preprocessing step, we have $x_2, x_3 \in N_{x_1}$ and $x_3 \in N_{x_2}$. When the loops in Line 8-9 begin with $v = x_1$ and $u = x_2$, node x_3 appears in S (Line 10-11), and the triangle (x_1, x_2, x_3) is counted once. But this triangle cannot be counted for any other values of v and u (in Line 8-9) because $x_1 \notin N_{x_2}$ and $x_1, x_2 \notin N_{x_3}$. □

In NodeIteratorN, $|N_v| = \hat{d}_v$, the effective degree of v. When N_v and N_u are sorted, $N_u \cap N_v$ can be computed in $O(\hat{d}_u + \hat{d}_v)$ time. Then we have $O\left(\sum_{v \in V} d_v \hat{d}_v\right)$ time complexity for NodeIteratorN as shown in Theorem 2, in contrast to $O\left(\sum_v \left(\hat{d}_v d_v + \hat{d}_v^2 \log d_{\max}\right)\right)$ for NodeIterator++.

Theorem 2 *The time complexity of algorithm NodeIteratorN is $O\left(\sum_{v \in V} d_v \hat{d}_v\right)$.*

Proof: Time for the construction of N_v for all v is $O\left(\sum_v d_v\right) = O(m)$, and sorting these N_v requires $O\left(\sum_v \hat{d}_v \log \hat{d}_v\right)$ time. Now, computing intersection $N_v \cap N_u$ takes $O(\hat{d}_u + \hat{d}_v)$ time. Thus, the time complexity of NodeIteratorN is

$$
\begin{aligned}
& O(m) + O\left(\sum_{v \in V} \hat{d}_v \log \hat{d}_v\right) + O\left(\sum_{v \in V} \sum_{u \in N_v} (\hat{d}_u + \hat{d}_v)\right) \\
= \;& O\left(\sum_{v \in V} \hat{d}_v \log \hat{d}_v\right) + O\left(\sum_{(v,u) \in E} (\hat{d}_u + \hat{d}_v)\right) \\
= \;& O\left(\sum_{v \in V} \hat{d}_v \log \hat{d}_v\right) + O\left(\sum_{v \in V} d_v \hat{d}_v\right) = O\left(\sum_{v \in V} d_v \hat{d}_v\right).
\end{aligned}
$$

The second last step follows from the fact that for each $v \in V$, term $\hat{d}_v$ appears d_v times in this expression. □

Notice that set intersection operation can also be used with NodeIterator++ by replacing Line 4-6 of NodeIterator++ in Figure 3.1 with the following three lines as shown in [22] (Page 674):

```
1: S ← 𝒩_v ∩ 𝒩_u
2: for w ∈ S and u ≺ w do
3:     T ← T + 1
```

Table 3.1: Running time for the two sequential algorithms for counting triangles, NodeIterator++ and NodeIteratorN.

Networks	Runtime (sec.)		Triangles
	NodeIterator++	NodeIteratorN	
Email-Enron	0.14	0.07	0.7M
web-BerkStan	3.5	1.4	64.7M
LiveJournal	106	42	285.7M
Miami	46.35	32.3	332M
PA(25M, 50)	690	360	1.3M
Gnp(500K, 20)	1.81	0.6	1308

However, with this set intersection operation, the runtime of NodeIterator++ is $O\left(\sum_v d_v^2\right)$ since $|\mathcal{N}_v| = d_v$ in NodeIterator++, and computing $\mathcal{N}_v \cap \mathcal{N}_u$ takes $O(d_u + d_v)$ time. Further, the memory requirement for NodeIteratorN is half of that for NodeIterator++. NodeIteratorN stores $\sum_v \hat{d}_v = m$ elements in all N_v and NodeIterator++ stores $\sum_v d_v = 2m$ elements. Here we would like to note that the two algorithms presented in [42, 66] take the same asymptotic time complexity as NodeIteratorN. However, the algorithm in [66] requires three times more memory than NodeIteratorN. The algorithm in [42] requires more than twice the memory as NodeIteratorN, maintains a list of indices for all nodes, and the hidden constant in the runtime can be much larger.

We also experimentally compare the performance of NodeIteratorN and NodeIterator++ using both real-world and artificial networks. NodeIteratorN is significantly faster than NodeIterator++ for these networks as shown in Table 3.1.

3.2.1 Optimal Node Ordering

A total ordering $\prec$ of the nodes helps avoid duplicate counts of the same triangle. Any ordering of the nodes, e.g., ordering based on node IDs, random ordering, k-coreness based ordering, make sure each triangle is counted exactly once. By avoiding duplicate counts, these orderings also improve running time of the algorithm. However, different orderings lead to different runtimes. Figure 3.3 shows the runtime of our sequential algorithm for triangle counting with four orderings of nodes: ordering based on node IDs, degree, k-coreness, and random ordering. Node IDs and degrees are readily available with network data and do not require any additional computation. On the other hand, k-coreness based ordering requires computing coreness of nodes, and for random ordering, we generate n random numbers. Figure 3.3 (left) shows the comparison of runtime of counting triangles without considering the cost for computing orderings. Figure 3.3 (right) shows the comparison with total runtime of counting triangles and computing orderings. In both cases, degree based ordering provides the best runtime efficiency among all orderings. For networks with relatively even degree distribution such as Miami, all the orderings provide similar runtimes. However, for networks with skewed degree distribution, degree based ordering provides the least runtime. In our datasets, nodes with large degrees somehow appear at the beginning (having smaller IDs) giving ID based ordering almost the opposite effect of degree based ordering. As a result, ID based ordering provides the largest runtime for our datasets.

Now that our experimental results show degree based ordering provides the best runtime efficiency,

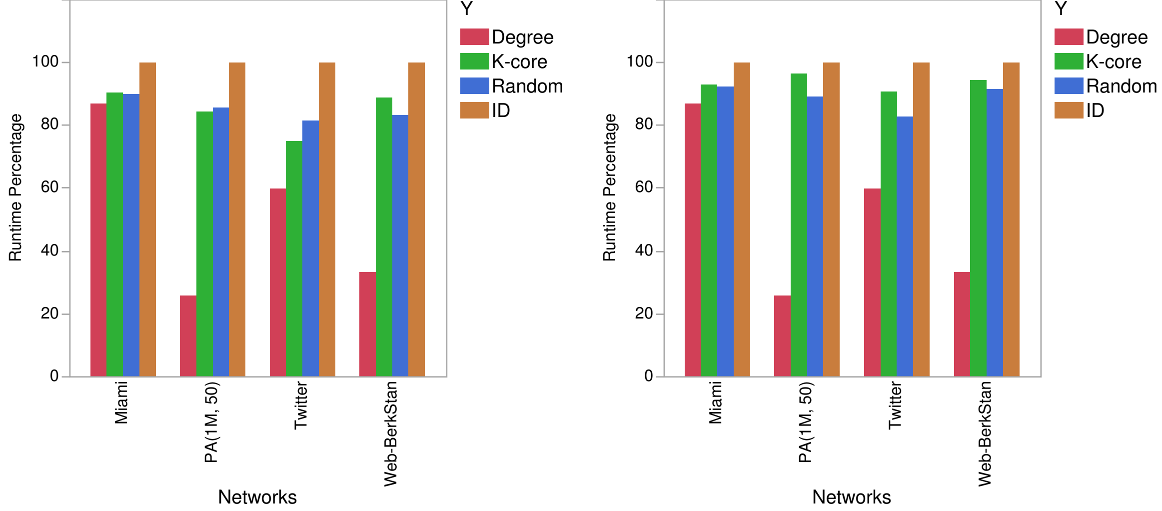


Figure 3.3: Comparison of runtime of sequential triangle counting (NodeIteratorN) with four distinct orderings of nodes. For each network, we compute the percentage of runtime with respect to the maximum runtime given by any of these orderings. In all cases, the degree based ordering gives the least runtime. Note that we compute the average runtime from 25 independent runs for the random ordering.

next we show in Theorem 4 that the degree based ordering is, in fact, the optimal ordering that minimizes the runtime of algorithm *NodeIteratorN*.

We denote the degree based ordering as $\prec_{\mathcal{D}}$ which is defined as follows:

$$u \prec_{\mathcal{D}} v \iff d_u < d_v \text{ or } (d_u = d_v \text{ and } u < v). \tag{3.2}$$

Assume there is another total ordering $\prec_{\mathcal{K}}$ based on some quantity k_v of nodes v:

$$u \prec_{\mathcal{K}} v \iff k_u < k_v \text{ or } (k_u = k_v \text{ and } u < v). \tag{3.3}$$

We now define a function that quantifies how ordering $\prec_{\mathcal{K}}$ agrees with $\prec_{\mathcal{D}}$ on the relative order of $x, y \in V$.

Definition 2 (Agreement function Y) *The agreement function $Y : V \times V \rightarrow \mathbb{Z}$ is defined as follows:*

$$Y(x,y) = \begin{cases} -1, & \textit{if } (x,y) \in E \textit{ and } x \prec_{\mathcal{D}} y \textit{ and } y \prec_{\mathcal{K}} x \\ 1, & \textit{if } (x,y) \in E \textit{ and } y \prec_{\mathcal{D}} x \textit{ and } x \prec_{\mathcal{K}} y \\ 0, & \textit{Otherwise} \end{cases}$$

It is, then, easy to see that $Y(x,y) = -Y(y,x)$.

We now prove an important result in the following lemma, which we subsequently use in Theorem

4.

Lemma 3 *For any* $(x, y) \in E$, $Y(x, y)(d_x - d_y) \geq 0$.

Proof: Let $c_{xy} = Y(x, y)(d_x - d_y)$. If orderings $\prec_{\mathcal{K}}$ and $\prec_{\mathcal{D}}$ agree on the relative order of x and y, then $Y(x, y) = 0$ by definition, and hence, $c_{xy} = 0$. Otherwise, consider the following three cases.

- $d_x = d_y$: This gives $d_x - d_y = 0$, and thus, $c_{xy} = 0$.
- $d_x < d_y$: We have $x \prec_{\mathcal{D}} y$ and $y \prec_{\mathcal{K}} x$, and thus, $Y(x, y) = -1$. Since $d_x - d_y < 0$, $c_{xy} > 0$.
- $d_x > d_y$: We have $y \prec_{\mathcal{D}} x$ and $x \prec_{\mathcal{K}} y$, and thus, $Y(x, y) = 1$. Since $d_x - d_y > 0$, $c_{xy} > 0$.

Therefore, for any $(x, y) \in E$, $c_{xy} = Y(x, y)(d_x - d_y) \geq 0$. □

Theorem 4 *Degree based ordering* $\prec_{\mathcal{D}}$ *minimizes the runtime for counting triangles using algorithm NodeIteratorN.*

Proof: Let $\hat{d}_v$ be the effective degree of vertex v with ordering $\prec_{\mathcal{D}}$. Then, the corresponding runtime for counting triangles is $\Theta\left(\sum_{i \in V} d_i \hat{d}_i\right)$. We provide a proof by contradiction. Assume that $\prec_{\mathcal{D}}$ is not an optimal ordering. Then there exists another ordering $\prec_{\mathcal{K}}$ that leads to a lower runtime for counting triangles than that of $\prec_{\mathcal{D}}$. Let $\prec_{\mathcal{K}}$ yields an effective degree $\tilde{d}$, the corresponding runtime for counting triangles is $\Theta\left(\sum_{i \in V} d_i \tilde{d}_i\right)$. Let $C_{\mathcal{D}} = \sum_{i \in V} d_i \hat{d}_i$ and $C_{\mathcal{K}} = \sum_{i \in V} d_i \tilde{d}_i$. Then, we have $C_{\mathcal{K}} < C_{\mathcal{D}}$.

Now, using Definition 2, the effective degree $\tilde{d}_x$ of node x obtained by $\prec_{\mathcal{K}}$ can be expressed as,

$$\tilde{d}_x = \hat{d}_x + \sum_{y \in \mathcal{N}_x} Y(x, y).$$

Now, we have,

$$\begin{aligned}
C_{\mathcal{K}} &= \sum_{x \in V} d_x \tilde{d}_x \\
&= \sum_{x \in V} d_x \left(\hat{d}_x + \sum_{y \in \mathcal{N}_x} Y(x, y) \right) \\
&= \sum_{x \in V} d_x \hat{d}_x + \sum_{x \in V} \left(d_x \sum_{y \in \mathcal{N}_x} Y(x, y) \right) \\
&= \sum_{x \in V} d_x \hat{d}_x + \sum_{(x,y) \in E} \left(d_x Y(x, y) + d_y Y(y, x) \right) \\
&= \sum_{x \in V} d_x \hat{d}_x + \sum_{(x,y) \in E} Y(x, y) \left(d_x - d_y \right).
\end{aligned}$$

The second last step follows from rearranging terms of the second summation and distributing them over edges. The last step follows from the fact that $Y(y, x) = -Y(x, y)$. Now, from Lemma 3 we have, $Y(x, y)(d_x - d_y) \geq 0$ for any $(x, y) \in E$. Thus, $\sum_{(x,y)\in E} Y(x, y)\,(d_x - d_y) \geq 0$, and therefore,

$$C_{\mathcal{K}} \geq \sum_{x\in V} d_x \hat{d}_x = C_{\mathcal{D}}.$$

This contradicts our assumption of $C_{\mathcal{K}} < C_{\mathcal{D}}$. Therefore, degree based ordering $\prec_{\mathcal{D}}$ is an optimal ordering which minimizes the runtime for counting triangles of our algorithm. □

We now prove some additional results based on the theorem we have just proven.

Corollary 5 *The following two statements are equivalent.*

1. *$\mathcal{K}$ is an optimal ordering.*
2. *$\mathcal{K}$ follows $\mathcal{D}$ for the relative order of any pair of nodes x and y where $(x, y) \in E$ and $d_x \neq d_y$.*

Proof: At first, we assume that (2) is true. We need to show that $\mathcal{K}$ is an optimal ordering. Following the same derivation of Theorem 4,

$$C_{\mathcal{K}} = C_{\mathcal{D}} + \sum_{(x,y)\in E} Y(x, y)\,(d_x - d_y)$$

Since $\mathcal{K}$ follows $\mathcal{D}$ for the relative order of pair of nodes x and y with $(x, y) \in E$ and $d_x \neq d_y$, by the definition, $Y(x, y) = 0$. Further, for $d_x = d_y$, $d_x - d_y = 0$. This gives $\sum_{(x,y)\in E} Y(x, y)\,(d_x - d_y) = 0$, since each term of the summation is 0. Hence, $C_{\mathcal{K}} = C_{\mathcal{D}}$. Since $\mathcal{D}$ is an optimal ordering (by Theorem 4), so is $\mathcal{K}$.

Second, we need to show if (1) is true, then (2) is also true. We will prove this by contraposition, that is, assuming (2) is not true, we will show that (1) is not true. Again, following the same derivation of Theorem 4,

$$C_{\mathcal{K}} = C_{\mathcal{D}} + \sum_{(x,y)\in E} Y(x, y)\,(d_x - d_y)$$

We assume $\mathcal{K}$ doesn't follow $\mathcal{D}$ for the relative order of some pair of nodes x and y with $(x, y) \in E$ and $d_x \neq d_y$. Then, by applying the same logic of cases $2b$ and $2c$ of Lemma 3, $Y(x, y)(d_x - d_y) > 0$. Since all other terms of the summation are ≥ 0 by the same lemma, we have,

$$\sum_{(x,y)\in E} Y(x, y)\,(d_x - d_y) > 0.$$

Hence, $C_{\mathcal{K}} > C_{\mathcal{D}}$. Since $\mathcal{D}$ is an optimal ordering (by Theorem 4), $\mathcal{K}$ is not. This proves the contraposition.

□

Corollary 5 offers us a useful hint to search for other orderings that incur the same triangle counting cost as of $\mathcal{D}$ and are, therefore, optimal.

We use algorithm *NodeIteratorN* with degree based ordering in our parallel algorithms for counting triangles.

3.3 The Parallel Algorithm

In this section, we present our parallel algorithm PATRIC for counting triangles in massive networks with overlapping partitioning and novel parallel load balancing schemes.

3.3.1 Overview of the Algorithm

We assume that the network does not fit in the local memory of a single computing node. Only a part of the entire graph is available to a processor. Let p be the number of processors used in the computation. The network is partitioned into p subgraphs, and each processor P_i is assigned one such subgraph $G_i(V_i, E_i)$ (formally defined below). P_i performs computation on its subgraph G_i. The main steps of our fast parallel algorithm are given in Figure 3.4. In the following subsections, we describe the details of these steps and several load balancing schemes.

```
1: Each processor P_i, in parallel, executes the following:(lines 2-4)
2: G_i(V_i, E_i) ← COMPUTEPARTITION(G, i)
3: T_i ← COUNTTRIANGLES(G_i, i)
4: BARRIER
5: Find T = ∑_i T_i
6: return T
```

Figure 3.4: The main steps of our fast parallel algorithm.

3.3.2 Partitioning the Network

The memory restriction poses a difficulty where the graph must be partitioned in such a way that the memory required to store a subgraph is minimized and at the same time each processor contains sufficient information to minimize communications among processors. For the input graph $G(V, E)$, processor P_i works on $G_i(V_i, E_i)$, which is a subgraph of G induced by V_i. The subgraph G_i is constructed as follows: First, set of nodes V is partitioned into p disjoint subsets

$V_0^c, V_1^c, \ldots, V_{p-1}^c$, such that, for any j and k, $V_j^c \cap V_k^c = \emptyset$ and $\bigcup_k V_k^c = V$. Second, set V_i is constructed containing all nodes in V_i^c and $\bigcup_{v \in V_i^c} N_v$. Edge set $E_i \subset E$ is the set of edges $\{(u, v) : u \in V_i \text{ and } v \in N_u\}$.

Each processor P_i is responsible for counting triangles incident on the nodes in V_i^c. We call any node $v \in V_i^c$ a *core* node of subgraph G_i. Each $v \in V$ is a core node in exactly one subgraph. How the nodes in V are distributed among the core sets V_i^c for all P_i ffect the load balancing and hence performance of the algorithm crucially. Later in Section 3.3.4, we present several load balancing schemes and the details of how sets V_i^c are constructed.

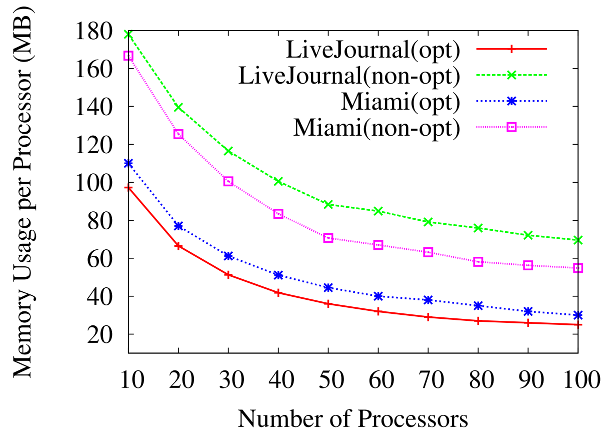


Figure 3.5: Memory usage with optimized and non-optimized data storing.

Now, P_i stores the set of neighbors N_v of all $v \in V_i$. Notice that for a node $w \in (V_i - V_i^c)$, N_w may contain some nodes $x \notin V_i$. Such nodes x can be safely removed from N_w and the number of triangles incident on all $v \in V_i^c$ can still be computed correctly. But, the presence of these nodes in N_w does not affect the correctness of the algorithm either. However, as our experimental results in Figure 3.5 show, we can save about 50% of memory space by not storing such nodes $x \notin V_i$ in N_w. Figure 3.5 also demonstrates the memory-scalability of our algorithm: as the more processors are used, each processor consumes less memory space.

3.3.3 Counting Triangles

Once each processor P_i has its subgraph $G_i(V_i, E_i)$, it uses the modified sequential algorithm *NodeIteratorN* presented in Section 3.2 to count triangles in G_i for each *core* node $v \in V_i^c$. Neighbor sets N_w for the nodes $w \in V_i - V_i^c$ help only in finding the edges among the neighbors of the core nodes. To be able to use an efficient intersection operation, N_v for all $v \in V_i$ are sorted. The code executed by processor P_i is given in Figure 3.6.

Once all processors complete their counting steps, the counts from all processors are aggregated into a single count by an MPI reduce function, which takes $O(\log p)$ time. Ordering of the nodes, construction of N_v, and disjoint partitioning of V into V_i^c make sure that each triangle in the

network appears exactly in one subgraph G_i. Thus, the correctness of the sequential algorithm *NodeIteratorN* shown in Section 3.2 ensures that each triangle is counted exactly once.

```
1: for v ∈ V_i do
2:     sort N_v in ascending order
3: T ← 0
4: for v ∈ V_i^c do
5:     for u ∈ N_v do
6:         S ← N_v ∩ N_u
7:         T ← T + |S|
8: return T
```

Figure 3.6: Algorithm executed by processor P_i to count triangles in $G_i(V_i, E_i)$.

3.3.4 Load Balancing in PATRIC

A parallel algorithm is completed when all of the processors complete their tasks. Thus, to reduce the running time of a parallel algorithm, it is desirable that no processor remains idle and all processors complete their executions almost at the same time. Furthermore, to deal with a massive network, it is also desirable that all subgraphs $G_i(V_i, E_i)$ require almost the same amount of memory space.

In Section 3.2, we discussed how degree based ordering of the nodes can reduce the running time of the sequential algorithm, and hence it reduces the running time of the local computation in each processor P_i. We observe that, interestingly, this degree-based ordering also provides load balancing to some extent, both in terms of running time and space, at no additional cost. Consider the example network shown in Figure 3.7. With an arbitrary ordering of the nodes, $|N_{v_0}|$ can be as much as $n-1$, and a single processor that contains v_0 as a core node is responsible for counting all triangles incident on v_0. Then the running time of the parallel algorithm can essentially be the same as that of a sequential algorithm. With the degree-based ordering, we have $|N_{v_0}| = 0$ and $|N_{v_i}| \leq 3$ for all i. Now if the core nodes are equally distributed among the processors, both space and computation time are almost balanced.

Although degree-based ordering helps mitigate the effect of skewness in degree distribution and balance load to some extent, working with more complex networks and highly skewed degree distribution reveals that distributing core nodes equally among processors does not make the load well-balanced in many cases. Figure 3.8 shows speedup of the parallel algorithm with an equal number of core nodes assigned to each processor. The speedup factor due to a parallelization is defined as t_s/t_p, where t_s and t_p are computation time required by a sequential and the parallel algorithm, respectively. As shown in Figure 3.8, LiveJournal networks show poor speedup, whereas the Miami network shows a relatively better speedup. This poor speedup for LiveJournal network is a consequence of a highly unbalanced computation load across the processors as shown in Figure 3.9. Although most of the processors complete their tasks in less than a second, a few

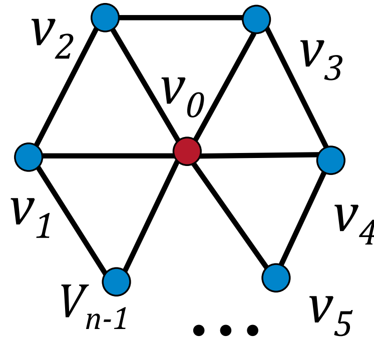


Figure 3.7: A network with a skewed degree distribution: $d_{v_0} = n - 1$, $d_{v_{i \neq 0}} = 3$.

of them take an unusually longer time leading to poor speedup. Unlike the Miami network, the LiveJournal network has a very skewed degree distribution. (Note that we used 100 processors for our experiments on load distribution. Although we could use a higher number of processors, using fewer processors helped demonstrate the pattern of imbalance of loads more clearly. In our subsequent experiments on scalability, we use a higher number of processors. In fact, we show that our algorithm scales to a larger number of processors when networks grow larger.) Next, we present several load balancing schemes that improve the performance of our algorithm significantly.

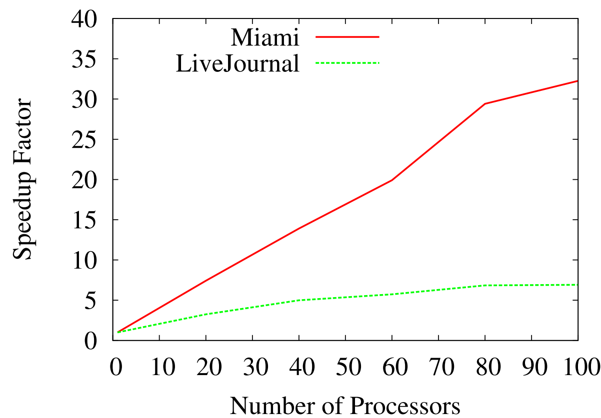


Figure 3.8: Speedup with equal number of core nodes in all processors.

Proposed Load Balancing Schemes

The balanced loads are determined before counting triangles. Thus, our parallel algorithm works in two phases:

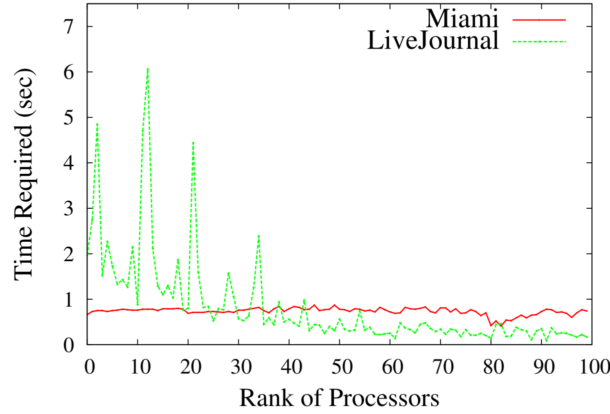


Figure 3.9: Computing load of individual processors (equal number of core nodes).

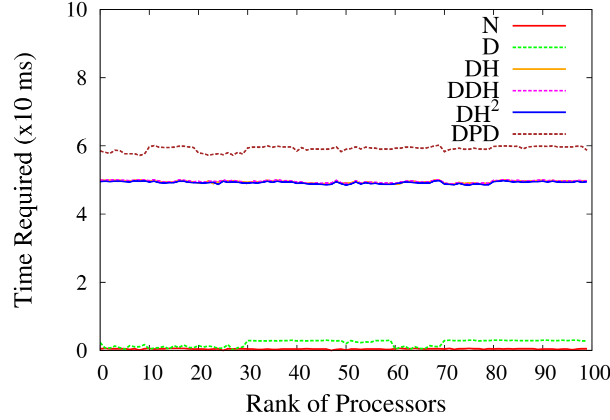


Figure 3.10: Load balancing cost for LiveJournal network with different schemes.

1. *Computing balanced load:* This phase computes V_i^c so that the computational loads are well-balanced.

2. *Counting triangles:* This phase counts the triangles following the algorithms in Figure 3.4 and 3.6.

Computational cost for phase 1 is referred to as *load-balancing cost*, for phase 2 as *counting cost*, and the total cost for these two phases as *total computational cost*. In order to be able to distribute load evenly among the processors, we need an estimation of computation load for computing triangles. For this purpose, we define a *cost function* $f : V \to \mathbb{R}$, such that $f(v)$ is the computational cost for counting triangle incident on node v (Lines 4-7 in Figure 3.6). Then, the total cost incurred to processor P_i is given by $\sum_{v \in V_i^c} f(v)$. To achieve a good load balancing, $\sum_{v \in V_i^c} f(v)$ should be almost equal for all i. Thus, the computation of balanced load consists of the following two steps:

1. **Computing f:** Compute $f(v)$ for each $v \in V$
2. **Computing partitions:** Determine p disjoint subsets $V_i^c \subset V$ such that

$$\sum_{v \in V_i^c} f(v) \approx \frac{1}{p} \sum_{v \in V} f(v) \tag{3.4}$$

The above computation must also be done in parallel. Otherwise, this computation takes at least $\Omega(n)$ time, which can wipe out the benefit gained from balancing load completely or even have a negative effect on the performance. Parallelizing the above computation, especially Step 2 (computing partitions), is a non-trivial problem. Next, we describe a parallel algorithm to perform the above computation.

Computing f:
It might not be possible to exactly compute the value of $f(v)$ before the actual execution of counting triangles takes place. Fortunately, Theorem 2 provides a mathematical formulation of counting cost in terms of the number of vertices, edges, original degree d, and effective degree $\hat{d}$. Guided by Theorem 2, we have come up with several approximate cost functions $f(v)$ that are listed in Table 3.2. Each function corresponds to one load balancing scheme. The rightmost column of the table shows identifying notations of the individual schemes.

Table 3.2: Cost functions $f(.)$ for our load balancing schemes.

Node Function	Identifying Notation
$f(v) = 1$	$\mathbb{N}$
$f(v) = d_v$	$\mathbb{D}$
$f(v) = \hat{d}_v$	$\mathbb{DH}$
$f(v) = d_v \hat{d}_v$	$\mathbb{DDH}$
$f(v) = \hat{d}_v^{\,2}$	$\mathbb{DH}^2$
$f(v) = \sum_{u \in N_v} (\hat{d}_v + \hat{d}_u)$	$\mathbb{DPD}$

The input graph is given as a sequence of adjacency lists: adjacency list of the first node followed by that of the second node, and so on. The input sequence is considered divided by size (number of bytes) into p chunks. However, it is made sure that adjacency list of a particular node reside in only one processor. Initially, processor P_i stores the ith chunk in its memory. Let C_i be the set of all nodes in the i-th chunk. Next, P_i computes $f(v)$ for all nodes $v \in C_i$ as follows.

- **Scheme** $\mathbb{N}$: Function $f(v) = 1$ requires no computation. This scheme, essentially, assigns an equal number of core nodes to each processor.
- **Scheme** $\mathbb{D}$: Function $f(v) = d_v$ requires no computation. This scheme, essentially, assigns an equal number of edges to each processor.
- **Scheme** $\mathbb{DH}$: Computing function $f(v) = \hat{d}_v$ requires degrees of all $u \in \mathcal{N}_v$. Let $u \in C_j$. Then, P_i sends a request message to P_j, and P_j replies with a message containing d_u.

- **Scheme** $\mathbb{DDH}$: For $f(v) = d_v \hat{d}_v$, $\hat{d}_v$ is computed as above.
- **Scheme** $\mathbb{DH}^2$: For $f(v) = \hat{d}_v^{\,2}$, $\hat{d}_v$ is computed as above.
- **Scheme** $\mathbb{DPD}$: Function $f(v) = \sum_{u \in \mathcal{N}_v} (\hat{d}_v + \hat{d}_u)$ is computed as follows.

   i. Each P_i computes $\hat{d}_v$, $v \in C_i$, as discussed above.

   ii. Then P_i finds $\hat{d}_u$ for all $u \in \mathcal{N}_v$: Let $u \in C_j$. P_i sends a request message to P_j, and P_j replies with a message containing $\hat{d}_u$.

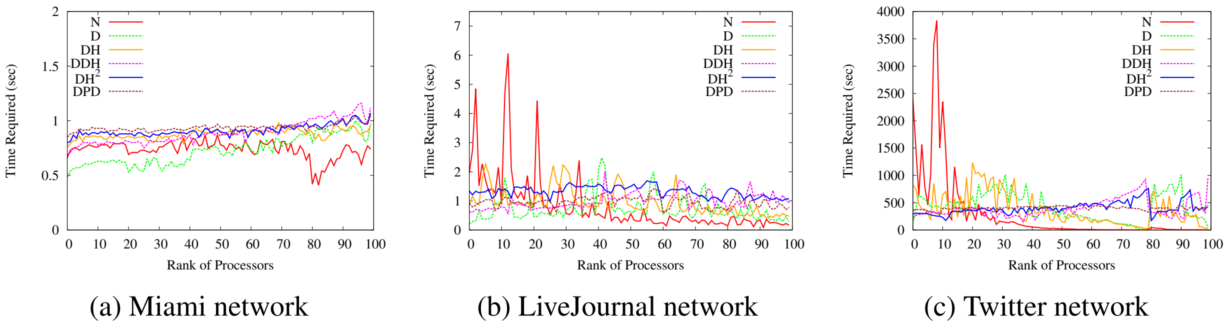
   iii. Now, $f(v) = \sum_{u \in \mathcal{N}_v} (\hat{d}_v + \hat{d}_u)$ is computed using $\hat{d}_v$ and $\hat{d}_u$ obtained in (i) and (ii).

Computing partitions:
Given that each processor P_i knows $f(v)$ for all $v \in C_i$, our goal is to partition V into p disjoint subsets V_i^c such that $\sum\limits_{v \in V_i^c} f(v) \approx \frac{1}{p} \sum\limits_{v \in V} f(v)$.

We first compute cumulative sum $F(t) = \sum\limits_{v=0}^{t} f(v)$ in parallel by using a parallel prefix sum algorithm [5]. Processor P_i computes and stores $F(t)$ for nodes $t \in C_i$. This computation takes $O\left(\frac{n}{p} + \log p\right)$ time. Notice that P_{p-1} computes $F(n-1) = \sum\limits_{v=0}^{n-1} f(v)$, cost for counting all triangles in the graph. P_{p-1} then computes $\alpha = \frac{1}{P} \sum\limits_{v \in V} f(v) = \frac{1}{p} F(n-1)$ and broadcast α to all other processors. Now, let $V_i^c = \{x_i, x_i + 1 \ldots, x_{(i+1)} - 1\}$ for some node $x_i \in V$. We call x_i the *start* or *boundary* node of partition i. Node x_j is the jth boundary node if and only if $F(x_j - 1) < j\alpha \leq F(x_j)$ or equivalently, $x_j = argmin_{v \in V} (F(v) \geq j\alpha)$. A chunk C_i may contain $0, 1$, or multiple boundary nodes in it. Each P_i finds the boundary nodes x_j in its chunk: we use the algorithm presented in [3] to compute boundary nodes of partitions, which takes $O(n/p + p)$ time in the worst case. At the end of this execution, each processor P_i knows boundary nodes x_i and $x_{(i+1)}$. Now P_i can construct V_i^c and compute its subgraph $G_i(V_i, E_i)$ as described in Section 3.3.2.

Since scheme $\mathbb{DPD}$ requires two levels of communication for computing $f(.)$, it has the largest load balancing cost among all schemes. Computing $f(.)$ for $\mathbb{DPD}$ requires $O(\frac{m}{p} + p \log p)$ time. Computing partitions has a runtime complexity of $O(\frac{m}{p} + p)$. Therefore, the load balancing cost of $\mathbb{DPD}$ is given by $O(\frac{m}{p} + p \log p)$. Figure 3.10 shows an experimental result of the load balancing cost for different schemes on the LiveJournal network. Scheme $\mathbb{N}$ has the lowest cost and $\mathbb{DPD}$ the highest. Schemes $\mathbb{DH}$, $\mathbb{DH}^2$, and $\mathbb{DDH}$ have a quite similar load balancing cost. However, since scheme $\mathbb{DPD}$ gives the best estimation of the counting cost, it provides better load balancing. Figure 3.11 demonstrates *total computation cost* (load) incurred in individual processors with different schemes on Miami, LiveJournal, and Twitter networks. Miami is a network with an almost even degree distribution. Thus, all load balancing schemes, even simpler schemes like $\mathbb{N}$ and $\mathbb{D}$, distribute loads almost equally among processors. However, LiveJournal and Twitter have a very skewed degree distribution. As a result, partitioning the network based on number of nodes ($\mathbb{N}$) or degree ($\mathbb{D}$) do not provide good load balancing. The other schemes capture the computational

load more precisely and produce a very even load distribution among processors. In fact, for such networks, scheme $\mathbb{DPD}$ provides the best load balancing.



(a) Miami network (b) LiveJournal network (c) Twitter network

Figure 3.11: Load distribution among processors for LiveJournal, Miami and Twitter networks by different schemes.

3.3.5 Performance Analysis

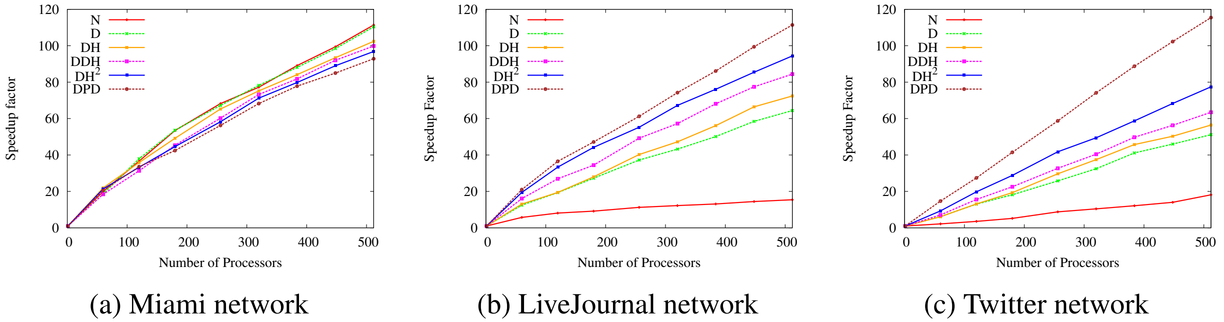
In this section, we present the experimental results evaluating the performance of our algorithm and the load balancing schemes.

Strong Scaling

Strong scaling of a parallel algorithm shows how much speedup a parallel algorithm gains as the number of processors increases. Figure 3.12 shows strong scaling of our algorithm on LiveJournal, Miami and Twitter networks with different load balancing schemes. The speedup factors of these schemes are almost equal on Miami network. Schemes $\mathbb{N}$ and $\mathbb{D}$ have a little better speedup than the others. On the contrary, for LiveJournal and Twitter networks, speedup factors for different load balancing schemes vary quite significantly. Scheme $\mathbb{DPD}$ achieves better speedup than other schemes. As discussed before, for Miami network, all load balancing schemes distribute loads equally among processors. This produces an almost same speedup on Miami network with all schemes. A lower load balancing cost of schemes $\mathbb{N}$ and $\mathbb{D}$ (Figure 3.10) yields a little higher speedup. However, for LiveJournal and Twitter networks, scheme $\mathbb{DPD}$ gives the best load distribution (Figure 3.11) and thus provides the best speedups. Although $\mathbb{DPD}$ has a higher load balancing cost than others, the benefit gained from $\mathbb{DPD}$ as an even load distribution outweighs this cost. Thus we recommend for using $\mathbb{DPD}$ on real-world big graphs. Our subsequent results will be based on scheme $\mathbb{DPD}$.

Weak Scaling

Weak scaling of a parallel algorithm shows the ability of the algorithm to maintain constant computation time when the problem size grows proportionally with the increasing number of pro-



(a) Miami network (b) LiveJournal network (c) Twitter network

Figure 3.12: Speedup gained from different load balancing schemes for LiveJournal, Miami and Twitter networks.

cessors. We use PA(n, m) networks for this experiment, and for x processors, we use network PA($x/10 \times 1M, 50$). The weak scaling of our algorithm is shown in Figure 3.13. Triangle counting cost remains almost constant (blue line). Since the load-balancing step has a communication overhead of $O(p \log p)$, load-balancing cost increases gradually with the increase of processors. It causes the total computation time to grow slowly with the addition of processors (red line). Since the growth is very slow and the runtime remains almost constant, the weak scaling of our algorithm is very good.

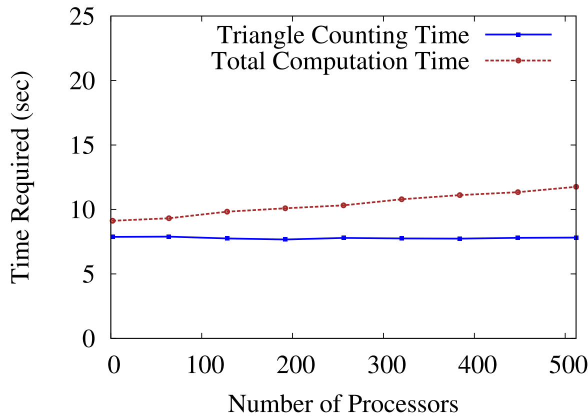


Figure 3.13: Weak scaling on PA($P/10 \times 1M, 50$) networks.

Comparison with Previous Algorithms

The runtime of our algorithm on several real and artificial networks are shown in Table 3.3. We also compare our algorithm with another distributed-memory parallel algorithm for counting triangles given in [72]. We select three of the five networks used in [72]. Twitter and LiveJournal are the two largest among the networks used in [72]. We also use web-BerkStan which has a very skewed

degree distribution. No artificial network is used in [72]. For all of these three networks, our algorithm is more than 45 times faster than the algorithm in [72]. The improvement over [72] is due to the fact that their algorithm generates a huge volume of intermediate data, which are all possible 2-paths centered at each node. The amount of such intermediate data can be significantly larger than the original network. For example, for the Twitter network, 300B 2-paths are generated while there are only 2.4B edges in the network. The algorithm in [72] shuffles and regroups these 2-paths, which take significantly larger time and also memory.

Table 3.3: Runtime performance of PATRIC using 200 processors and the algorithm in [72].

Networks	**Runtime (sec.)**		**Triangles**
	PATRIC	[72]	
Twitter	9.4m	423m	34.8B
web-BerkStan	0.10s	1.70m	65M
LiveJournal	0.8s	5.33m	286M
Miami	0.6s	–	332M
PA(1B, 20)	15.5m	–	0.403M

Scaling with Network Size

The load-balancing cost of our algorithm, as shown in Section 3.3.4, is $O(m/p + p\log p)$ where p is the number of processors used in the computation. For the algorithm given in Figure 3.6, the counting cost is $O(\sum_{v\in V_i^c}\sum_{u\in N_v}(\hat{d}_u+\hat{d}_v))$. Thus, the total computational cost of our algorithm is,

$$\begin{aligned} F(p) &= O(\frac{m}{p} + p\log p + \max_i \sum_{v\in V_i^c}\sum_{u\in N_v}(\hat{d}_u+\hat{d}_v)) \\ &\approx c_1\frac{m}{p} + c_2 p\log p + c_3\max_i \sum_{v\in V_i^c}\sum_{u\in N_v}(\hat{d}_u+\hat{d}_v), \end{aligned}$$

where c_1, c_2, and c_3 are constants. Now, the quantity denoting computation cost is given by,

$$c_1 m/p + c_3 \sum_{v\in V_i^c}\sum_{u\in N_v}(\hat{d}_u+\hat{d}_v), \tag{3.5}$$

which decreases with the increase of p, but communication cost $p\log p$ increases with p. Thus, initially when p increases, the overall runtime decreases (hence the speedup increases). But, for some large value of p, the term $p\log p$ becomes dominating, and the overall runtime increases with the addition of further processors. Notice that communication cost $p\log p$ is independent of network size. Therefore, when networks grow larger, computation cost increases, and hence they scale to a higher number of processors, as shown in Figure 3.14. This is, in fact, a highly desirable behavior of our parallel algorithm which is designed for real world massive networks. We need large number of processors when the network size is large and computation time is high.

Consequently, there is an optimal value of p, p_{opt}, for which the total time $F(p)$ drops to its minimum and the speedup reaches its maximum. To have an estimation of p_{opt}, we replace d and $\hat{d}$ with average degree $\bar{d}$ and $\bar{d}/2$, respectively, and have $F(p) \approx c_1 n\bar{d}/p + c_2 p \log p + c_3 n\bar{d}^2/p$. At the minimum point, $\frac{\mathrm{d}}{\mathrm{d}p}\big(F(p)\big) = 0$, which gives the following relationship of p_{opt}, n and $\bar{d}$: $p^2(1 + \log p) = \frac{n}{c_2}(c_3\bar{d}^2 + c_1\bar{d})$. Thus, p_{opt} has roughly a linear relationship with $\sqrt{n}$ and $\bar{d}$.

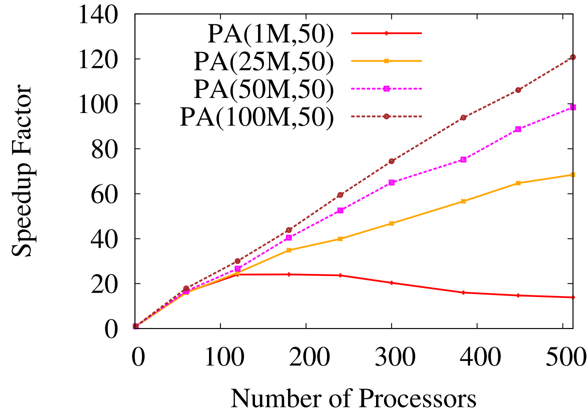


Figure 3.14: Improved scalability with increased network size.

Assume that a network with the number of nodes n' and average degree $\bar{d}'$ experimentally shows an optimal p of p'_{opt}. Then, another network with n nodes and an average degree $\bar{d}$ has an approximate optimum number of processors,

$$p_{opt} \approx p'_{opt} \frac{\bar{d}}{\bar{d}'} \sqrt{\frac{n}{n'}}. \tag{3.6}$$

Thus, if we compute p'_{opt} experimentally by trial and error for an available network (let's call it the *base network*), we can estimate p_{opt} for all other networks. The base network might be a small network for which this trial-error should be fairly fast. From the result presented in Figure 3.14, the network $PA(1M, 50)$ can serve as a base network, and p_{opt} for the network $PA(25M, 50)$ can be estimated as $p_{opt} \approx 600$ which is approximately 5 times of that of $PA(1M, 50)$ ($p'_{opt} \approx 120$). The relationship is also justified when we vary average degree of the networks.

3.4 A Sparsification-based Parallel Approximation Algorithm

In this section, we integrate a sparsification technique, called DOULION, proposed in [76] with our parallel algorithm. Our adapted version of DOULION provides more accuracy than DOULION. Sparsification of a network is a sampling technique where some randomly chosen edges are retained and the rest are deleted, and then computation is performed in the resulting network. Sparsification of a network saves both computation time and memory space and provides an approximate result.

Let $G(V, E)$ and $G'(V, E' \subset E)$ be the networks before and after sparsification, respectively. Network $G'(V, E')$ is obtained from $G(V, E)$ by retaining each edge, independently, with probability q and removing it with probability $1 - q$. Now any algorithm can be used to find the exact number of triangles in G'. Let $T(G')$ be the number of triangles in G'. The estimated number of triangles in G is given by $\frac{1}{q^3}T(G')$, which is an unbiased estimation since $E\left[\frac{1}{q^3}T(G')\right] = T(G)$.

As shown in [76], the variance of the estimated number of triangles is

$$\text{Var} = \left(\frac{1}{q^3} - 1\right) T(G) + 2k\left(\frac{1}{q} - 1\right), \tag{3.7}$$

where k is the number of pairs of triangles in G with an overlapping edge (see Figure 3.15).

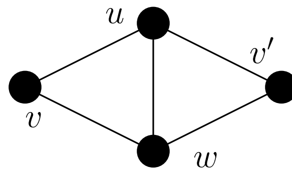


Figure 3.15: Two triangles (v, u, w) and (v', u, w) with an overlapping edge.

In our parallel algorithm, sparsification is done as follows: each processor P_i independently performs sparsification on its partition $G_i(V_i, E_i)$. While loading partition G_i into its local memory, it retains each edge $(u, v) \in E_i$ with probability q and discards it with probability $1 - q$ as shown Figure 3.16. If T' is the number of triangles obtained after sparsification, $\frac{1}{q^3}T'$ is the estimated number of triangles in G.

1: **for** $v \in V_i$ **do**
2:     **for** $(v, u) \in E$ **do**
3:         **if** $v \prec u$ **then**
4:             toss a biased coin with $success$ prob. q
5:             **if** $success$ **then**
6:                 store u to N_v
7: $T_i \leftarrow$ count of triangles
8: Find Sum $T' = \sum_i T_i$ using MPIREDUCE
9: $T \leftarrow \frac{1}{q^3} \times T'$

Figure 3.16: Counting the number of triangles in a network with our parallel sparsification method.

Notice that the sparsification of our algorithm is not exactly the same as that of DOULION. Consider two triangles (v, u, w) and (v', u, w) with an overlapping edge (u, w) as shown in Figure 3.15. In DOULION, if edge (u, w) is not retained, none of the two triangles survive, and as a result, survivals of (v, u, w) and (v', u, w) are not independent events. Now, in our case, if v and v' are core nodes in two different partitions G_i and G_j, processor P_i may retain edge (u, w) while

processor P_j discards (u, w), and vice versa. As P_i and P_j perform sparsification independently, survivals of triangles (v, u, w) and (v', u, w) are independent events.

However, our estimation is also unbiased, and in fact, this difference (with DOULION) improves the accuracy of the estimation by our parallel algorithm. Since the probability of survival of any triangle is still exactly $\frac{1}{q^3}$, we have $E\left[\frac{1}{q^3}T'\right] = T$. To calculate variance of the estimation, let k'_i be the number of pairs of triangles with an overlapping edge such that both triangles are in partition G_i, and $k' = \sum_i k'_i$. Let k'' be the number of pairs of triangles (v, u, w) and (v', u, w) with an overlapping edge (u, w) (as shown in Figure 3.15) and v and v' are core nodes in two different partitions. Then clearly, $k' + k'' = k$ and $k' \leq k$. Now following the same steps as in [76], one can show that the variance of our estimation is

$$\text{Var}' = \left(\frac{1}{q^3} - 1\right) T(G) + 2k' \left(\frac{1}{q} - 1\right). \tag{3.8}$$

Comparing Eqn. 3.7 and 3.8, if $k'' > 0$, we have $k' < k$ and reduced variance leading to improved accuracy. This observation is verified by experimental results on two real-world networks (Table 4.4). It also suggests that accuracy can be improved with a larger number of processors.

Table 3.4: Accuracy of our parallel sparsification algorithm and DOULION [76] with $q = 0.1$. Our parallel algorithm was run with 100 processors. Variance, max error and average error are calculated from 25 independent runs for each of the algorithms.

Networks	**Variance**		**Avg. error (%)**		**Max error (%)**	
	Our Alg.	DOULION	Our Alg.	DOULION	Our Alg.	DOULION
web-BerkStan	1.287	2.027	0.389	0.392	1.02	1.08
LiveJournal	1.770	1.958	1.46	1.86	3.88	4.75

Table 3.5: Comparison of our parallel sparsification algorithm and DOULION [76] on LiveJournal network with 100 processors.

Metrics	q	0.1	0.2	0.3	0.4	0.5
Accuracy	Our Alg.	99.9914	99.9917	99.9924	99.9936	99.9971
	DOULION	99.6310	99.7544	99.8392	99.9121	99.9584
Speedup	Our Alg.	57.88	24.36	11.04	6.19	4.0
	DOULION	30.96	11.96	6.71	4.31	3.03

In [76], it was shown that due to sparsification with parameter q, the computation can be faster as much as $1/q^2$ times. However, in practice the speed up is typically smaller than $1/q^2$ but larger than $1/q$. Table 4.5 shows the accuracy and speedup factor with varying q for the LiveJournal network. The speedup factor, due to sparsification, of our algorithm is better than that of DOULION. For the LiveJournal network, DOULION shows a speedup of 31 with $q = 0.1$, while our algorithm has a speedup of 58. Sparsification also reduces memory requirement since only a subset of the edges are

stored in the main memory. As a result, adaptation of sparsification allows our parallel algorithm to work with even larger networks. With sampling probability q (the probability of retaining an edge), the expected number of edges to be stored in the main memory is $q|E|$. Thus, we can expect that the use of sparsification with PATRIC will allow us to work with a network $1/q$ times larger, a network with few hundreds billion edges.

3.5 Conclusion

We presented a parallel algorithm, called PATRIC, for counting triangles in a massive network. This parallel algorithm can work with networks that have billions of nodes and edges. Such capability of PATRIC will enable various types of analysis of massive real-world networks, networks that otherwise do not fit in the main memory of a single processor. PATRIC shows very good scalability with both the number of processors and the problem size and performs well on both real-world and artificial networks. PATRIC has been able to count triangles of a massive network with $1B$ nodes and $10B$ edges in 16 minutes using 40 processors. We presented several load balancing schemes and showed that such schemes provide very good balancing. Further, we have adopted the sparsification approach of DOULION in our parallel algorithm with improved accuracy. This adoption will allow us to deal with even larger networks.

Chapter 4

A Space-efficient Parallel Algorithm for Counting the Exact Number of Triangles in Massive Networks

In this chapter, we present a space-efficient MPI based parallel algorithm for counting the *exact* number of triangles in massive networks. Although there exist several MapReduce and only one MPI (Message Passing Interface) based distributed-memory parallel algorithms for counting triangles, those have limitations regarding space efficiency. MapReduce based algorithms generate prohibitively large intermediate data. The MPI based algorithm can work on quite large networks, however, the overlapping partitioning employed by the algorithm limit its capability to deal with very massive networks. Our space-efficient algorithm partitions the network into non-overlapping subgraphs. Our results demonstrate up to 25-fold space saving over the algorithm with overlapping partitioning. This space efficiency allows the algorithm to deal with networks that are 25 times larger. We present a novel approach that reduces communication cost drastically (up to 90%) leading to both a space- and runtime-efficient algorithm. Our adaptation of a parallel partitioning scheme by computing a novel weight function adds further to the efficiency of the algorithm.

4.1 Introduction

The algorithm presented in Chapter 3 divides the input graph into a set of p overlapping partitions where some edges (u, v) might be repeated (overlapped) in multiple partitions. Such overlapping allows the algorithm to count triangles without any communication among processors leading to faster computation. Further, since each processor works on a part of the entire graph, the algorithm can work on large graphs. However, for instances where the graph has a high average degree or a few nodes with high degrees, overlapping partitions can be large. Now, if overlapping of edges among partitions are avoided, we can further improve the space efficiency of the algorithm. In this chapter, we present a parallel algorithm which divides the input graph into non-overlapping partitions. Each edge resides in a single partition, and the sizes of all partitions sum up to the size of the graph. Non-overlapping partitioning leads to a more space efficient algorithm and thus

allows to work on larger graphs. In fact, non-overlapping partitioning offers as much as $\bar{d}$ (average degree of the graph) times space saving over the overlapping partitions. Table 4.1 shows the space requirement of non-overlapping partitions which is up to 25 times smaller than that overlapping partitions for the networks we experimented on.

Table 4.1: Memory usage of our algorithms (size of the largest partition) with both overlapping and non-overlapping partitioning. Number of partitions used is 100.

Networks	Memory (MB)		Ratio	$\bar{d}$	d_{max}
	Non-overlap.	Overlap.			
web-Google	1.49	11.3	7.85	11.6	6332
LiveJournal	9.41	110.75	11.75	18	20333
Miami	10.63	109.58	10.32	47.6	425
Twitter	265.82	4254.18	16.004	57.1	1001159
PA(10M, 100)	121.11	2120.94	17.5	100	25068
PA(1M, 1000)	138.20	3427.36	24.8	1000	19255

Notice the space requirement of the other distributed-memory parallel algorithms for counting the exact number of triangles in literature: the first MapReduce based algorithm proposed in [72] generates a huge amount of intermediate data which is significantly larger than the original network (e.g., 125 times larger for Twitter network). The second MapReduce based algorithm proposed in [72], the partition-based algorithm, has a space requirement of $O(mp)$ for the Map phase (when the network is partitioned into p subgraphs), which is p times larger than the network size. The algorithm in [54] also requires $O(mp)$ memory space.

Our space-efficient parallel algorithm partitions the input networks into non-overlapping subgraphs. The load balancing procedure makes sure that the computational cost is almost equal for each processor. We also observe experimentally that the largest subgraph has approximately $\frac{m}{p}$ edges. This algorithm requires only a total of $O(m)$ space for storing all p subgraphs. This partitioning offers as much as $\bar{d}$ times saving over the overlapping partitioning and thus allows to work on larger networks.

Our Contributions. We present a space-efficient MPI-based parallel algorithm for counting the exact number of triangles in massive networks. The algorithm employs a non-overlapping partitioning leading to a significant space saving. We present a novel approach that reduces communication cost drastically without requiring additional space, which leads to both a space- and runtime-efficient algorithm. Our adaptation of a parallel partitioning scheme by computing a novel weight function offers additional runtime efficiency to the algorithm. Our algorithm achieves up to $O(p^2)$-factor space saving over existing MapReduce based algorithms and up to $\bar{d}$-factor (approx.) over the algorithm with overlapping partitioning.

Remarks. Note that unlike approximation algorithms that provide an overall (global) estimate of the number of triangles in the graph, this paper presents an exact algorithm that can be used to count triangles incident on individual nodes (local triangles). Such *local* counts facilitate computing clustering coefficient of nodes and finding vertex neighborhood and community seeds [33]. To the best of our knowledge, among all exact algorithms, our algorithm has the lowest space complexity, without even compromising its runtime efficiency.

Although there exist a couple of standard parallel graph partitioning algorithms such as Parmetis and Zoltan [82], those might not work well for our problem. Those algorithms strive to minimize cut edges, which help reduce communication overhead, however, we also require the computation cost to be well-balanced among processors. We need to estimate weights of nodes (based on triangle counting cost) in parallel in the partitioning procedure. This parallel computation of weights is not readily available in standard algorithms. Hence we adapt the parallel partitioning scheme presented in Chapter 3, which considers the actual triangle counting cost incurred at nodes and thus helps in balancing computation loads.

We present our space-efficient parallel algorithm with non-overlapping partitioning in the following section.

4.2 A space-efficient Parallel Algorithm for Counting Triangles

First, we present an overview of the algorithm. A detailed description follows thereafter.

4.2.1 Overview of the Algorithm.

This algorithm partitions the input graph $G(V, E)$ into a set of p partitions constructed as follows: set of nodes V is partitioned into p disjoint subsets V_i^c, such that, for $0 \leq j, k \leq p-1$ and $j \neq k$, $V_j^c \cap V_k^c = \emptyset$ and $\bigcup_k V_k^c = V$. Edge set E_i^c, constructed as $E_i^c = \{(u, v) : u \in V_i^c, v \in N_u\}$, constitutes the i-th partition. Note that this partition is non-overlapping– each edge $(u, v) \in E$ resides in one and only one partition. For $0 \leq j, k \leq p-1$ and $j \neq k$, $E_j^c \cap E_k^c = \emptyset$ and $\bigcup_k E_k^c = E$. The sum of space required to store all partitions equals to the space required to store the whole graph.

Now, to count triangles incident on $v \in V_i^c$, processor P_i needs N_u for all $u \in N_v$ (Lines 7-10, Figure 3.2). If $u \in V_i^c$, information of both N_v and N_u is available in the i-th partition, and P_i counts triangles incident on (v, u) by computing $N_u \cap N_v$. However, if $u \in V_j^c$, $j \neq i$, N_u resides in partition j. Processor P_i and P_j exchange message(s) to count triangles incident on such (v, u). This exchanging of messages introduces a communication overhead, which is a crucial factor on the performance of the algorithm. We devise an efficient approach to reduce the communication overhead drastically and improve the performance significantly. Once all processors complete the computation associated with respective partitions, the counts from all processors are aggregated.

4.2.2 An Efficient Communication Approach

Processors P_i and P_j require to exchange messages for counting triangles incident on (v, u) where $v \in V_i^c$ and $u \in N_v \cap V_j^c$. A simple way to count such triangles is as follows: P_i requests P_j for N_u. P_j sends N_u to P_i, and P_i counts triangles incident on the edge (v, u) by computing $N_v \cap N_u$. For further reference, we call this approach the *direct approach*. This approach requires exchanging

as much as $O(m\bar{d})$ messages ($\bar{d}$ is the average degree of the network) which is substantially larger than the size of the graph.

The above approach has a high communication overhead due to exchanging a large number of redundant messages leading to a large runtime. Assume $u \in N_{v_1} \cap N_{v_2} \cap \cdots \cap N_{v_k}$, for $v_1, v_2, \ldots, v_k \in V_i^c$. Then P_i sends k separate requests for N_u to P_j while computing triangles incident on v_1, v_1, $\ldots$, v_k. In response to those requests, P_j sends N_u to P_i for a total of k times.

One seemingly obvious way to eliminate redundant messages is that instead of requesting N_u multiple times, P_i stores it in memory for subsequent use. However, space requirement for storing all N_u along with the partition i itself is the same as that of storing an overlapping partition. This diminishes our original goal of a space-efficient algorithm.

Another way of eliminating message redundancy is as follows. When N_u is fetched, P_i completes all computations that require N_u: P_i finds all k nodes $v \in V_i^c$ such that $u \in N_v$. It then performs all k computations $N_v \cap N_u$ involving N_u and discards N_u. Now, since $u \in N_v \implies v \notin N_u$, P_i cannot extract all such nodes v from the message N_u. Instead, P_i requires to scan through its whole partition to find such nodes v where $u \in N_v$. This *scanning* is very expensive, namely, $O(\sum_{v \in V_i^c} d_v)$ in the worst case for each message, which might even be slower than the direct approach with redundant messages.

All the above techniques to improve the efficiency of *Direct* approach introduce additional space or runtime overhead. Below we propose an efficient approach to reduce message exchanges drastically without adding further overhead.

Reduction of messages. To compute $N_v \cap N_u$ for $v \in V_i^c$ and $u \in N_v \cap V_j^c$, P_i requires fetching N_u from partition j. Instead, P_j can perform the same computation if P_i sends N_v to P_j. Specifically, we consider the following approach: P_i sends N_v to P_j instead of fetching N_u. P_j counts triangles incident on edge (u, v) by performing the operation $N_v \cap N_u$. We call this approach the *Surrogate* approach.

On the surface, this approach might seem to be a simple modification from *Direct* approach. However, notice the following implication, which is very significant to the algorithm: once P_j receives N_v, it can extract the information of all nodes u, such that u is in both N_v and V_j^c, by scanning N_v only. For all such nodes u, P_j counts triangles incident on edge (u, v) by performing the operation $N_v \cap N_u$. P_j then discards N_v, since it is no longer needed. Note that extracting all u such that $u \in N_v$ and $u \in V_j$ requires $O(d_v)$ time (compare this to $O(\sum_{v \in V_i^c} d_v)$ time of direct approach for the same purpose). In fact, this extraction can be done while computing triangles $N_v \cap N_u$ for first such u. This avoids any additional overhead.

As we noticed, if delegated, P_j can count triangles on multiple edges (u, v) from a single message N_v, where $v \in V_i^c$ and $u \in N_v \cap V_j^c$. Thus P_i does not require to send N_v to P_j multiple times for each such u. However, to avoid multiple sending, P_i needs to keep track of which processors it has already sent N_v to. This *message tracking* needs to be done carefully, otherwise any additional space or runtime overhead might compromise the efficiency of the overall approach.

It is easy to see that one can perform the above tracking by maintaining p *flag* variables, one for each processor. Before sending N_v to a particular processor P_j, P_i checks the j-th flag to see if it is already sent. This implementation is conceptually simple but the cost for resetting flags for each

$v \in V_i^c$ sums to a significant cost of $O(|V_i^c|p)$. Now notice that an overhead of $O(|V_i^c|p)$ will lead to a runtime of at least $\Omega(n)$ because $\max_i |V_i^c| \geq \frac{n}{p}$. An algorithm with $\Omega(n)$ will not be scalable to a large number of processors since with the increase of p, the runtime $\Omega(n)$ does not decrease.

Now, observe the following simple yet useful property of N_v: *Since V_j^c is a set of consecutive nodes, and all neighbor lists N_v are sorted, all nodes $u \in N_v \cap V_j^c$ reside in N_v in consecutive positions.* This property enables each P_i to track messages by only recording the last processor (say, *LastProc*) it has sent N_v to. When P_i encounters $u \in N_v$ such that $u \in V_j^c$, it checks *LastProc*. If *LastProc* $\neq P_j$, then P_i sends N_v to P_j and set *LastProc* $= P_j$. Otherwise, the node u is ignored, meaning it would be redundant to send N_v. Resetting a single variable *LastProc* has a overhead of $O(|V_i^c|)$ as opposed to $O(|V_i^c|.p)$.

Thus surrogate approach detects and eliminates message redundancy and allows multiple computation from a single message, without even compromising execution or space efficiency. The efficiency gained from this capability is shown experimentally in Section 4.3.

4.2.3 Pseudocode for Counting Triangles.

We denote a message by $\langle t, X \rangle$ where $t \in \{data, control\}$ is the type and X is the actual data associated with the message. For a data message ($t = data$), X refers to a neighbor list N_x whereas for a control ($t = control$), $X = \emptyset$. The pesudocode for counting triangles for an incoming data message $\langle data, X \rangle$ is given in Figure 4.1.

```
1: Procedure SURROGATECOUNT(X, i) :
2: T ← 0   // T is the count of triangles
3: for all u ∈ X such that u ∈ V_i^c do
4:     S ← N_u ∩ X
5:     T ← T + |S|
6: return T
```

Figure 4.1: The procedure executed by P_i after receiving message $\langle data, X \rangle$ from some P_j.

Once a processor P_i completes the computation on all $v \in V_i^c$, it broadcasts a completion message $\langle control, \emptyset \rangle$. However, it cannot terminate execution until it receives $\langle control, \emptyset \rangle$ from all other processors since other processors might send data messages for surrogate computation. Finally, P_0 sums up counts from all processors using MPI aggregation function. The complete pseudocode of our algorithm using surrogate approach is presented in Figure 4.2.

4.2.4 Partitioning and Load Balancing

While constructing partitions i, set of nodes V is partitioned into p disjoint subsets V_i^c of consecutive nodes. Ideally, the set V should be partitioned in such a way that the cost for counting triangles is almost equal for all processors. Similar to our fast parallel algorithm presented in Chapter 3, we

```
1:  T_i ← 0   //T_i is P_i's count of triangles
2:  for each v ∈ V_i^c do
3:     for u ∈ N_v do
4:        if u ∈ V_i^c then
5:           S ← N_v ∩ N_u
6:           T_i ← T_i + |S|
7:        else
8:           Send ⟨data, N_v⟩ to P_j, where u ∈ V_j, if not sent already
9:
10:    for each incoming message ⟨t, X⟩ do
11:       if t = data then
12:          T_i ← T_i+ SurrogateCount(X, i) // See Figure 4.2
13:       else
14:          Increment completion counter
15:
16: Broadcast ⟨control, ∅⟩
17: while completion counter < p-1 do
18:    for each incoming message ⟨t, X⟩ do
19:       if t = data then
20:          T_i ← T_i+ SurrogateCount(X, i) // See Figure 4.2
21:       else
22:          Increment completion counter
23:
24: MpiBarrier
25: Find Sum T ← ∑_i T_i using MpiReduce
```

Figure 4.2: An algorithm for counting triangles using surrogate approach. Each processor P_i executes Line 1-22. After that, they are synchronized, and the aggregation is performed (Line 24-25).

need to compute p disjoint partitions of V such that for each partition V_i^c,

$$\sum_{v \in V_i^c} f(v) \approx \frac{1}{p} \sum_{v \in V} f(v). \tag{4.1}$$

Several estimations for $f(v)$ were proposed in Chapter 3 among which $f(v) = \sum_{u \in N_v} (\hat{d}_v + \hat{d}_u)$ was shown experimentally as the best. Since our algorithm employs a different communication scheme for counting triangles, none of those estimations corresponds to the cost of this algorithm. Thus, we derive a new cost function $f(v)$ to estimate the computational cost of our algorithm more precisely.

Deriving An Estimation for Cost Function $f(v)$. We want to find $f(v)$ such that $\sum_{v \in V_i^c} f(v)$ gives a good estimation of the computation cost incurred on processor P_i. We derive $f(v)$ as

follows.

Recall that $\mathcal{N}_v = \{u : (u, v) \in E\}$ and $N_v = \{u : (u, v) \in E, v \prec u\}$. Then, it is easy to see that

$$u \in \mathcal{N}_v - N_v \Leftrightarrow v \in N_u. \tag{4.2}$$

Now, P_i performs two types of computations due to all $v \in V_i^c$ as follows.

1. *Surrogate or delegated computation:* P_i compute $N_v \cap N_u$ for all $v \in N_u$ and $u \in V_j^c$, $i \neq j$, i.e., $u \in (\mathcal{N}_v - N_v) \cap (V - V_i^c)$. The cost incurred on P_i for such u and v is given by

$$\Theta\left(\sum_{v \in V_i^c} \sum_{u \in (\mathcal{N}_v - N_v) \cap (V - V_i^c)} (\hat{d}_v + \hat{d}_u)\right).$$

2. *Local computation:* P_i compute $N_v \cap N_u$ for all $u \in N_v \cap V_i^c$. Let E_i^c be the set of edges (u, v) where both u and v are in V_i^c, i.e., $E_i^c = \{(u, v) \in E | u, v \in V_i^c\}$. Now, the cost incurred on P_i for local computations is given by

$$\begin{aligned} & \Theta\left(\sum_{v \in V_i^c} \sum_{u \in N_v \cap V_i^c} (\hat{d}_v + \hat{d}_u)\right) \\ = \; & \Theta\left(\sum_{(u,v) \in E_i^c} (\hat{d}_v + \hat{d}_u)\right) \\ = \; & \Theta\left(\sum_{v \in V_i^c} \sum_{u \in (\mathcal{N}_v - N_v) \cap V_i^c} (\hat{d}_v + \hat{d}_u)\right). \end{aligned}$$

By adding costs from (1) and (2) above, we get the computation cost,

$$\Theta\left(\sum_{v \in V_i^c} \sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)\right).$$

Now, if we assign $f(v) = \left(\sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)\right)$, the computation cost incurred on P_i becomes $\sum_{v \in V_i^c} f(v)$. Thus, we use the following cost function:

$$f(v) = \left(\sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)\right).$$

Parallel Computation of the Cost Function $f(v)$. In parallel, each processor P_i computes $f(v)$ for all $v \in C_i$. Recall that C_i is the set of all nodes in the i-th chunk, as discussed in Section 3.3.4. Function $f(v) = \left(\sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)\right)$ is computed as follows.

i. First P_i computes $\hat{d}_v$, $v \in C_i$: computing $\hat{d}_v$ requires d_u for all $u \in \mathcal{N}_v$. Let $u \in C_j$. Then, P_i sends a request message to P_j, and P_j replies with a message containing d_u.

ii. Then P_i finds $\hat{d}_u$ for all $u \in \mathcal{N}_v - N_v$: let $u \in C_j$. P_i sends a request message to P_j, and P_j replies with a message containing $\hat{d}_u$.

iii. Now, $f(v) = \sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)$ is computed using $\hat{d}_v$ and $\hat{d}_u$ obtained in step (i) and (ii).

Computing Balanced Partitions. Once $f(v)$ is computed for all $v \in V$, we compute V_i^c using the same algorithm we used for overlapping partitioning as described in Chapter 3.

4.2.5 Correctness of the Algorithm

The correctness of our space efficient parallel algorithm is formally presented in the following theorem.

Theorem 6 *Given a graph $G = (V, E)$, our space efficient parallel algorithm counts every triangle in G exactly once.*

Proof. Consider a triangle (x_1, x_2, x_3) in G, and without the loss of generality, assume that $x_1 \prec x_2 \prec x_3$. By the constructions of N_x (Line 2-4 in Figure 3.2), we have $x_2, x_3 \in N_{x_1}$ and $x_3 \in N_{x_2}$. Now, there are two cases:

- *case 1.* $x_1, x_2 \in V_i^c$: Nodes x_1 and x_2 are in the same partition i. Processor P_i executes the loop in Line 2-6 (Figure 4.2) with $v = x_1$ and $u = x_2$, and node x_3 appears in $S = N_{x_1} \cap N_{x_2}$, and the triangle (x_1, x_2, x_3) is counted once. But this triangle cannot be counted for any other values of v and u because $x_1 \notin N_{x_2}$ and $x_1, x_2 \notin N_{x_3}$.
- *case 2.* $x_1 \in V_i^c, x_2 \in V_j^c, i \neq j$: Nodes x_1 and x_2 are in two different partitions i and j, respectively. P_i attempts to count the triangle executing the loop in Line 2-6 with $v = x_1$ and $u = x_2$. However, since $x_2 \notin V_i^c$, P_i sends N_{x_1} to P_j (Line 8). P_j counts this triangle while executing the loop in Line 10-12 with $X = N_{x_1}$, and node x_3 appears in $S = N_{x_2} \cap N_{x_1}$ (Line 4 in Figure 4.1). This triangle can never be counted again in any processor, since $x_1 \notin N_{x_2}$ and $x_1, x_2 \notin N_{x_3}$.

Thus, each triangle in G is counted once and only once. $\square$

4.2.6 Analysis of the Number of Messages

For $v \in V_i^c$, we call $(v, u) \in E$ a *cut edge* if $u \in V_j^c$, $j \neq i$. Let ℓ_{vj} is the number of cut edges emanating from node v to all nodes u in partition j with $v \prec u$. Now, in *Surrogate* approach, for all such cut edges (v, u), processor P_i sends N_v to P_j at most once instead of ℓ_{vj} times. This leads to a saving of the number of messages by a factor of ℓ_{vj} for each $v \in V_i^c$. To get a crude estimate of

how the number of messages for *direct* and *surrogate* approaches compare, let ℓ be the number of cut edges ℓ_{vj} averaged over all $v \in V_i^c$ and partitions j. Then, the number of messages exchanged in *direct* approach is roughly ℓ larger than *surrogate* approach.

As shown experimentally in Table 4.2, *direct* approach exchanges messages that is 4 to 12 times larger than that of *surrogate* approach. Thus, *surrogate* approach reduces approx. 70% to 90% of messages leading to faster computations as shown in the following section.

Table 4.2: Number of messages exchanged in Direct and Surrogate approaches.

Networks	**# of Messages**		*Ratio*
	Direct	Surrogate	
Miami	$16,321,478$	$3,987,871$	4.09
web-Google	$493,488$	$99,221$	4.97
LiveJournal	$23,138,824$	$4,002,575$	5.78
Twitter	$247,821,246$	$25,341,984$	9.78
PA(10M, 100)	$99,436,823$	$8,092,340$	12.29

4.3 Experimental Evaluation

In this section, we present the performance of our parallel algorithm with non-overlapping partitioning and compare it with other related algorithms. We will denote our algorithm with overlapping partitioning presented in Chapter 3 as **AOP** and the algorithm with non-overlapping partitioning as **ANOP** for the convenience of discussion.

Comparison with Previous Algorithms.

Algorithm *AOP* does not require message passing for counting triangles leading to a very fast algorithm (Table 4.3). In the contrary, *ANOP* achieves huge space saving over *AOP* (Table 4.1), although *ANOP* requires message passing for counting triangles. Our proposed communication approach (surrogate) reduces number of messages quite significantly leading to an almost similar runtime efficiency to that of *AOP*. In fact, *ANOP* loses only ∼20% runtime efficiency for the gain of a significant space efficiency of up to 25 times, thus allowing it to work on larger networks.

A runtime comparison among other related algorithms [54, 55, 72] for counting triangles in Twitter network is given in Figure 4.3. Our algorithm *ANOP* is 35, 17, and 7 times faster than that of [72], [54], and [55], respectively. Further, *ANOP* is almost as fast as *AOP*.

Strong Scaling.

Figure 4.4 shows strong scaling (speedup) of our algorithm *ANOP* on Miami, LiveJournal, and web-BerkStan networks with both direct and surrogate approaches. Speedup factors with the surrogate approach are significantly higher than that of the direct approach due to its capability to reduce communication cost drastically. Our algorithm demonstrates an almost linear speedup to a large number of processors.

Table 4.3: Runtime performance of our algorithms *AOP* and *ANOP*. We used 200 processors for this experiment. We showed both direct and surrogate approaches for *ANOP*.

Networks	**Runtime**			**Triangles**
	AOP	**Direct**	**Surrogate**	
web-BerkStan	0.10s	0.8s	0.14s	65M
Miami	0.6s	3.85s	0.79s	332M
LiveJournal	0.8s	5.12s	1.24s	286M
Twitter	9.4m	35.49m	12.33m	34.8B
PA(1B, 20)	15.5m	78.96m	20.77m	0.403M

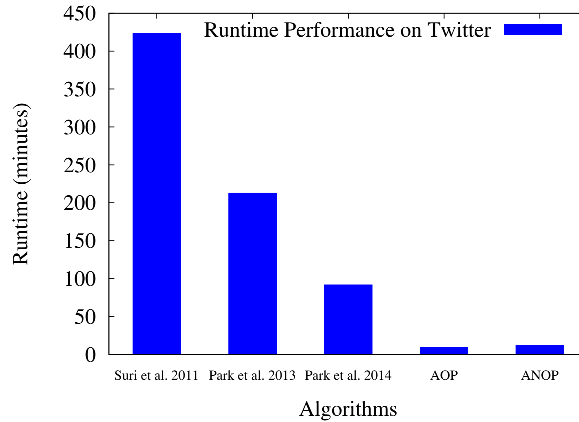


Figure 4.3: Runtime reported by various algorithms for counting triangles in Twitter network.

Further, *ANOP* scales to a higher number of processors when networks grow larger, as shown in Figure 4.5. This is, in fact, a highly desirable behavior since we need a large number of processors when the network size is large and computation time is high.

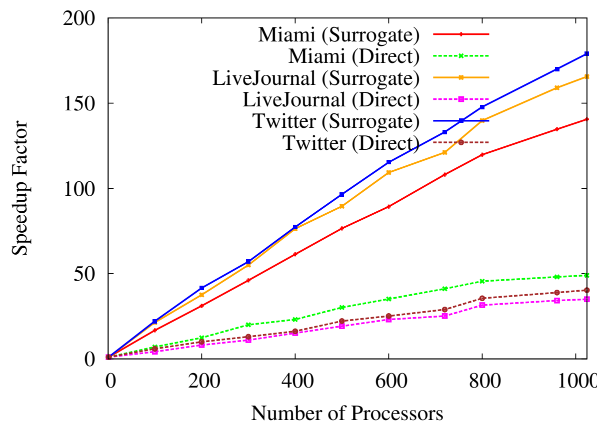


Figure 4.4: Speedup factors of our algorithm with both direct and surrogate approaches.

Effect of Estimation for f(v). We show the performance of our algorithm *ANOP* with the new cost

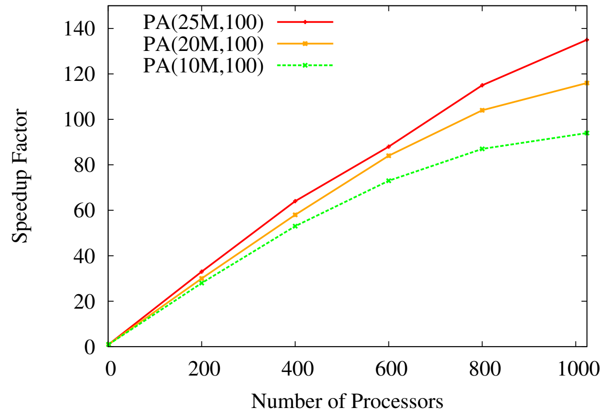


Figure 4.5: Improved scalability of our algorithm with increasing network size.

function $f(v) = \sum_{u \in \mathcal{N}_v - N_v} (\hat{d}_v + \hat{d}_u)$ and the best function $g(v) = \sum_{u \in N_v} (\hat{d}_v + \hat{d}_u)$ computed for *AOP*. As Figure 4.6 shows, *ANOP* with $f(v)$ provides better speedup than that with $g(v)$. Function $f(v)$ estimates the computational cost more precisely for *ANOP* with surrogate approach, which leads to improved load balancing and better speedup.

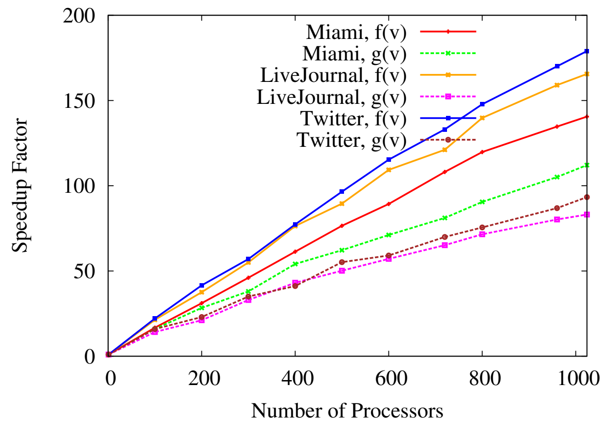


Figure 4.6: Comparison of the cost function $f(v)$ estimated for our algorithm with non-overlapping partitioning and the best function $g(v)$ in Chapter 3.

Weak Scaling. Weak scaling of a parallel algorithm measures its ability to maintain constant computation time when the problem size grows proportionally with processors. The weak scaling of our algorithm is shown in Figure 4.7. Since the addition of processors causes the overhead for exchanging messages to increase, the runtime of the algorithm increases slowly. However, as the change in runtime is rather slow (not drastic), our algorithm demonstrates a reasonably good weak scaling.

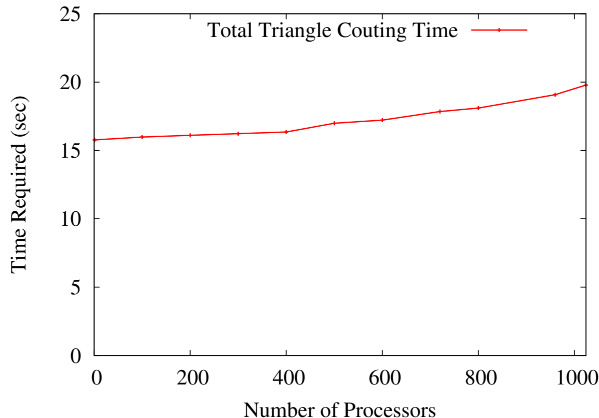


Figure 4.7: Weak scaling of our algorithm, experiment performed on PA($t/10 * 1M, 50$) networks, $t =$ number of processors used.

Table 4.4: Accuracy of our parallel sparsification algorithm and DOULION [76] with $q = 0.1$. Our parallel algorithm was run with 100 processors. Variance, max error and average error are calculated from 25 independent runs for each of the algorithms. The best values for each attribute are marked as bold.

Networks	Variance			Avg. error (%)			Max error (%)		
	AOP	ANOP	DOULION	AOP	ANOP	DOULION	AOP	ANOP	DOULION
web-BerkStan	**1.287**	1.991	2.027	**0.389**	0.391	0.392	**1.024**	1.082	1.082
LiveJournal	**1.770**	1.952	1.958	**1.463**	1.857	1.862	**3.881**	4.774	4.752
web-Google	**1.411**	2.003	1.998	**1.327**	1.564	1.580	**2.455**	3.923	3.942
Miami	**1.675**	2.105	2.112	**1.55**	1.921	1.905	**3.45**	4.88	4.75

4.4 Sparsification-based Parallel Approximation Algorithms

We discussed in Section 3.4 how our parallel algorithms with overlapping partitioning (AOP) can be adapted to design a parallel approximation algorithm. For all networks, our parallel sparsification algorithm with AOP results in smaller variance and errors than that of DOULION. We can also adapt our space-efficient algorithm with non-overlapping partitioning (ANOP) to devise an approximation algorithm based on DOULION.

Although our adapted version of DOULION with AOP provides more accuracy than DOULION, the adaptation with ANOP provides the same accuracy as original DOULION. That is, the accuracy does not improve for parallel sparsification with non-overlapping partitioning. This is evident in our experimental results presented in Table 4.4. Since the partitioning is non-overlapping, the effect of parallel sparsification is the same as that of the sequential sparsification. Further, we show in Table 4.5 a comparison of accuracies of parallel sparsification with both AOP and ANOP with the original DOULION for various sparsification factors q. AOP has better accuracies than the other two, and the accuracies with ANOP and DOULION are effectively the same.

The use of sparsification with our parallel algorithm ANOP will allow us to work with even larger

Table 4.5: Comparison of accuracy between our parallel sparsification algorithms and DOULION on one realistic synthetic and three real-world networks with 100 processors. The best values for each q are marked as bold.

Networks	Algorithms	$q = 0.1$	$q = 0.2$	$q = 0.3$	$q = 0.4$	$q = 0.5$
web-BerkStan	AOP	**99.9921**	**99.9927**	**99.9932**	**99.9947**	**99.9979**
	ANOP	99.6308	99.7490	99.8392	99.9168	99.9565
	DOULION	99.6309	99.7484	99.8401	99.9171	99.9566
LiveJournal	AOP	**99.9914**	**99.9917**	**99.9924**	**99.9936**	**99.9971**
	ANOP	99.6325	99.7488	99.8412	99.9178	99.9575
	DOULION	99.6310	99.7544	99.8392	99.9121	99.9584
web-Google	AOP	**99.9917**	**99.9923**	**99.9929**	**99.9939**	**99.9975**
	ANOP	99.6299	99.7391	99.8435	99.9168	99.9577
	DOULION	99.6305	99.7398	99.8428	99.9170	99.9574
Miami	AOP	**99.9916**	**99.9919**	**99.9926**	**99.9938**	**99.9974**
	ANOP	99.6285	99.7495	99.8384	99.9168	99.9562
	DOULION	99.6288	99.7494	99.8381	99.9169	99.9563

networks. Further, sparsification technique also offers additional speedup due to working on a reduced graph. For applications requiring only an approximate count of the total triangles with a reasonable accuracy, such parallel sparsification algorithm will be useful.

4.5 Conclusion

We present a space-efficient parallel algorithms for counting the *exact* number of triangles in massive networks. The algorithm employs non-overlapping partitions and reduces the space requirement significantly leading to the ability to work on larger networks. An efficient communication approach reduces message passing drastically to provide a fast algorithm. Our computation of a novel weight function for a parallel partitioning scheme adds further to the efficiency of the algorithm. We also provide a comprehensive theoretical analysis to justify the performance of the algorithm. We believe that for emerging massive networks, this algorithm will prove very useful.

Chapter 5

A Fast Parallel Algorithm for Counting Triangles in Networks using Dynamic Load Balancing

In this chapter, we present a fast MPI-based parallel algorithm for counting triangles in large networks using dynamic load balancing. Existing distributed memory parallel algorithms for counting the exact number of triangles are either Map-Reduce or message passing interface (MPI) based. Map-Reduce based algorithms generate prohibitively large intermediate data and do not demonstrate reasonably good runtime efficiency. The MPI-based algorithms offer fast computation of the number of triangles. However, the partitioning and load balancing schemes these algorithms employ are static in nature; the partitions are precomputed based on some estimations. In this work, we consider the case where the main memory of each compute node is large enough to contain the entire network. We observe that for such a case, computation load can be balanced dynamically and present a dynamic load balancing scheme that improves the performance of the algorithm significantly. Our algorithm demonstrates very good speedups and scales to a large number of processors. The algorithm computes the exact number of triangles in a network with 1 billion edges in 2 minutes with only 100 processors. Our results demonstrate that the algorithm is significantly faster than the related algorithms with static partitioning. In fact, for the real-world networks we experimented on, our algorithm achieves at least 2 times runtime efficiency over the fastest algorithm with static load balancing.

5.1 Introduction

We presented an MPI-based parallel algorithm [8] for counting the *exact* number of triangles in Chapter 3. The algorithm employs an overlapping partitioning scheme and a novel load balancing scheme. This algorithm does not require any inter-processor communication and is demonstrated to be very fast. Another MPI-based parallel algorithm [9] is presented in Chapter 4, which employs a non-overlapping partitioning and provides a space-efficient algorithm. Both of these algorithms partition the network such that each processor works on a single part (subgraph) of the network.

This allows these algorithm to work on very large networks. Further, both algorithms offer very fast computation. However, both algorithms are based on static load balancing. Besides, the second algorithm [9] involves exchanging data messages among processors, which reduces its runtime efficiency to some extent.

Now, with the overlapping partitioning scheme in [8], if the average degree of the input network is large (or the network has a few high degree nodes), the largest subgraph contains almost the entire network. Thus the algorithm requires storing the whole network in the memory of a single machine (which is assigned the largest subgraph). In such a case, we observe that if the system being used can accommodate the entire network in the main memory of a single machine, we can apply a dynamic load balancing scheme to further improve the runtime efficiency.

As reported by Leskovec et al. [56], due to the advancement of hardware technology, big-memory machines are becoming increasingly available and affordable. Designing efficient algorithms in such big-memory machine setting has also become an interesting line of work.

Contributions. In this chapter, we present an efficient MPI-based parallel algorithm for finding the exact number of triangles in a network where the memory of each machine is large enough to contain the entire network. We present a dynamic load balancing scheme that improves the performance of the algorithm significantly. Further, we not only assign computational task dynamically among processors, but also vary the task granularity on-the-fly. This dynamic re-adjustment of task granularity offers additional runtime efficiency. Our algorithm achieves very good speedups and scales well to a large number of processors. The algorithm computes the exact number of triangles in a network with 1B edges in only 2 minutes using 100 processors. Our results demonstrate that the algorithm is the fastest among the algorithms for counting the exact number of triangles. In fact, the algorithm is more than twice as fast as the previous fastest algorithm.

5.2 Comparison with Related Parallel Algorithms

The MapReduce based algorithm proposed in [72] works in two rounds of Map and Reduce phases. In Map phases, the algorithm generates a huge amount of intermediate data, which are all possible 2-paths w-v-u centered around each node $v \in V$ such that $u, w \in \mathcal{N}_v$. The algorithm then check whether such 2-paths are closed by an edge, i.e. if $(w, u) \in E$. Since the number of these 2-paths is very large, even larger than the network size, shuffling and regrouping these data requires a large runtime and enormous memory. As instance, for Twitter network, $300B$ 2-paths are generated whereas the network has only $2.4B$ edges. Even for smaller networks, if there are few nodes with high degrees, say $O(n)$, this algorithm generates $O(n^2)$ 2-paths centered at those nodes, which is quite unmanageable. Many real networks demonstrate power-law degree distributions where some nodes have very large degrees (see d_{max} in Table 5.1).

The MPI-based algorithm in [8] divides the input graph into a set of p overlapping subgraphs $G_i(V_i, E_i)$ as follows. First, V is partitioned into p disjoint subsets V_i^c, such that $\bigcup_{0 \le k < p} V_k^c = V$. Then, a set V_i is constructed as $V_i = V_i^c \cup \left(\bigcup_{v \in V_i^c} N_v \right)$. Now, set of edges E_i is defined as $E_i = \{(u, v) | u, v \in V_i, (u, v) \in E\}$. Processor P_i works on G_i. Note that edges in $E_i^c =$

$\{(u, v)|u \in V_i^c, v \in N_u\}$ constitute the *disjoint* (non-overlapping) portion of the partition i. Rest of the edges $(u, v) \in E_i - E_i^c$ overlaps with some other partitions.

Now, the overlapping partitioning allows the algorithm to count triangles without any communication among processors leading to faster computation. However, with overlapping partitioning, each processor requires a larger memory to store G_i. In fact, this is significantly larger when degrees of nodes of the network are large. Even if the average degree is small but the network has few nodes with high degrees, some subgraphs can be almost equal to the size of the original graph. Table 5.1 shows that real world networks have high degree nodes. In many cases, average degrees of networks are also high.

Table 5.1: Memory required for storing networks along with their average and maximum degree statistics.

Network	**Memory (GB)**	**Avg. d**	d_{max}
web-Google	0.127	11.6	6332
Miami	2.7	47.6	425
LiveJournal	2.4	18	20333
Twitter	23.7	57.1	1001159
PA(10M, 100)	18.3	100	25068

Another MPI-based algorithm presented in [9] divides the input networks into non-overlapping subgraphs. This partitioning provides the best space efficiency among the related algorithms. Space required to store individual subgraphs add up to the space required to store the whole network. However, such partitioning requires inter-processor communications for counting triangles. Although the paper [9] presents an efficient method to reduce the communication cost drastically making it a reasonably fast algorithm, exchanging messages still reduces its runtime efficiency to some extent. Note that algorithms in both [8, 9] employ static load balancing schemes based on some estimates for the cost of counting triangles. Different estimations (as referred to as *cost functions*) offer varying degree of performance in load balancing, and none of them are entirely precise. Thus, some processors might experience idle time.

Now consider the case that each computing machine has enough memory for storing the whole network. For such a case, we observe, unlike the algorithms in [8, 9], we can apply a dynamic load balancing scheme to reduce idle time of processors drastically and make the computation even faster. Further, since all processors store the whole network, we do not require the procedure to exchange data messages as required in [9].

In this chapter, we present an efficient parallel algorithm with dynamic load balancing, which is faster than the algorithms with static partitioning. Our algorithm exchanges only small control messages (request, response, or termination messages). This has very little communication overhead compared with [9]. To the best of our knowledge, this algorithm is the fastest among algorithms producing the exact count of triangles in big networks. We present a trade-off between space and runtime efficiency of three related MPI-based algorithms in Table 5.2.

Table 5.2: Trade-off between space and runtime efficiency of algorithms in [8, 9] and this chapter.

Algorithm	Space Eff.	Runtime Eff.
Non-overlapping part. [9]	Most efficient	Efficient
Overlapping part. [8]	Medium	Faster
Alg. in this chapter	Least efficient	Fastest

5.3 A Fast Parallel Algorithm with Dynamic Load Balancing

We present our parallel algorithm for counting triangles with an efficient dynamic load balancing scheme. First, we provide an overview of the algorithm, and then a detailed description follows.

5.3.1 Overview of the Algorithm

Let p be the number of processors used in our computation. Our algorithm distributes the computation of counting triangles on all nodes $v \in V$ in the network among these processors. We refer the computation assigned to and performed by a processor as a task. For the convenience of future discussion, we present the following definitions related to computing tasks.

Definition 3 ***Task:*** *Given a graph $G = (V, E)$, a task denoted by $\langle v, t\rangle$, refers to counting triangles incident on nodes in $\{v, v+1, \ldots, v+t-1\} \subseteq V$. The task referring to counting triangles in the whole network is $\langle 0, n\rangle$.*

Definition 4 ***An atomic task:*** *A task $\langle v, 1\rangle$ referring to counting triangles incident on a single node v is an atomic task. An atomic task cannot be further divided.*

Definition 5 ***Task size:*** *Let, $f : V \to \mathcal{R}$ be a cost function such that $f(v)$ denotes some measure of the cost for counting triangles on node v. We define the size $S(v, t)$ of a task $\langle v, t\rangle$ as follows.*

$$S(v, t) = \sum_{i=0}^{t-1} f(v + i).$$

A number of estimations for cost function $f(v)$ has been given in Chapter 3. Examples include $f(v) = 1$, $f(v) = d_v$, and $f(v) = \sum_{u \in N_v} (\hat{d_v} + \hat{d_u})$. Some of those provide better estimations than others but have a larger computational overhead. Since our algorithm balances load dynamically, using a computationally expensive cost function can increase the runtime of our algorithm. In this work, we use the cost functions $f(v) = 1$ and $f(v) = d_v$ since those are known for all $v \in V$ and have no computational overhead. The function $f(v) = 1$ corresponds to the same cost for each node, whereas $f(v) = d_v$ implies that the cost is proportional to the degree of node v.

In a static load balancing scheme, each processor works on a pre-computed partition. Since the partitioning is based on the estimated computing cost, which might not equal to the actual computing cost, some processors will remain idle after finishing computation ahead of others. Our algorithm employs a dynamic load balancing scheme to reduce idle time of processors, leading to

improved performance. The algorithm divides the total computation into several tasks and assign them dynamically. Determining how and when to assign a task requires communications among processors. The schemes for communication and deciding task granularity are crucial to the performance of our algorithm. Next, we describe the details of these schemes.

5.3.2 An Efficient Dynamic Load Balancing Scheme

We design a dynamic load balancing scheme with a dedicated processor for coordinating balancing decisions. We distinguish this processor as the *coordinator* and the rest as *workers*. The *coordinator* assigns tasks, receives notifications and re-assigns tasks to idle workers, and *workers* are responsible for actually performing *tasks*. At the beginning, each worker is assigned an initial task. Once any worker i completes its current task, it sends a request to the *coordinator* for an additional task. From the available un-assigned tasks, the *coordinator* assigns a new task to worker i. At the end, some processors still compute their respective tasks and some remain idle.

Assume the time required by some worker to compute the last completed task is q. The amount of time a worker remains idle, denoted by a continuous random variable X, can be assumed to be uniformly distributed over the interval $[0, q]$, i.e., $X \sim U(0, q)$. Since $E[X] = q/2$, a worker remains idle for $q/2$ amount of time on average. Now, the coordinator may divide the computation into tasks of equal size and assign them dynamically. However, the size of tasks is a crucial determinant of the performance of the algorithm. If the size $S(v, t)$ of tasks $\langle v, t\rangle$ is large, time q required to complete the last task becomes large, and consequently, idle time $q/2$ also grows large. In contrast, if the task size is small, the idle time is expected to decrease. However, if task size is very small, the total number of tasks becomes large, which increases communication overhead for task requests and re-assignments.

Therefore, instead of keeping the size of tasks $S(v, t)$ constant throughout the execution, our algorithm adjusts $S(v, t)$ dynamically, initially assigning large tasks and then gradually decreasing them. In particular, initially half of the total computation $\langle 0, n\rangle$ is assigned among the workers in tasks of almost equal sizes. Let t' be an integer such that $S(0, t') \approx \frac{1}{2}S(0, n)$. Task $\langle 0, t'\rangle$ is divided among $(P - 1)$ processors initially. The remaining computations $\langle t', n - t'\rangle$ are assigned dynamically with the granularity of tasks decreasing gradually, as described below.

Initial Assignment. The set of $(p - 1)$ initial tasks corresponds to counting triangles on nodes $v \in \{0, 1, \dots, t' - 1\}$ such that $S(0, t') \approx S(t', n - t')$. Thus we need to find node t' which divides the set of nodes V into two disjoint subsets in such a way that $\sum_{v=0}^{t'-1} f(v) \approx \sum_{v=t'}^{n-1} f(v)$, given $f(v)$ for each $v \in V$. Now if we compute sequentially, it takes $O(n)$ time to perform the above computations. However, we observe that a parallel algorithm for computing balanced partitions of V proposed in [8] can be used to perform the above computation which takes $O(n/p + \log p)$ time. Once t' is determined, the task $\langle 0, t'\rangle$ is divided into $(p - 1)$ tasks $\langle v_i, t_i\rangle$, one for each worker, in almost equal sizes, that is,

$$S(v_i, t_i) = \frac{S(0, t')}{p - 1}. \tag{5.1}$$

That is, the set of nodes $\{0, 1, \dots, t' - 1\}$ is divided into $(p - 1)$ subsets such that for each subset $\{v_i, v_i + 1, \dots, v_i + t_i - 1\}$, $\sum_{k=0}^{t_i-1} f(v_i + k) \approx \frac{1}{p-1} \sum_{v=0}^{t'-1} f(v)$. This computation can also be

done using the parallel algorithm [8] mentioned above. At the end of the algorithm, each worker P_i knows its initial task $\langle v_i, t_i \rangle$. All workers execute their initial tasks independently without involving the coordinator.

Dynamic Re-assignment. Once any worker completes its current task and becomes idle, the *coordinator* assigns it a new task dynamically.

Let the current task available to the coordinator to be assigned to a requesting worker be $\langle \hat{v}, \hat{t} \rangle$. Our algorithm decreases the size $S(\hat{v}, \hat{t})$ of each dynamically assigned tasks gradually. This is done using the following equation.

$$S(\hat{v}, \hat{t}) = \frac{S(\hat{v}, n - \hat{v})}{p - 1}. \tag{5.2}$$

Initially, $\hat{v} = t'$. After each assignment, $\hat{v}$ is updated as $\hat{v} \leftarrow \hat{v} + \hat{t}$. The coordinator knows the size of the remaining unassigned task, which is initially $S(t', n - t')$, and updates it each time by subtracting the size $S(\hat{v}, \hat{t})$ of the newly assigned task. To determine a new task $\langle \hat{v}, \hat{t} \rangle$, the coordinator finds $\hat{t}$ that satisfies Eqn. 5.2 by using $f(v)$ for $v \in \{\hat{v}, \dots, n - 1\}$.

By the definition of atomic task (in definition 4), $\hat{t}$ is at least 1 and thus we have a finite number of tasks. When the coordinator has no more unassigned tasks, it sends a special termination message $\langle terminate \rangle$ to the requesting workers. Once the coordinator completes sending termination messages to all workers, it aggregates counts of triangles from all workers, and the algorithm terminates.

Note that this scheme is quite efficient. However, while the coordinator determines a new task for dynamic assignment, a requesting worker might need to wait. This waiting can be avoided by pre-computing tasks $\langle \hat{v}, \hat{t} \rangle$. In fact, while workers are performing the initial assignment, the coordinator proceeds to determine tasks $\langle \hat{v}, \hat{t} \rangle$ for subsequent assignments and fills a task queue W. It can also determine tasks when it has no requests to serve. Thus when any worker requests further tasks, the coordinator can readily respond. Further, responding and receiving task requests have low communication overhead. Thus, the coordinator does not become a bottleneck in this algorithm.

5.3.3 Counting Triangles

Once a processor i has an assigned task $\langle v, t \rangle$, it uses the algorithm presented in Figure 5.1 to count the triangles incident on nodes in $\{v, v + 1, \dots, v + t - 1\}$.

The complete pseudocode of our algorithm for counting triangles with an efficient dynamic load balancing scheme is presented in Figure 5.2.

5.3.4 Correctness of the Algorithm

We establish the correctness of our algorithm as follows. Consider a triangle (x_1, x_2, x_3) with $x_1 \prec x_2 \prec x_3$, without the loss of generality. Now, the triangle is counted only when $x_1 \in$

```
1: Procedure COUNTTRIANGLES(v, t) :
2: T ← 0   // T is the count of triangles
3: for v ∈ {v, v+1, ..., v+t−1} do
4:     for u ∈ N_v do
5:         S ← N_v ∩ N_u
6:         T ← T + |S|
7: return T
```

Figure 5.1: A procedure executed by processor P_i to count triangles corresponding to the task $\langle v, t \rangle$.

$\{v, v+1, \ldots, v+t-1\}$ for some task $\langle v, t \rangle$. The triangle is never counted again since $x_1 \notin N_{x_2}$ and $x_1, x_2 \notin N_{x_3}$ by the construction of N_x (Line 1-3 in Figure 3.2).

5.3.5 Performance

We perform our experiments using a high performance computing cluster with 64 computing nodes (QDR InfiniBand interconnect), 16 processors (Sandy Bridge E5-2670, 2.6GHz) per node, memory 4GB per processor, and operating system CentOS Linux 6. The experimental evaluation of the performance our parallel algorithm for counting triangles with dynamic load balancing is presented below.

Strong Scaling. Strong scaling of a parallel algorithm shows how much speedup a parallel algorithm gains as the number of processors increases. We present the strong scaling of our algorithm on Miami, LiveJournal, and web-BerkStan networks with both cost functions $f(v) = 1$ and $f(v) = d_v$ in Figure 5.3. Our algorithm demonstrates very good speedups and scales almost linearly to a large number of processors. Further, speedup factors are significantly higher with the function $f(v) = d_v$ than with $f(v) = 1$. The function $f(v) = 1$ refers to equal cost of counting triangles for all nodes whereas the function $f(v) = d_v$ relates the cost to the degree of v. Distributing tasks based on the sum of degrees of nodes (Eqn. 5.1 and 5.2) reduces the effect of skewness of degrees and makes tasks more balanced leading to higher speedups. Our subsequent experiments will be based on cost function $f(v) = d_v$.

We also observe that the larger networks Miami and LiveJournal achieve higher speedups than web-BerkStan. This is, in fact, a desirable advantage when we want to process big graphs. For small networks, the communication overhead in load balancing becomes relatively significant affecting the speedups to some extent.

Comparison with Previous Algorithms. We compare the runtime of our parallel algorithm with the algorithms in [8] and [9] on a number of real and artificial networks. Note that both algorithms in Chapter 3 and 4 are demonstrated to be faster than the MapReduce based algorithms presented in [54, 55, 72] (Figure 4.3 in Chapter 4). We compare the runtime of our algorithm with these two state-of-the-art fast parallel algorithms. As shown in Table 5.3, our algorithm is more than 2 times faster than [8] and about 3 times than [9] for all these networks. We also count triangles in

1: **All processors, in parallel, do the following:**
2: **Determine** t' s.t. $S(0,t') = \frac{1}{2}S(0,n)$ using parallel alg. in [8]
3:
4: **All *workers* do the following:**
5: **Determine** initial tasks $\langle v_i, t_i \rangle$ using parallel alg. in [8]
6:
7: **The *coordinator* does the following:**
8: $W \leftarrow \emptyset$
9: $\hat{v} \leftarrow t'$
10: $t^r \leftarrow n - t'$
11: **while** $t^r > 0$ *OR* $W \neq \emptyset$ **do**
12:     **if** $t^r > 0$ **then**
13:         **Compute** $\hat{t}$ s.t. $S(\hat{v},\hat{t}) = \frac{S(\hat{v},t^r)}{p-1}$
14:         $W.\textsc{EnQueue}\left(\langle \hat{v},\hat{t} \rangle\right)$
15:         $\hat{v} \leftarrow \hat{v} + \hat{t}$
16:         $t^r \leftarrow n - \hat{v}$
17:     *PrevRequest*$\leftarrow$ *true*
18:     **while** $W \neq \emptyset$ *AND PrevRequest* **do**
19:         **if** Any *task* request $\langle i \rangle$ received **then**
20:             $\langle \hat{v},\hat{t} \rangle \leftarrow W.\textsc{DeQueue}()$
21:             **Send** message $\langle \hat{v},\hat{t} \rangle$ to worker i
22:         **else**
23:             *PrevRequest* $\leftarrow$ *false*
24: **Send** $\langle terminate \rangle$ for next $(p-1)$ task requests $\langle i \rangle$
25:
26: **Each *worker* P_i does the following:**
27: $T_i \leftarrow 0$
28: $T_i \leftarrow T_i + \textsc{CountTriangles}(v_i, t_i)$ //for *initial* task
29: *done* $\leftarrow$ *false*
30: **while** *not done* **do**
31:     **Send** message $\langle i \rangle$ to *coordinator*
32:     **Receive** message M from *coordinator*
33:     **if** M is $\langle terminate \rangle$ **then**
34:         *done* $\leftarrow$ *true*
35:     **else if** M is a *task* $\langle \hat{v},\hat{t} \rangle$ **then**
36:         $T_i \leftarrow T_i + \textsc{CountTriangles}(\hat{v},\hat{t})$
37:
38: $\textsc{MpiBarrier}$
39: Find Sum $T \leftarrow \sum_i T_i$ in parallel using $\textsc{MpiReduce}$
40: **return** T

Figure 5.2: An algorithm for counting triangles with dynamic load balancing.

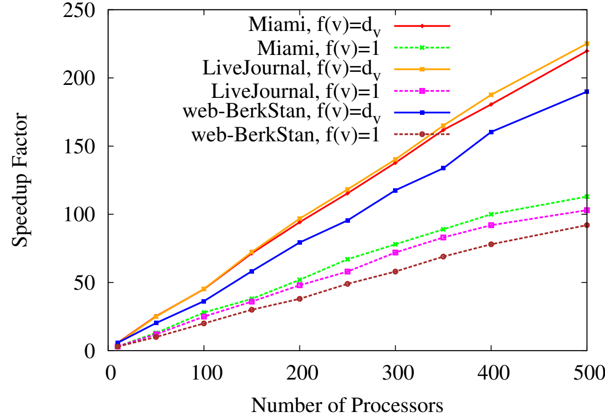


Figure 5.3: Speedup factors of our algorithm on Miami, LiveJournal and web-BerkStan networks with both $f(v) = 1$ and $f(v) = d_v$ cost functions.

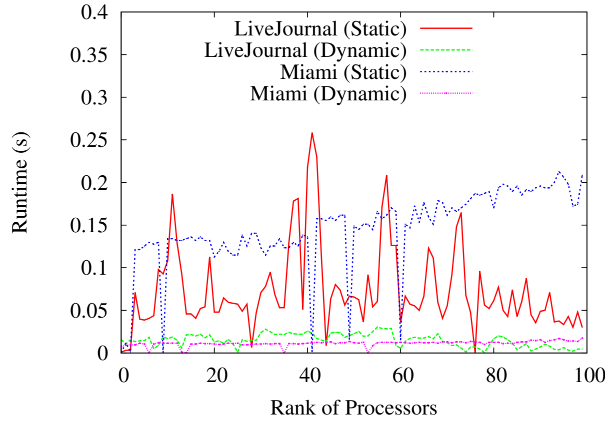


Figure 5.4: Runtime required by processors (rankwise) with both static tasks and dynamic adjustment of task granularity.

Twitter network (having 2.4B edges), which requires 8 minutes with only 100 processors. This is significantly faster than [8] and [9] which use even twice as much processors. The algorithm in [8] and [9] are based on static partitioning whereas our algorithm employs a dynamic load balancing scheme to reduce idle time of processors leading to improved performance.

We also present a comparison of speedup factors for our algorithm and the algorithms in [8] and [9] on Miami and LiveJournal networks in Figure 5.7. Our algorithm achieves significantly higher speedups.

We also notice the reported performance of several shared memory parallel algorithms. The parallel approximation algorithm in [73] demonstrates a speedup of ≈ 11 with 12 cores. However, it is not clear how the algorithm will scale for a larger number of cores (or processors). As we demonstrated, our algorithm scales almost linearly to a large number of processors. Another shared memory based parallel approximation algorithm is proposed in [60]. The paper reports speedups

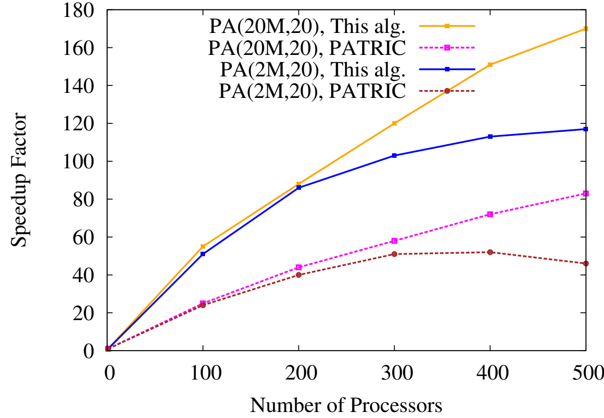


Figure 5.5: Our algorithm with dynamic load balancing shows improved scalability with increasing network size. Further, this algorithm achieves higher speedups than PATRIC (in Chapter 3).

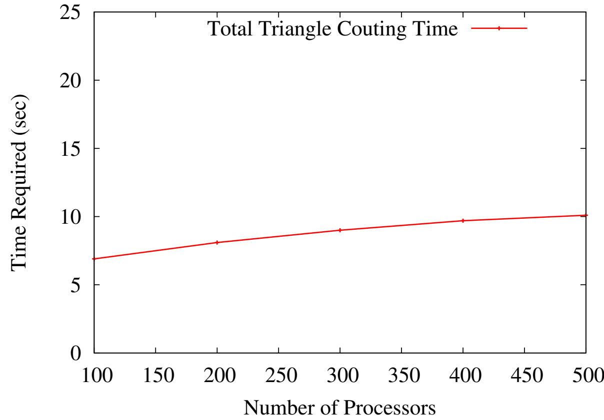


Figure 5.6: Weak scaling of our algorithm. We perform this experiment on PA($t/10 * 1M, 50$) networks, $t =$ number of processors used.

using only 32 cores. Further, these speedups are due to both approximation and parallel threads. For example, with a sample factor $p = 0.01$, the paper reports a speedup of 837.74 for Wiki-1 graph with 32 threads, where the approximation contributes a factor of 33.54 in the speedup. The results

Table 5.3: Runtime performance of our algorithm and algorithm [8].

Networks	**Runtime**			**Triangles**
	[9]	[8]	Our algo.	
web-BerkStan	0.14	0.10s	0.041s	65M
LiveJournal	1.24	0.8s	0.384s	286M
Miami	0.79	0.6s	0.301s	332M

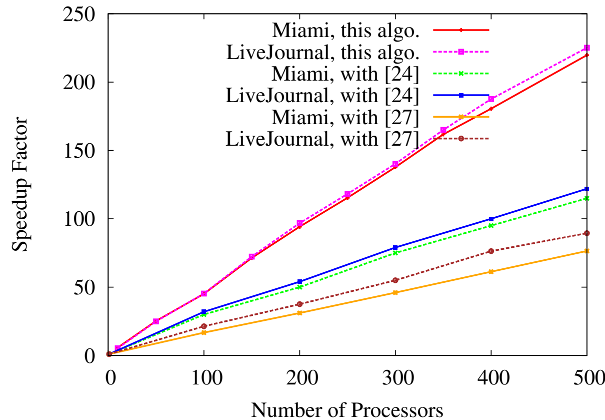


Figure 5.7: Comparison of speedup factors of our algorithm with [8] and [9] on Miami and Live-Journal networks.

for other networks demonstrate a parallelization speedup between 1.44 and 24 with 32 threads. Though some networks show good speedups, many of them do not. Further, results for a larger number of cores are not shown in the paper. Similarly, the shared memory algorithm in [68] is reported to scale to 64 cores and achieves speedups ranging from 17 to 50 .

Effect of Dynamic Adjustment of Task Granularity. We show how the granularity of tasks affects the idle time of worker processors for Miami and LiveJournal networks. As Figure 5.4 shows, with tasks of static (equal) size, the distribution of runtime among processors are very uneven leading to large idle times of some processors. However, dynamic adjustment of task granularity (gradual decrease of task size) provides an almost even distribution of runtime leading to very short idle times. This allows balanced computing loads among processors and consequently improves the runtime performance of the algorithm. Note that we used 100 processors for this experiment. Although we could use a higher number of processors, using fewer processors helped demonstrate the differences in idle times more clearly. In our next experiment, we show that our algorithm scales to higher number of processors when networks grow larger.

Scaling with Processors and Network Size. Our algorithm scales to a higher number of processors when networks grow larger, as shown in Figure 5.5. This is, in fact, a highly desirable behavior since we need a large number of processors when the network size is large and computation time is high. Scaling of our algorithm with number of processors is very comparable to that of [8]. To our advantage, our algorithm achieves significantly higher speedup factors than [8].

Weak Scaling. Weak scaling of a parallel algorithm shows the ability of the algorithm to maintain constant computation time when with the increase of the number of processors, the problem size also grows proportionally. The weak scaling of our algorithm is shown in Figure 5.6. With the addition of processors, communication overhead increases since idle workers exchange messages with the coordinator for new tasks. However, since the overhead for requesting and assigning tasks is very small, the increase of runtime with additional processors is rather slow (not drastic). Thus, our algorithm demonstrates a reasonably good weak scaling.

5.4 Conclusion

We present a fast parallel algorithms for counting triangles in large networks. When the main memory of each computing machine is large enough to store the whole network, our parallel algorithm with dynamic load balancing can be used for faster analysis of the network. We believe that for emerging big networks, this algorithm will be proven very useful.

Chapter 6

Applications of Our Algorithms for Counting Triangles

In this chapter, we present how our parallel algorithms for counting triangles can be used for listing all triangles in networks. Such listing or enumeration has useful applications in finding local patterns and computing clustering coefficients of nodes in networks. We also present a scalable parallel algorithm for computing clustering coefficients based on our algorithms for enumerating triangles. Finally, we discuss some other applications of counting triangles and demonstrate how the number of triangles can be used to comment on the general structure of networks.

6.1 Listing Triangles in Graphs

Our parallel algorithms for counting triangles in Chapter 3, 4 and 5 can easily be extended to list all triangles in graphs. Triangle listing has various applications in the analysis of graphs such as the computation of clustering coefficients, transitivity, triangular connectivity, and trusses [22]. Our parallel algorithms counts the exact number of triangles in the graph. To count the number of triangles incident on an edge (u, v), the algorithms perform a set intersection operation $N_v \cap N_u$. After each intersection operation, all associated triangles can be listed simply by the code shown in Figure 6.1.

1: $S \leftarrow N_v \cap N_u$
2: **for** $w \in S$ **do**
3: *Output* triangle (u, v, w)

Figure 6.1: Listing triangles after performing the set intersection operation for counting triangles.

6.2 Computing Clustering Coefficient of Nodes

Our parallel algorithms can be extended to compute local clustering coefficient without increasing the cost significantly. In a sequential setting, an algorithm for counting triangles can be directly used for computing clustering coefficients of the nodes by simply keeping the counts of triangles for each node individually. However, in a distributed-memory parallel system, combining the counts from all processors for a node poses another level of difficulty. We present an efficient aggregation scheme for combining the counts for a node from different processors.

Parallel Computation of Clustering Coefficients. Recall that clustering coefficients of nodes v is computed as follows:

$$C_v = \frac{T_v}{\binom{d_v}{2}} = \frac{2T_v}{d_v(d_v - 1)},$$

where T_v is the number of triangles containing node v.

Our parallel algorithms for counting triangles count each triangle only once. However, all triangles containing a node v might not be computed by a single processor. Consider a triangle (u, v, w) with $u \prec_{\mathcal{D}} v \prec_{\mathcal{D}} w$. Further, assume that $u \in V_i^c$, $v \in V_j^c$, and $w \in V_k^c$, where $i \neq j \neq k$. Now, for our parallel algorithm *AOP* (presented in Chapter 3), the triangle (u, v, w) is counted by P_i. Let T_v^i be the number of triangles incident on node v computed by P_i. We also call such counts *local counts* of v in processor P_i. For the triangle (u, v, w), P_i tracks local counts of all of u, v, and w. Thus, the total count of triangles incident on a node v might be distributed among multiple processors. Each processor P_i needs to aggregate local counts of $u \in V_i^c$ from other processors. (For our algorithm *ANOP* presented in Chapter 4, the above triangle (u, v, w) is counted by P_j, and a similar argument as above holds.)

To aggregate local counts from other processors, the following approach can be adopted: for each processor, we can store local counts T_v^i in an array of size $\Theta(n)$ and then use MPI *All-Reduce* function for the aggregation. However, for a large network, the required system buffer to perform the MPI aggregation on arrays of size $\Theta(n)$ might be prohibitive. Another approach for aggregation might be as follows. Instead of using main memory, local counts can be written to disk files based on some hash functions of nodes. Each processor P_i then aggregates counts for nodes $v \in V_i^c$ from P disk files. Even though this scheme saves the usage of main memory, performing a large number of disk I/O leads to a large runtime.

Both of the above approach compromises either the runtime or space efficiency. We use the following approach which is both time and space efficient.

Our approach involves two steps. First, for each triangle counted by P_i, it tracks local counts $T.^i$ as shown in Figure 6.2.

Second, processor P_i aggregates local counts of nodes $v \in V_i^c$ from other processors. Total number of triangles T_v incident on v is given by $T_v = \sum_{j \neq i} T_v^j$. Each processor P_j sends local counts T_v^j of nodes $v \in V_i^c$ encountered in any triangles counted in partition j. P_i receives those counts and aggregates to T_v. We present the pseudocode of this aggregation in Figure 6.3. Finally, P_i computes $C_v = \frac{2T_v}{d_v(d_v-1)}$ for each $v \in V_i^c$.

```
1: for for each triangle (v, u, w) counted in G_i do
2:     T_v^i ← T_v^i + 1
3:     T_u^i ← T_u^i + 1
4:     T_w^i ← T_w^i + 1
```

Figure 6.2: Tracking local counts by processor P_i. Each triangle (v, u, w) is detected by the triangle listing algorithm shown in Figure 6.1.

```
1: for v ∈ V_i^c do
2:     T_v ← T_v^i
3: for each processor P_j do
4:     Construct message ⟨Y_i^j, 𝒯_i^j⟩ s.t.:
       Y_i^j ← {v|v ∈ N_u, u ∈ V_i^c} ∩ V_j^c, 𝒯_i^j ← {T_v^i|v ∈ Y_i^j}.
5:     Send message ⟨Y_i^j, 𝒯_i^j⟩ to P_j
6: for each processor P_j do
7:     Receive message ⟨Y_j^i, 𝒯_j^i⟩ from P_j
8:     T_v ← T_v + T_v^j
```

Figure 6.3: Aggregating local counts for $v \in V_i^c$ by P_i.

Our approach tracks local counts for nodes $v \in V_i^c$ and neighbors of such v which requires, in practice, significantly smaller than $\Theta(n)$ space. Next, we show the performance of our algorithm.

Performance. We show the strong and weak scaling of our algorithm for computing clustering coefficients of nodes in Figure 6.4 and 6.5, respectively. The algorithm shows good speedups and scales almost linearly to a large number of processors. Since aggregating local counts introduces additional inter-processor communication, the speedups are a little smaller than that of the triangle counting algorithms. For the same reason, the weak scalability of the algorithm is a little smaller than that of the triangle counting algorithms. However, the increase of runtime with additional processors is still not drastic, and the algorithm shows a good weak scaling.

6.3 Other Applications for Counting Triangles

The number of triangles in graphs have many important applications in data mining. Becchetti et al. [15] showed how the number of triangles can be used to detect spamming activity in web graphs. They used a public web spam dataset and compared it with a non-spam dataset: first, they computed the number of triangles for each host and plotted the distribution of triangles and clustering coefficients for both dataset. Using Kolmogorov-Smirnov test, they concluded the distributions are significantly different for spam and non-spam datasets. Further, the authors also showed how to comment on the role of individual nodes in a social network based on the number of triangles they participate. Eckmann et al. [30] used triangle counting in uncovering the thematic structure of

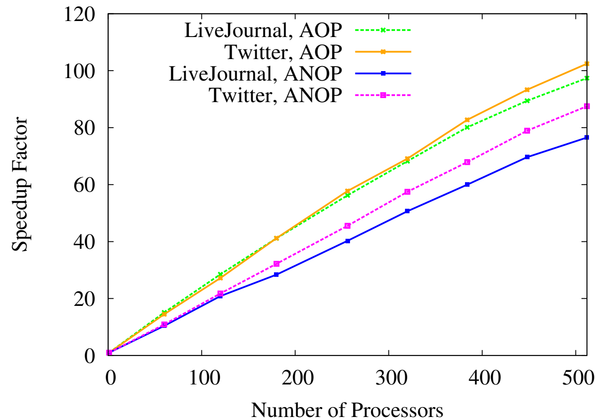


Figure 6.4: Strong scaling of clustering coefficient algorithm with both *AOP* and *ANOP* on LiveJournal and Twitter networks.

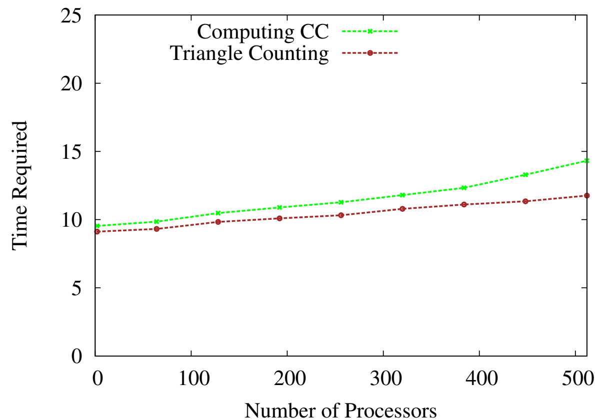


Figure 6.5: Weak scaling of the algorithms for computing clustering coefficient (CC) and counting triangles (TC).

the web. The abundance of triangles also implies community structures in graphs. Nodes forming a subgraph of high triangular density usually belong to the same community. In fact, the number of triangles incident on nodes has been used by several methods in the literature of community detection [57, 70, 81]. The computation of clustering coefficients also requires the number of triangles incident on nodes. Social networks usually demonstrate high average clustering coefficients. We show how clustering coefficients can be computed using our parallel algorithms in Section 6.2.

In this section, we discuss how the number of triangles can be used to characterize various types of networks. There is a multitude of real-world networks including social contact networks, online social networks, web graphs, and collaboration networks. These networks vary in terms of triangular density and community or social structure in them. As a result, it is possible to characterize real-world networks based on their triangle based statistics. We define the *normalized triangle count* (**NTC**) as the mean number of triangles per node in the network. We compute NTC for a

Table 6.1: Comparison of the number of triangles ($\triangle$) and normalized triangle count (NTC) in various networks. We used both artificially generated and real-world networks.

Network	n	$\triangle$	**NTC**$(\triangle/n)$
Gnp$(500K, 20)$	500K	1308	0.0026
PA$(25M, 50)$	25M	1.3M	0.052
Email-Enron	37K	727044	19.815
web-Google	0.88M	13.39M	15.293
LiveJournal	4.85M	285.7M	58.943
web-BerkStan	0.69M	64.69M	94.408
Miami	2.1M	332M	158.095
com-Orkut	3.07M	628M	204.262
Twitter	42M	34.8B	828.571

variety of networks and show the comparison in Table 6.1. Many random graph models such as Erdős-Réyni and Preferential Attachment models do not generate many triangles, and the resulting NTCs are also very low. Some communication and web graphs (e.g., Email-Enron) generate a descent number of triangles because of the nature of the communication and links among web pages in the host domain. When social or cluster structure exists in the network, we get a larger number of triangles per node, as shown in Table 6.1 for LiveJournal and web-BerkStan networks. Further, for networks with a more developed social structure and realistic person-to-person interactions, NTCs are very large, as evident for Miami, com-Orkut, and Twitter networks. Thus the number of triangles offers good insights about the underlying social and community structures in networks.

Part II

Characterizing Networks Based on Common Neighbor Statistics

Chapter 7

How Much Common Neighbors Can Reveal about Networks

Characterizing social and information networks based on some properties has been of growing interest. Degree distribution, the number of triangles, clustering coefficients, and diameter are among the most explored properties. An important property, related to triangles, of many networks, mostly social networks, is high transitivity, which states that two nodes having common neighbors tend to become neighbors to one another. In this chapter, we present a characterization of networks by quantifying the number of common neighbors and demonstrate its relationship with other network properties. Among others, we answer the following questions: how much does the number of common neighbors tell about forming an edge between two nodes? How do common neighbor statistics relate to community structure of networks? Based on the Jaccard indices of edges, we observe that there is an interesting threshold behavior of two nodes connecting by an edge in the social and information networks we examined. We also demonstrate how common neighbor statistics relate to community structure of networks.

7.1 Introduction

Since a graph is a powerful abstraction of a complex system, graph analysis helps us to understand the underlying system. This understanding is vital to improving or modifying the system or rather generally, to making any pertinent decision about the system. Some significant examples of systems studied through graphs are the Web, various social networks, e.g., Facebook and Twitter [41], patterns of scientific collaborations [50], infrastructure networks, e.g., transportation networks, and many forms of biological networks [32]. Though such interaction data is available for several popular systems, there is still a considerable obstacle in obtaining such data for many systems due to security and privacy concerns. Thus, generating synthetic but realistic graphs has received considerable attention [39, 44]. Ideally, these generative models should capture important features of the networks being modeled. As a consequence, in a related line of work, researchers have given attention to understand the important *features* inherent to networks. In particular, efforts are being made to find the distinguishing characteristics of real-world networks. Among questions

asked in this context are as follows: what rules or properties hold for natural graphs? How can we contrast natural graphs with random graphs? To answer these (and similar) questions, researchers have focused on finding metric or properties that occur regularly in natural graphs. The prominent ones found in the literature include power-law degree distribution, small diameter, and community structures. Now, though the notion of community structure is quite intuitive, there is no consensus of how to define and formalize it. This provides an open avenue for researchers to explain the implicit communities in natural networks through a computationally efficient explicit measure (metric) or phenomenon. The work in this chapter aims at understanding real-world social and information networks through the implicit notion of communities based on common neighbors of a pair of nodes. The main results of this chapter are outlined as follows.

A threshold phenomenon. A popular sociological belief is that people having common friends tend to become friends themselves [47]. The more common friends a pair of people have, the greater chance it creates for those two people becoming friends. However, there is no quantifiable analysis in this regard. Specifically, we do not know how many common friends suffice to generate a high likelihood for those two people to become friends. To pose the question in a graph setting, how much does the number of common neighbors tell about forming edges between two nodes? Based on the Jaccard indices of edges, we observe that there is an interesting threshold behavior of two nodes connecting by an edge for the social and information networks we examined. We introduce the *Jaccard transition curves* that capture this threshold phenomenon. Above this threshold the chance of two nodes being connected by an edge rises sharply.

Contrasting bi-partitions. We show that based on the threshold of edge strength, a network can be partitioned into two subgraphs that show contrasting behavior in terms of network model and construction. One subgraph is induced by the edges with Jaccard indices larger than a threshold, e.g., 0.1, whereas the other subgraph is induced by the remaining edges. We observe that the maximum degree in the subgraph with high Jaccard edges are bounded by a small number (≈ 100). This hints that a dense part of the graph is contained by the edges with high Jaccard indices.

Common neighbor statistics and communities. We demonstrate how common neighbor statistics represented by Jaccard indices can differentiate real-world social networks from random networks. We observe that networks with social (community) structure demonstrate a distinguishing pattern in the Jaccard transition curves. We show how common neighbor statistics relate to community structure of networks.

Characterizing networks based on Jaccard statistics. Since different networks show different patterns in Jaccard transition curves, we investigate the following question: can Jaccard transition curves reveal any global features of networks? Or, can we characterize a network based on Jaccard transition curves? Based on a popular classification method ($C4.5$ algorithm) in data mining literature, we show that we can successfully classify networks into categories such as collaboration, Facebook, and autonomous system networks. Further, using regression analysis, we also predict community sizes of networks from Jaccard statistics with reasonable accuracies.

Table 7.1: Datasets used in our experiments.

Network	Nodes	Edges	Source
ca-AstroPhysics	18772	198K	SNAP [69]
Amazon CP	> 200K	> 1M	SNAP [69]
Oregon AS	10K	22K	SNAP [69]
Anonymous FB	> 10K	> 200K	networkrepository.com
Email-Enron	37K	0.36M	SNAP [69]
web-BerkStan	0.69M	13M	SNAP [69]
LiveJournal	4.8M	86M	SNAP [69]
Twitter	42M	2.4B	[41]
Gnp(n, d)	n	$\frac{1}{2}nd$	Erdős-Réyni
PA(n, d)	n	$\frac{1}{2}nd$	Pref. Attachment

7.2 Preliminaries

In this section, we describe the notations used throughout the chapter and the datasets we examined.

Notations. We denote a network by $G(V, E)$, where V and E are the sets of nodes (vertices) and edges, respectively, with $m = |E|$ edges and $n = |V|$ nodes. The adjacency list of node v is denoted by N_v and the degree of node v, $d_v = |N_v|$.

Jaccard similarity coefficients or Jaccard indices quantify the number of common elements of a pair of sets normalized by the number of all distinct elements. It is one of the most widely used similarity metrics. Jaccard index of two sets A and B is defined as,

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{7.1}$$

For our purpose, sets are adjacency lists of nodes. We define,

$$J_{uv} = J(N(u), N(v)) = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}, \tag{7.2}$$

where $N(v)$ is the adjacency list of v. Given a network $G(V, E)$, we compute the Jaccard similarity J_{uv} for all pairs (u, v), $u, v \in V$.

Datasets. We have examined a large number of real-world and artificially generated networks. Table 7.1 provides a subset of networks we used in our experiments.

We experimented on several types of networks: (i) social networks that consist of online social or contact networks, (ii) co-authorship networks in various disciplines, (iii) web-graphs where nodes represent web pages and edges represent hyperlinks, (iv) internet networks, (v) infrastructure networks such as road networks, and (vi) few random or artificially generated networks. Artificial network PA(n, d) is generated using the preferential attachment (PA) model [13] with n nodes and average degree d. Network Gnp(n, d) is generated using the Erdős-Réyni random graph model

```
1: {C_uv: Number of common neighbors of u and v}
2: {J_uv: Jaccard index of the pair (u, v)}
3: for v ∈ V do
4:     for each pair u, w ∈ N_v do
5:         C_uv ← C_uv + 1
6: for v ∈ V do
7:     for each pair u, w ∈ N_v do
8:         J_uv ← C_uv / (d_u + d_v − C_uv)
```

Figure 7.1: Algorithm for computing all-pair Jaccard indices with wedge enumeration. Pairs with a Jaccard index of 0 are omitted.

[17], also known as $G(n, p)$ model, with n nodes and edge probability $p = \frac{d}{n-1}$ so that the expected degree of each node is d. Note that, we consider all our networks undirected.

Table 7.1 also shows the number of nodes and edges in all networks. The sizes of the networks we studied range from about 10,000 nodes up to nearly tens of millions of nodes and from about 20,000 edges up to hundreds of millions of edges. The networks are also of varying sparsity: the average degrees vary from about 10 to several hundreds.

7.3 Computing Jaccard Index and Transition Plots

In this section, we discuss our quantification of the common neighborhood of a pair of nodes in a network. We then introduce the *transition plot* to characterize networks based on such quantification.

7.3.1 Computing Jaccard Index

A naïve approach to compute all-pair Jaccard index in a graph $G(V, E)$ is to enumerate all possible pairs (u, v), and find the number of common neighbors of u and v. There are $\binom{n}{2}$ such pairs. If neighbor lists N_x are sorted, then finding common neighbors of u and v requires $\Theta(d_u + d_v)$ time. Thus this algorithm takes $O(\sum_{i \in V} \sum_{j \in V - \{i\}} (d_i + d_v)) \approx O(n^2 d_{max})$ time.

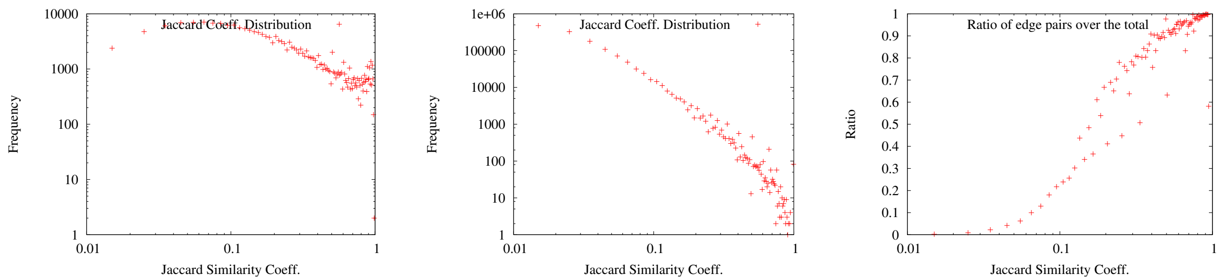
Notice that a pair of nodes (u, v) cannot have a non-zero Jaccard coefficient unless they are the end points of at least one wedge (u, w, v). Thus enumerating all wedges gives us all pairs (u, v) such that $J_{uv} > 0$. Based on this observations, we devise the following algorithm (Figure 7.1) based on wedge enumeration.

7.3.2 Transition Plots

We compute Jaccard indices for all pairs of nodes in the network. Our goal is to understand how Jaccard indices for the edges differ from those of the non-edge pairs.

First, we plot distribution of Jaccard indices J_{uv} of edges $(u, v) \in E$. We divide the range of Jaccard indices (0 to 1) into a number of bins. If we use k bins, then size of each bin i is $1/k$ and it ranges from $(i-1)/k$ to i/k. For each bin i, we count the number of edges E_i having a Jaccard index between $(i-1)/k$ and i/k. We plot a curve with these bins i in x-axis and the number of edges E_i having Jaccard indices in a particular bin i in y-axis.

Second, we plot another distribution of Jaccard indices of J_{uv} of non-edge pairs $(u, v) \notin E$. The plots are constructed in the same way as above. Let, the number of non-edge pairs having a Jaccard index between $(i-1)/k$ and i/k be $\bar{E}_i$.



(a) Jaccard index distribution for edges

(b) Jaccard index distribution for non-edges

(c) Transition curve

Figure 7.2: Transition curve for Jaccard indices for Astrophysics collaboration network.

Figures 7.2a and 7.2b show the Jaccard distribution curves of edges and non-edge pairs, respectively, for Astrophysics network. Many edges have very high Jaccard indices, whereas only a few non-edge pairs have high Jaccard indices. We combine these observations by constructing another plot as follows. For each bin i, we compute the ratio of the number of edges having Jaccard index between $(i-1)/k$ and i/k, to the number of all pairs of nodes (both edges and non-edges) having Jaccard index in the same range. We plot a curve with the ratio in y-axis and the bins of Jaccard indices in x-axis and refer it to as the *Jaccard transition curve*. For a given bin i, the value along y-axis is defined by,

$$y_i = \frac{E_i}{E_i + \bar{E}_i}. \tag{7.3}$$

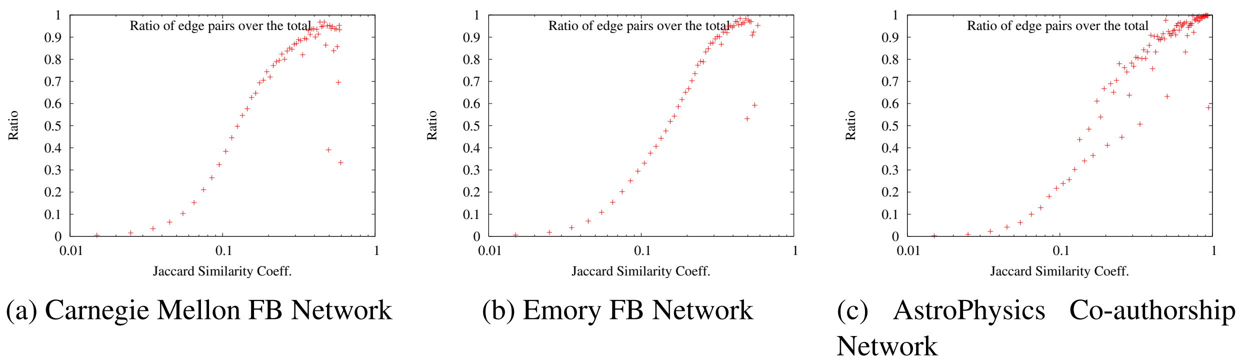
Figure 7.2c shows Jaccard transition curve for Astrophysics network. We observe an interesting transition pattern in the curve. Specifically, the curve shows a sharp rise for the Jaccard indices roughly between 0.1 and 0.2. We will further investigate this pattern in our next subsection for a variety of networks.

Note the following alternative interpretation of the transition curves: let, (x, y) be a point in the transition curve and the Jaccard index J_{uv} of a pair of nodes u, v is x. Then, the probability that

these nodes u, v are connected by an edge is y. Thus, the curve quantifies how much common neighbors contribute to the existence of an edge between a pair of nodes. It would also be interesting to see if different kinds of graphs demonstrate different trends or patterns in transition curves, which we investigate next.

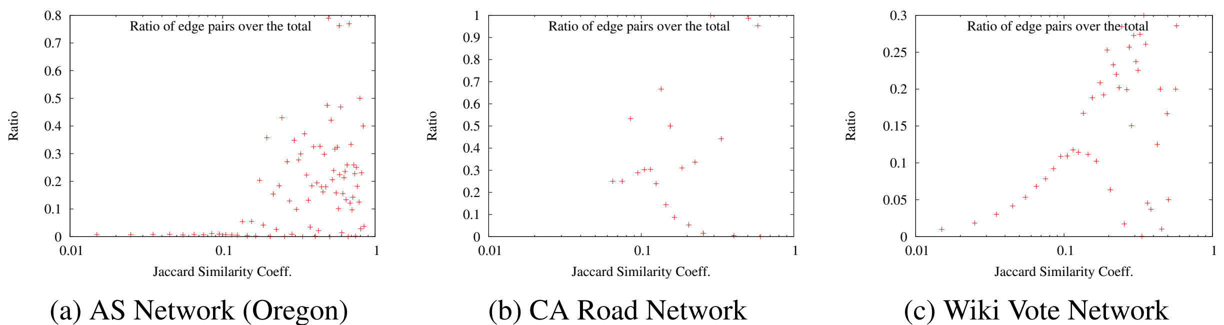
7.3.3 Transition Plots for Variety of Networks

We compute transition plots for different kinds of networks. Figure 7.3 shows the transition curves for social networks (and similar) graphs. Transition plots for some other kinds of graphs (internet, infrastructure and wiki) are shown in Figure 7.4.



(a) Carnegie Mellon FB Network (b) Emory FB Network (c) AstroPhysics Co-authorship Network

Figure 7.3: Transition curve for Jaccard indices on social-network-like graphs.



(a) AS Network (Oregon) (b) CA Road Network (c) Wiki Vote Network

Figure 7.4: Transition curve for Jaccard indices on non social-network-like graphs.

We observe, for social networks (and likes), the transition curves demonstrate an interesting pattern. Non-edge pairs are abundant when bins have Jaccard indices smaller than 0.1. Thus the y-value of the transition curve is very low. However, from Jaccard indices 0.1 to 0.2, there is a very sharp transition to a higher ratio indicating a larger number of edges compared to non-edges. We find this threshold behavior interesting as other non-social networks do not demonstrate similar trends.

An interesting interpretation of this threshold is as follows. When the Jaccard index of a pair of nodes (u, v) crosses the threshold 0.1, they will be connected by an edge with a very high probability. Taking social networks into consideration, when two people have as many common friends as to generate a Jaccard threshold higher than the threshold 0.1, they will more likely be friends themselves.

7.3.4 An Alternative Justification of the Threshold

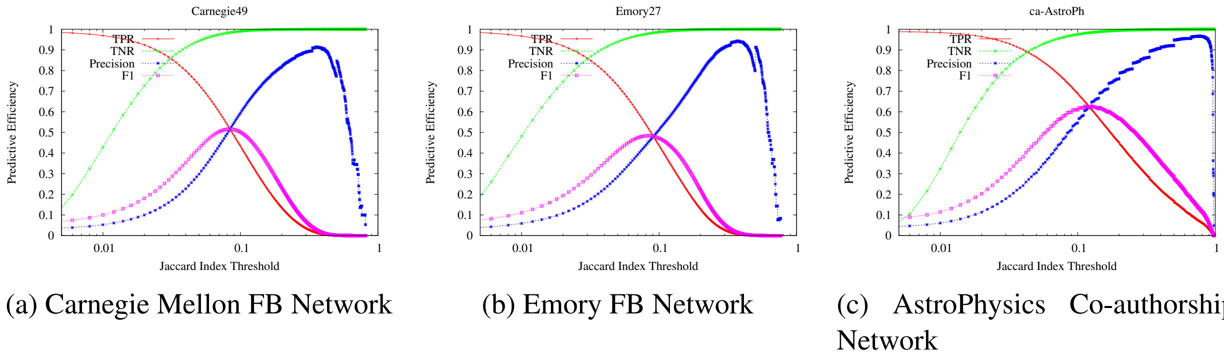
We experiment on the prediction of edges in a network based on Jaccard index of associated pair of nodes. We set a threshold and predict an edge if a pair has a Jaccard index above the threshold. We define the following quantities to assess the performance of prediction based on Jaccard index threshold t ranging from 0 to 1.

- *True Positive:* Number of edges (u, v) with $J_{uv} \geq t$.
- *False Negative:* Number of edges (u, v) with $J_{uv} < t$.
- *True Negative:* Number of non-edges (u, v) with $J_{uv} < t$.
- *False Positives:* Number of non-edges (u, v) with $J_{uv} \geq t$.
- *True Positive Rate (TPR):* Number of true positive over number of edges.
- *True Negative Rate (TNR):* Number of true negative over number of non-edges.
- *Precision:* Number of true positive over sum of number of true and false positive.
- *F1 Score:* Harmonic mean of precision and recall (TPR).

We experiment on a group of networks to see how prediction performance changes by varying the threshold values. We pick the optimum threshold based on the maximum F_1 score. We determine the optimum threshold value based on how it contributes to the accuracy for predicting edges. Figure 7.5 shows the F_1 score for different threshold values. F_1 score is maximum at Jaccard threshold value 0.1. Further, as Tables 7.2 and 7.3 show, we can predict edges with maximum accuracy when we set the Jaccard threshold to 0.1. These experiments provide another justification of our previous observation of a sharp rise of Jaccard transition curve at Jaccard index of 0.1.

7.4 Other Implications of Threshold Behavior

We describe some useful implications of the threshold behavior of transition curves.



(a) Carnegie Mellon FB Network (b) Emory FB Network (c) AstroPhysics Co-authorship Network

Figure 7.5: Change of the prediction performance in terms of F_1 scores by varying the threshold of Jaccard indices on networks with social structures.

Table 7.2: Jaccard indices that achieve the maximum F_1 scores for several Facebook networks.

Networks	$argmax_j\{F_1\}$	Max F_1
Brown	0.072	0.4756
Caltech	0.138	0.5562
Carnegie	0.082	0.5157
Emory	0.082	0.4838
Michigan	0.084	0.4909
AVG, J_{tr}=	0.0916	

7.4.1 Contrasting Bi-partitions

Based on the threshold of edge strength, a network can be partitioned into two partitions consisting of strong edges and weak edges, respectively. We observe degree distribution, number of triangles, among others, in each partition. As shown in Figure 7.7, a subgraph induced by strong edges has a smaller maximum degree (~ 100). This hints that strong edges might form dense subgraphs. We will further explore this hypothesis while exploring the relationship of Jaccard threshold and community structure.

7.4.2 Random Network Models and the Threshold Behavior

We experimented with Erdos-Reyni, Chung-Lu, and BTER [39] graphs. Clearly, Erdos-Reyni and Chung-Lu graphs do not demonstrate Jaccard threshold behavior. On the other hand, BTER graphs show the threshold to some extent (even though not in a clear pattern). Figures 7.8, 7.9, and 7.10 show related plots.

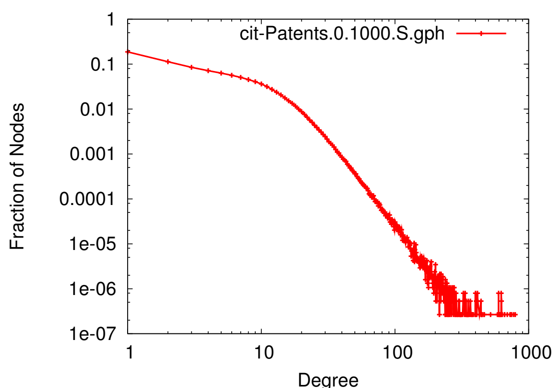
We next try to generate some random networks (with n nodes and m edges) where information about common neighbors are taken into consideration while generating new edges. This is done as follows:

Table 7.3: Accuracies for predicting edges based on the optimum Jaccard index J_{tr} achieved from the training data in Table 7.2.

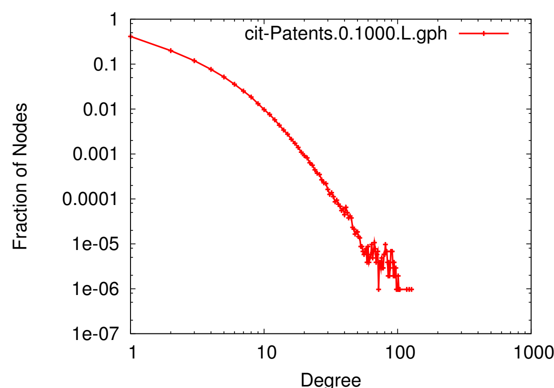
Networks	$argmax_j\{F_1\}$	Max F_1	$F_1(J_{tr})$	Accuracy
Reed	0.112	0.475115	0.449163	0.945377435
Rice	0.092	0.527669	0.527669	1

Table 7.4: Comparison of m, the number of triangles $\triangle$, maximum degree d_{max}, and average degree d_{avg} in the network induced by weak edges $G_{<t=0.1}$ and the Chung-Lu network G_{cl} constructed with the same degree distribution as $G_{<t=0.1}$. The weak edges are the edges with Jaccard indices < 0.1.

Networks	**m**		$\triangle$		d_{max}		d_{avg}	
	$G_{<t}$	G_{cl}	$G_{<t}$	G_{cl}	$G_{<t}$	G_{cl}	$G_{<t}$	G_{cl}
amazon0312	979707	978819	147193	11585	2747	2763	5.71	6.39
amazon0505	1011268	1011160	170718	11857	2760	2744	5.75	6.43
amazon0601	1003649	1003538	167652	11835	2752	2737	5.83	6.49
cit-Patents	14929679	14926796	1595110	1281	793	835	7.94	8.73
roadNet-CA	2417605	2419392	0	3	10	18	2.50	2.90
soc-Epinions1	364544	364729	852009	763233	3036	2700	9.75	12.39
web-BerkStan	4428553	4083915	2489940	32569300	84224	52673	13.81	13.9
web-Google	2056738	2050946	585260	1586210	6325	5898	5.25	6.26
web-NotreDame	657631	649799	120876	850256	10706	8679	4.10	5.23
web-Stanford	1277475	1175510	319074	5647400	38622	24294	9.64	9.87
wiki-Talk	4648277	4461047	8806200	82114200	100029	55546	3.88	5.28
wiki-Vote	82184	82155	98771	187537	942	832	23.10	26.76



(a) cit-Patents network with weak edges

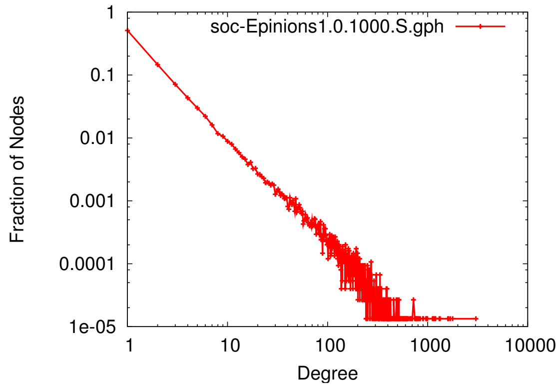


(b) cit-Patents network with strong edges

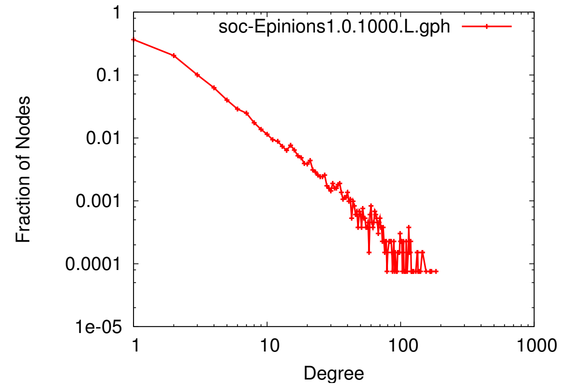
Figure 7.6: Degree distribution of two contrasting partitions– partitions with weak and strong edges, respectively, with strength determined by Jaccard index threshold=0.1.

Table 7.5: Comparison of m, the number of triangles $\triangle$, maximum degree d_{max}, and average degree d_{avg} in the network $G_{<t=0.1}$ induced by weak edges and the network $G_{>t=0.1}$ induced by strong edges. The weak and strong edges are determined based on the Jaccard index < 0.1.

Networks	**m**		$\triangle$		d_{max}		d_{avg}	
	$G_{<t}$	$G_{>t}$	$G_{<t}$	$G_{>t}$	$G_{<t}$	$G_{>t}$	$G_{<t}$	$G_{>t}$
amazon0312	979707	1370162	147193	2557090	2747	56	5.71	7.9
amazon0505	1011268	1428169	170718	2717710	2760	56	5.75	8.07
amazon0601	1003649	1439759	167652	2744450	2752	55	5.83	8.09
cit-Patents	14929679	1589268	1595110	1250540	793	127	7.94	3.08
roadNet-CA	2417605	349002	0	120535	10	7	2.50	2.19
soc-Epinions1	364544	41196	852009	206864	3036	184	9.75	6.23
web-BerkStan	4428553	2220917	2489940	15148900	84224	444	13.81	9.00
web-Google	2056738	2265313	585260	7.8888e+06	6325	158	5.25	8.54
web-NotreDame	657631	432477	120876	7.12771e+06	10706	154	4.10	9.24
web-Stanford	1277475	715161	319074	2.73476e+06	38622	418	9.64	6.81
wiki-Talk	4648277	11288	8806200	7677	100029	95	3.88	1.68
wiki-Vote	82184	18578	98771	104701	942	173	23.10	20.62



(a) soc-Epinions network with weak edges



(b) soc-Epinions network with strong edges

Figure 7.7: Degree distribution of two contrasting partitions– partitions with weak and strong edges, respectively, with strength determined by Jaccard index threshold=0.1.

1. Pick two nodes u and v, randomly.
2. If there is no edge between u and v, compute the number of common neighbors (k) between them.
   - If $k = 0$, add an edge (u, v) with a small predefined probability p_0.
   - Else if $k > 0$, add an edge (u, v) with a probability $p(k)$.

   Repeat until all m edges are added.

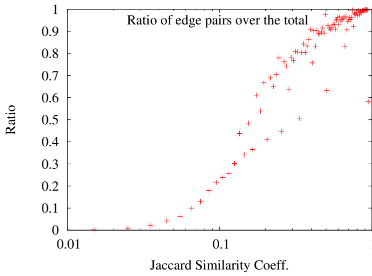


Figure 7.8: Jaccard transition curve of AstroPhysics Network.

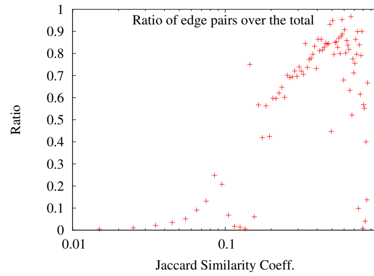


Figure 7.9: Jaccard transition curve of the BTER graph constructed from the same degree distribution and degree-wise CC of AstroPhysics Network.

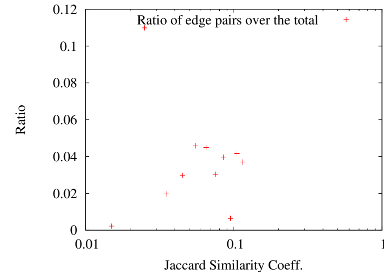


Figure 7.10: Jaccard transition curve of ER graph Gnp(1k, 10k).

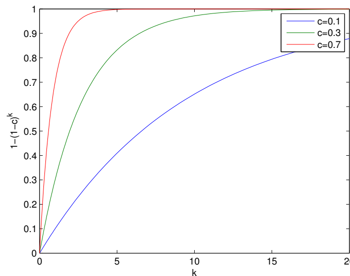


Figure 7.11: Edge probability $p(k) = 1 - (1 - c)^k$ with varying c.

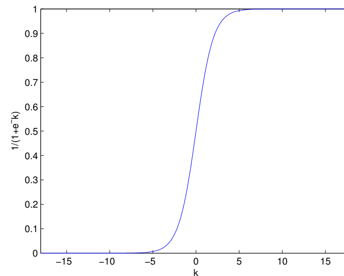


Figure 7.12: Edge probability $p(k) = 1/(1 + e^{-k})$, a sigmoid function.

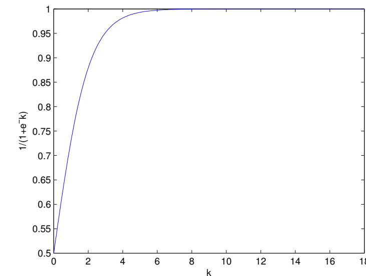
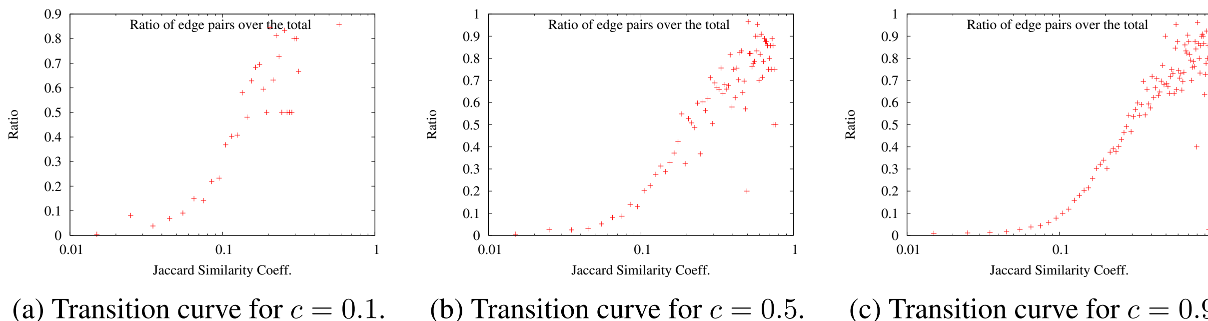


Figure 7.13: Edge probability $p(k) = 1/(1+e^{-k})$ for positive k.

We considered several functions for $p(k)$ (Figures 7.11, 7.12). First, taking $p(k) = 1 - (1 - c)^k$ with varying c (average clustering coefficient of the network), we generate some random graphs and compute Jaccard transition curves (Figure 7.14). These curves look similar (to a large extent) to what we observed with social networks. However, the transition at 0.1 is not as sharp as with the social networks. Further, the degree distribution of the generated networks is, of course, poisson, and the clustering coefficients (CC) are somewhat low. However, the CC values increase with the size of the network (see Figure 7.15, 7.16, 7.17).

We next consider another function for $p(k)$, namely a sigmoid function, in the form $p(k) = 1/(1 + e^{-}k)$. Transition curves of the generated networks demonstrate a transition starting from 0.1 but in a gradual manner (Figure 7.18). The CC of generated networks is low as well, similar to the previous random networks (Figure 7.19). Thus, models using common neighbor information somewhat shows the transition without any sharp threshold. This leads us to the hypothesis that the community structures in real social networks may be contributing to such a threshold. We examine this hypothesis in the following section.



(a) Transition curve for $c = 0.1$. (b) Transition curve for $c = 0.5$. (c) Transition curve for $c = 0.9$.

Figure 7.14: Jaccard transition curves for networks with 1000 nodes and 10000 edges generated with $p(k) = 1 - (1 - c)^k$ and varying c, where c is the input average clustering coefficient (CC-in).

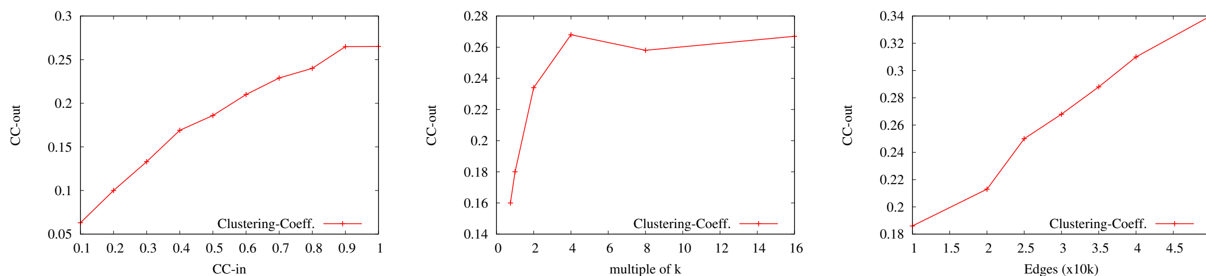


Figure 7.15: Average CC of the generated networks (CC-out) as compared to the input value (CC-in) of c in the function $1 - (1 - c)^k$.

Figure 7.16: Average CC-out in the generated network with varying the multiple a in $p(k) = 1 - (1 - c)^{ak}$ and CC-in=0.5.

Figure 7.17: Average CC-out with varying number of edges in the generated network with $p(k) = 1 - (1 - c)^k$ and CC-in=0.5. Larger graph with same setting has larger average CC.

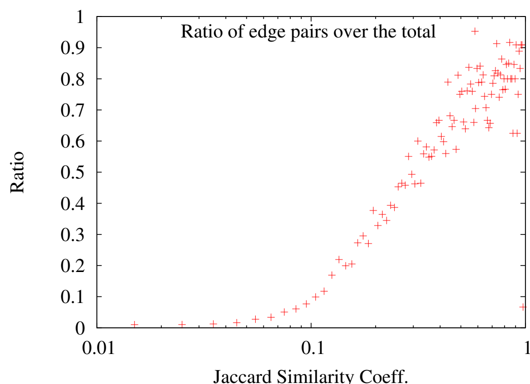


Figure 7.18: Jaccard transition curve for the network generated with $P(k) = 1/(1 + e^{-4k})$ (sigmoid function).

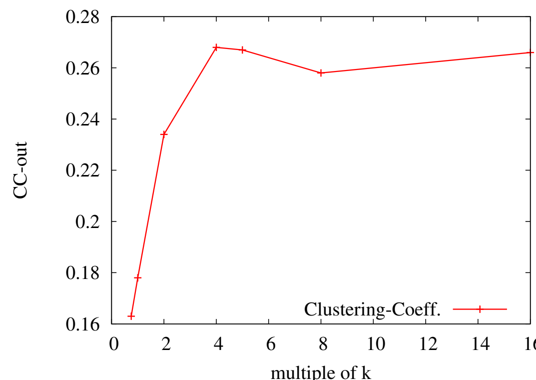


Figure 7.19: Average CC-out with varying the constant a in the sigmoid function $P(k) = 1/(1 + e^{-ak})$.

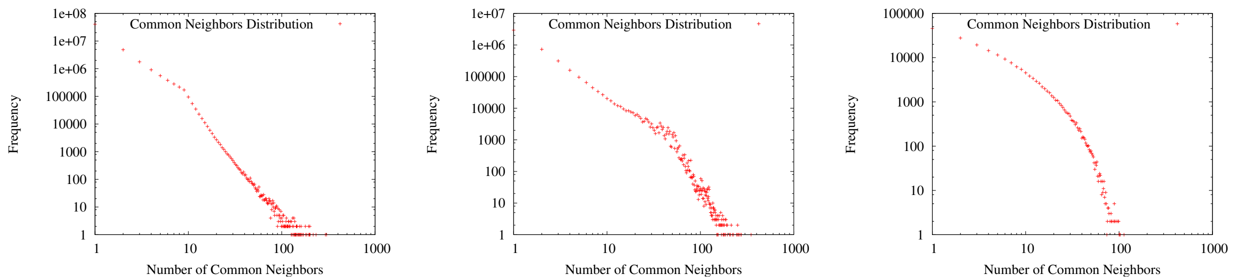
7.5 Common Neighbors and Communities

In this section, we explore the relation between common neighbor statistics (represented with Jaccard transition curves) and community structure in networks.

7.5.1 Common Neighbor Distribution in Networks

We first plot common neighbor distribution (wedge distribution for edges) for both types of networks: networks with and without known community structures. Figure 7.20 and 7.21 show these distributions for such networks. Networks with a community structure show a distinct pattern in their wedge distributions (Figure 7.20). It seems to be a power-law distribution. A network with a partial community structure (Figure 7.21, a) shows hints of such pattern, even though there exist many scattered outlier points. On the other hand, Gnp and road networks do not have a community structure, and wedge distributions for them do not show any pattern (Figure 7.21, b and c). In fact, these two networks do not have a high number of common neighbors for a pair of nodes.

Now, the above plots demonstrate the difference in wedge distributions while there is a difference in community structure among networks. We are further interested in the opposite direction: whether the difference in wedge distribution (or common neighbor statistics) can tell anything about community structure.



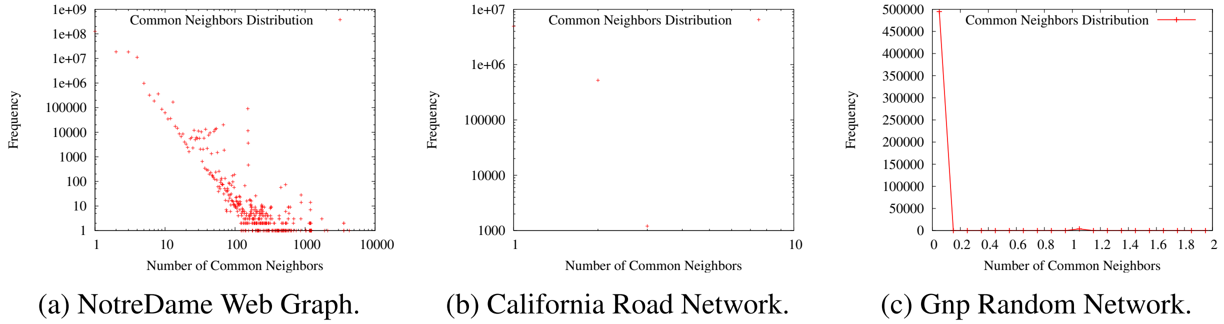
(a) Amazon Copurchase Network. (b) AstroPhysics Coauthorship Network. (c) Facebook Network (Caltech).

Figure 7.20: Wedge distribution (equivalently, common neighbors distribution) curves for networks with communities.

We start with deriving a simple relationship of community size, global CC (transitivity), and degree of the networks. Thereon, we attempt to find a relationship of community and common neighbor statistics.

7.5.2 Clustering Coefficients, Community Size and Degree Distribution

First we make a simplifying assumption based on the work of Rishi *et al.* [37] as follows.



(a) NotreDame Web Graph. (b) California Road Network. (c) Gnp Random Network.

Figure 7.21: Wedge distribution curves for a network with partial community structure (in a) and for networks without communities (in b and c).

Assumption. In triangle-dense graphs (as are social networks), a significant portion of the network is contained in unions of cliques. If we take the edges with Jaccard index above the threshold, we can still retain a significant number of strong edges and triangles. Thus we will be able to extract communities from the network. Relaxing a little from the original notion of clique in the paper [37], let us assume the networks of interest consist of overlapping cliques, which we call communities. It is easy to see that we can convert such networks to bipartite networks such that in one set there are nodes denoting communities and in the other set, nodes are the constituents nodes of the original network. There are links between two sets based on community membership. Obviously, in the single-mode projection, nodes belonging to the same community form a clique based on our assumption.

An analysis based on generating functions. To perform our intended analysis, we adopt the strategy of the work by Newman *et al.* [51] using generating functions [79].

Let, C be the number of communities, N number of nodes, μ average number of communities a node belongs to, and ν average number of nodes per community.

Now, let p_j be the probability that a node belongs to j communities. Alternatively, it is the degree distribution of nodes in the second set in the bipartite graph. Also, assume q_k is the probability that a community has size k.

The above distributions can be generated using the following generating functions, respectively.

$$f_0(x) = \sum_j p_j x^j \tag{7.4}$$

$$g_0(x) = \sum_k q_k x^k \tag{7.5}$$

By using the analysis of bipartite graphs by Newman *et al.* [51], we get,

$$f_0(1) = g_0(1) = 1 \tag{7.6}$$

$$f_0'(1) = \mu \tag{7.7}$$

$$g_0'(1) = \nu \tag{7.8}$$

Further, if we choose a random edge on the bipartite graph, then the distributions of the number of edges leaving two end nodes are generated by the following equations, respectively.

$$f_1(x) = \frac{1}{\mu} f_0'(x) \tag{7.9}$$

$$g_1(x) = \frac{1}{\nu} g_0'(x) \tag{7.10}$$

Then, the distribution of the numbers of co-inhabitants (in the same community) of a randomly chosen node in the second set is generated by,

$$G_0(x) = f_0(g_1(x)) \tag{7.11}$$

Denoting the transitivity (also known as average clustering coefficient or global CC) of the original network, i.e., one-mode projection of the bipartite network, by T, we get the following equation (similar to eqn. 81 in [51]),

$$T = \frac{C}{N} \cdot \frac{g_0'''(1)}{G_0''(1)}. \tag{7.12}$$

This establishes a relation among transitivity (global CC), community size distribution, and degree distribution of triangle-dense networks. We understand this equation is rather generic and any particular distribution for p_j and q_k can be plugged in. Further, note that, transitivity is a global measure rather than being local to nodes or edges. Our original interest was to see the relationship between common neighbor statistics and community size distribution. The above equation (eqn. 7.12) considers the average effect of wedges expressed within the measure *transitivity* (which is the ratio of 3 times the number of total triangles to the number of total wedges). However, by a careful observation we find the following implication.

Implication of Eqn. 7.12. In the equation, the denominator of the right hand side is related to the degree distribution of the original network, and N is the number of nodes. These can be fixed for a variety of networks (real, Chung Lu, etc.). Now the numerator C and $g_0'''(1)$ are related to the community structure of the network. For a network with well-structured communities, the numerator yields a higher value and transitivity is proportional to this community structure. By 'well-structured communities', we mean the presence of many large cliques. The third derivative $g_0'''(1)$ nullifies the effect of any cliques of size 1 or 2 (isolated nodes and edges), considers at least triangles, and favors large cliques. This is consistent with our notion of communities described by cliques.

Now, notice the implicit relationship between transitivity and the Jaccard transition phenomenon. Since the ratio in the y-axis of the transition curve is the number of edge pairs to the number of all pairs having a particular Jaccard index, it gives a sense of wedge closure. A sharp transition indicates the number of wedge closure increases rapidly (sharply), which corresponds to a high transitivity. The transition, beginning at an early stage (at 0.1), even favors the argument for a higher number of wedge closures. Thus, in conjunction with Eqn. 7.12, we conclude that, networks demonstrating sharp Jaccard transition hints at well-structured communities in the networks.

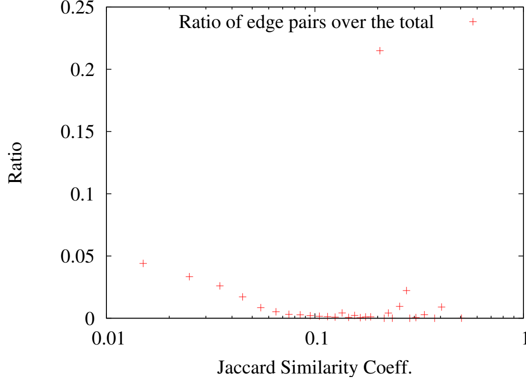


Figure 7.22: Jaccard transition curve for the CL network generated from the degree distribution of AstroPhysics network.

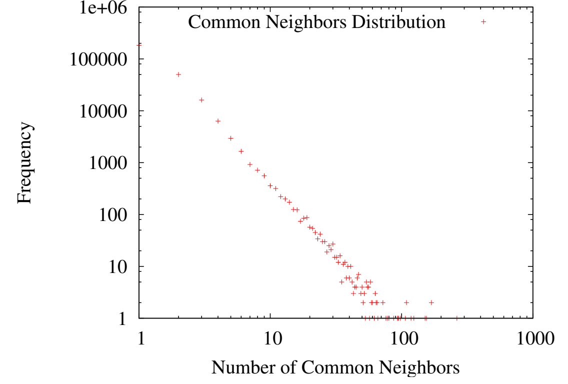


Figure 7.23: Wedge distribution for the CL network generated from the degree distribution of AstroPhysics network..

7.6 Characterizing Networks Based on Jaccard Statistics

We use Jaccard transition curves to characterize networks. Since we observe different patterns of Jaccard transition curves for different networks, we ask the following question: can we predict any global features of networks from the Jaccard statistics? To answer this question, first, we investigate if we can predict the class of a network, where classes are constructed from the thematic areas these networks emerged or statistics related to the community structure in the networks. Further, we perform regression analysis to predict the community sizes in a network.

7.6.1 Predicting Classes from Jaccard Statistics

Using Jaccard statistics, we classify networks into descriptive categories based on the area of emergence of these networks. We also perform classification based on community statistics in networks. The specifications of our experiments are outlined below.

- **Datasets:** We use 40 networks drawn from different categories such as web graphs, social, collaboration, co-purchase, citation, facebook, road, autonomous sytem, and p2p networks.
- **Training and Test data:** We split the whole dataset into half making sure to split among categories.
- **Classification/Characterization:** We use Weka data mining software. For classification, we use the decision tree algorithm C4.5.
- **Attributes:** Attributes are the y-values r_i on Jaccard transition curves for various x-values i as defined below:

$$r_i = \frac{E_i}{E_i + \bar{E}_i}, \tag{7.13}$$

where i is the value of a bin, E_i is the number of edges with Jaccard indices in bin i, and $\bar{E}_i$ is the number of non-edge pairs of nodes with Jaccard indices in bin i.

- **Classes:** We classify networks based on descriptive categories or community sizes.

Characterizing networks into descriptive categories. We experimented with 10 classes such as Facebook, co-purchase, citation, road, and autonomous system networks. A random classifier would have an accuracy of 10%. We observe a classification accuracy of 60% to 85%. Jaccard transition curves for a particular category show a good degree of resemblance. Thus, such curves can predict the category with a good accuracy.

Characterizing networks into categories created by the largest community. We assign classes according to the largest community each network forms as shown in Table 7.6. We use the CNM algorithm for community detection. We found the accuracy of predicting classes is not satisfactory (below 40%).

Table 7.6: Class assignments according to the largest community in the networks.

Class	Largest Community Size
A	0-2000
B	2000-3000
C	3000-4000
D	4000-5000
E	5000-

We also performed the above experiment with the average community size of networks. We assigned a similar classification as given in Table 7.6. The accuracy is again not satisfactory ($35\% - 55\%$).

Table 7.7: Class assignments according to the modularity values obtained for the networks.

Class	Modularity
A	0-0.2
B	0.2-0.35
C	0.35-0.50
D	0.50-0.65
E	0.65-

Characterizing networks into categories created by modularity value. We assign classes according to the modularity values achieved by the community detection algorithm on a network. The classification is shown in Table 7.7. We use the CNM algorithm for community detection. We found the accuracy is quite good ($60\% - 90\%$).

We want to predict community sizes of networks from Jaccard transition curves. The classification method has not proven effective for this purpose. One reason might be community size does not have any natural range with which we can separate categories. Instead of discretize such values, we can treat them as continuous values and apply regression analysis. Next we perform regression analysis on community sizes and Jaccard statistics.

7.6.2 Regression Analysis on Community Sizes and Jaccard Statistics

To predict community size of networks from Jaccard transition curve, we perform multiple linear regression. The *independent variables* are the y-values on Jaccard transition curves for various x-values. The dependent variable in the community size of networks.

For this analysis, we start with networks generated with the Lancichinetti-Fortunato-Radicchi (LFR) benchmark, which gives us networks with controllable community sizes. The LFR benchmark is an algorithm that generates artificial networks that resemble real-world networks. They have a priori known communities and are usually used to compare different community detection methods. One particular advantage of this benchmark is that it accounts for the heterogeneity in the distributions of node degrees and of community sizes. A satisfactory regression model with LFR networks hints the same for real-world networks.

Experiment with LFR networks. We generate sets of networks using LFR benchmark graph generator with various community sizes, average degrees, and numbers of nodes. We then compute Jaccard transition curves for those networks. The transition curve is specified using a number of values on the curve (equal to the number of bins used for the curve). We use multiple linear regression model in the following form:

$$y = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + \cdots + b_nx_n, \tag{7.14}$$

where y is the variable denoting community size and the variables x_i denote values on the Jaccard transition curve. We fit the regression model using standard least square fit from our data. The summary of results is as follows.

- Only the first few values of Jaccard bins can successfully construct the model. These values correspond the y-values on Jaccard transition curve for Jaccard values roughly between 0.05 and 0.25 (the sharp transition region).
- The regression diagnostic plots in Figure 7.24 fairly validate the fits. The 'predicted vs actual' plots demonstrate that the predictions are very close to the actual values. The 'residual by predicted' plot justifies the assumptions of multiple linear regression: no non-linear pattern is evident, values are nicely (evenly) spaced around the zero line, and no significant outliers are observed.
- We experimented for average and maximum community sizes. In both cases, the above observations hold.
- *Mixing parameter versus accuracy*: With LFR networks, the regression accuracy varies with mixing parameter. Our regression model demonstrates robustness to a wide range of values for mixing parameters. As shown in Figure 7.25, the accuracies are good (> 0.87) up to mixing parameter 0.6, and then it breaks down to lower accuracies.

Experiments with Real Networks. We perform the similar experiment as above with ~ 50 real-world networks. The summary of results of regression analysis on those networks is presented below.

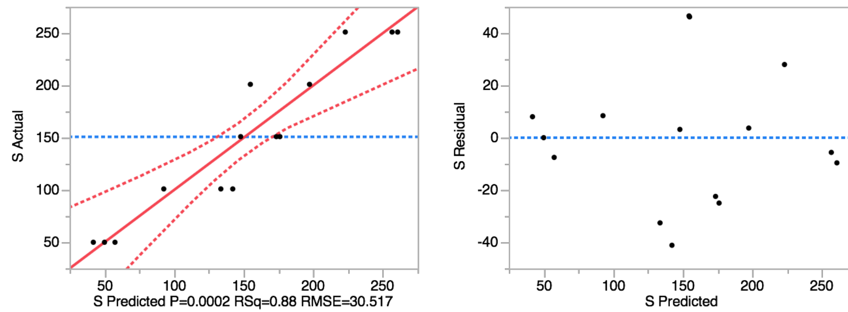


Figure 7.24: The predicted versus actual plot (left) and the residual by predicted plot (right) of the regression analysis on a set of LFR networks. These networks have 10000 nodes, an average degree of 40, community sizes varying from 50 to 500, and mixing parameter 0.2.

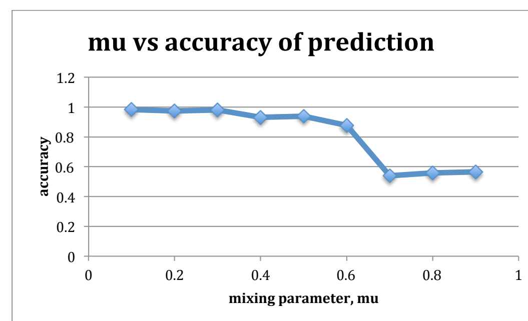


Figure 7.25: Mixing parameter versus the accuracy with our regression model with LFR networks..

- Since in real-world networks, there might be many community of small sizes, the average or minimum size of community might not be significant. Instead, we use percentile measures of community sizes for this experiment. We order the vertices according to sizes of the communities they belong to, and then tried to predict the size of the community of the pn-th ranked vertex (for $p * 100$ percentile and n nodes).

- We compute the accuracy (fitness of model) of regression model for various percentile of community sizes with the networks. For 10, 30, 50, 70, and 90-th percentile, and the maximum, the accuracies are 0.59, 0.73, 0.699, 0.65, 0.62, and 0.71, respectively. These values along with the randomness (absence of any non-linear pattern) of residual plots hint a good model. Figure 7.26 shows the regression plots for the regression analysis with 90-th percentile of community sizes.

7.7 Conclusion

We present a characterization of networks by quantifying the number of common neighbors and demonstrate its relationship with other network properties. We show how much the number of common neighbors contributes to the existence of an edge between two nodes. Based on the Jaccard

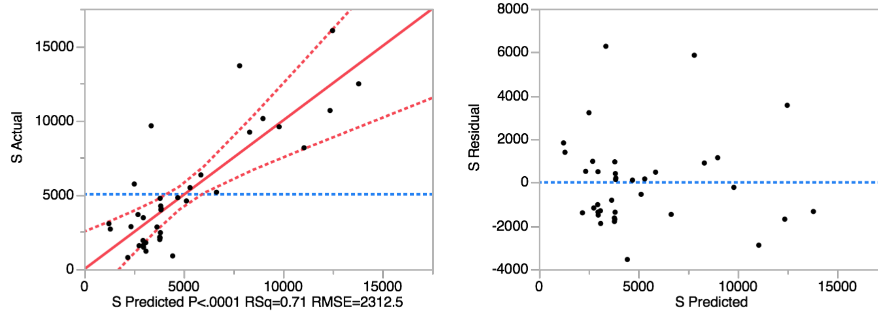


Figure 7.26: Regression diagnostic plots for our analysis on real-world networks: the predicted versus actual plot (left) and the residual by predicted plot (right).

indices of edges, we observe that there is an interesting threshold behavior of two nodes connecting by an edge in the social and information networks. We also demonstrate how common neighbor statistics relate to community structure of networks. We predict the class of a network from Jaccard statistics, where classes are formed from the thematic areas of emergence of networks. With regression analysis, we predict the community sizes of networks from Jaccard statistics with good accuracies.

Part III

Community Detection in Big Networks

Chapter 8

PASCL: Parallel Algorithms for Scalable Community Detection in Large Networks

Unraveling the clusters or communities in large networks (graphs) is an important problem in many scientific areas. Many algorithms have been proposed so far with varying computational complexity and efficiency. With the emergence of big data, the scale of real-world networks, often with millions of nodes and billions of edges and even beyond, poses challenges to their efficient analysis. Existing algorithms might require a large runtime, and a single main memory may fail to fit the network data. To address these issues, distributed network processing has become popular in recent years. In this chapter, we design MPI-based parallel algorithms for detecting communities in large networks. Although these algorithms are based on efficient sequential methods in the literature, parallelization of them for distributed-memory systems poses non-trivial challenges. We propose efficient load balancing and communication approaches to address those issues. Our parallel algorithms work on large networks and scale to a large number of processors. Further, we also combine variations of several known methods by an hybrid approach to compare speed and quality of the detection. Finally, we also demonstrate how our parallel algorithms can be adapted to come up with even faster computations by incorporating edge sparsification techniques.

8.1 Introduction

A network is a powerful abstraction for representing a complex system where the elementary parts of the system and their interactions are represented as nodes and links (edges), respectively. Complex systems are organized in clusters or communities, each having a distinct role or function. In the corresponding network representation, each functional unit (community) appears as a dense set of nodes having higher connection inside the set than outside. Finding communities may reveal the organization of complex systems and their function. For instance, a community is often interpreted as an organizational unit in social networks, a functional unit in biological networks, or a scientific discipline in citation networks [46]. Thus, detecting communities (clusters) in massive networks such as emerging social and information networks has become an interesting and fundamental problem in network science.

8.1.1 Background of Community Detection

The problem of community detection has a rich history and numerous methods exist for solving this problem [25, 32, 36, 58, 59].

Girvan et al. [32] proposed a hierarchical divisive algorithm that removes edges iteratively based on the betweenness centrality of edges. The authors proposed a measure, modularity, for assessing the quality of detected communities, which compares between the graph at hand and a null model, which is a class of random graphs with the same expected degree sequence of the original graph. The algorithm has a computational complexity $O(n^3)$ (n and m are the number of nodes and edges in the network, respectively). Since then, several other methods have been proposed to improve the complexity and quality of the detected communities. Clauset et al. [25] provides an $O(n \log^2 n)$ algorithm that starts from isolated nodes as the initial communities and then iteratively adds nodes to produce higher modularity.

A group of other works aims at exhaustive optimization of modularity. For example, [36] does so by applying the technique of simulated annealing. Blondel et al. [16] proposes a multi-step local optimization of modularity in the neighborhood of each node. This method provides an approximate optimization of modularity. Each identified partition is assimilated into a supernode yielding a smaller network and the process is iterated until modularity does not increase any more. This method offers a trade off between the quality of communities and the computational complexity which is essentially linear in the number of edges of the network. The method proposed by Radicchi et al. [58] computes edge clustering coefficients instead of betweenness values as given in [32] but still has a high complexity of $O(n^2)$.

An algorithm using a random walk on a network is proposed in [63] by Rosvall et al. Their method converts the problem of finding the best communities into a problem of optimally compressing the information of a dynamic process (random walk) taking place on the network. The optimal compression is obtained by optimizing a quality function (minimum description length of the random walk) that implicitly finds communities of good quality. Another fast method for community detection is proposed in [62] by Ronhovde et al. They use a Potts model to evaluate the hierarchical or multiresolution structure of a graph. The algorithm calculates correlations among multiple copies (replicas) of the same graph over a range of resolutions. Strongly correlated replicas identify significant multiresolution structures. In short, the method is based on the minimization of the Hamiltonian of a Potts-like spin model, where the spin state represents the membership of the node in a given community.

Raghavan et al. [59] presents a near linear time algorithm for community detection based on label propagation. A node takes the community label that is the label of the majority of its neighbors. The algorithm is quite fast but the detected community might be of lower quality (based on some quality measure such as modularity). Further, results can be unstable as different runs of the algorithm might produce different results based on the choice of synchronous or sequential update of labels.

Some other works using adjacency matrix representation of networks and computation of eigenvectors of the Laplacian matrix are given in [29] (Markov Cluster Algorithm) and [28] (spectral algorithm). The algorithms described above vary in terms of quality of detected communities and

the computational complexity of the algorithms. Adjacency matrix based and spectral algorithms cannot work on networks having more than a few hundred thousand of nodes. Some algorithms [32, 58] have a very high computational complexity and cannot work on large networks, whereas a few others [59] are faster at the cost of the quality of the detected communities. You can find two comprehensive surveys of some community detection methods in [31, 46].

8.1.2 Challenges with Massive Networks

In the present world of technological advancement, we are deluged with network data from a wide range of areas such as Web, business and finance, computational biology, and social science. Many social networks have millions to billions of users. The size of emerging networks motivates us to find novel algorithms that are both space and computationally efficient.

In many cases, these massive network do not fit into the main memory of a single computing node. Further, an algorithm for community detection having a high computational complexity might fail to work on networks with a few millions of nodes or edges. In addition to the classic problem of finding communities with 'good' quality, the emergence of massive networks poses additional complicacy.

8.2 Related Work on Parallel Algorithms

Despite the fairly large volume of work addressing this problem, only recently has attention been given to the problems associated with large graphs. In recent years, several parallel algorithms for shared-memory systems and only a few for distributed-memory parallel systems have been proposed [52, 61, 71, 81]. The distributed-memory algorithms were designed for the Bulk Synchronous Parallel (BSP) and MapReduce frameworks.

In [81], Zhang et al. proposed a parallel algorithm that adopts a *Bulk Synchronous Parallel* (BSP) model of computation. The overall computation proceeds in consecutive supersteps. There is a barrier between two successive supersteps. The communities detected by the algorithm are the connected components of the graph after iterations of adding and removing edges based on propinquity measures. The computational complexity of the algorithm is $O(k.(m+n)(m/n)^2)$, where k is the number of iteration the algorithm takes to converge to an acceptable result. The paper provides a clever technique to update propinquity value incrementally. However, the authors did not provide any analytical proof that the algorithm will eventually converge in a small number of steps. Further, there are a number of synchronization steps where processors need to exchange messages to keep their states consistent. The overall messages sent in a single superstep can easily exceed the memory quota. The largest network they processed has $\sim$2.5M nodes ($\sim$100M edges).

Another parallel algorithm for a multi-core and GPU architecture is proposed in [70]. They design a variant of the label propagation technique with a computational complexity of $O(m(k+d))$, where k is the number of iterations and d is the average degree. The largest network processed by the algorithm has 100M edges. The whole input network needs to be in memory to execute this

algorithm. Further, a mathematical model to predict the number of iterations of the algorithm and a tight bound on the quality of the algorithm are not provided in the paper. Another shared-memory parallel algorithm is given in [61]. The algorithm adopts an agglomerative approach merging pairs of connected intermediate subgraphs to optimize different graph properties. The algorithm achieves a moderate parallel scalability.

A MapReduce based distributed preprocessing algorithm for community detection is proposed in [52]. The algorithm identifies nucleuses (core groups) of communities and coarsens the original graph to the graph induced by the core groups' partition. An arbitrary community detection algorithm can be used to identify communities of the coarsened graph. The preprocessing step uses an ensemble of partitions, each created by a label propagation algorithm, and then finds the maximal overlap to get core groups. The algorithm generates multiple intermediate disk files consisting of node-to-node links and core-group-to-core-group links. A network with 3.3B edges is processed in a few hours.

Another shared memory parallel algorithm is given in [71]. They implement parallel variation of some known sequential algorithms and combine them by an ensemble approach to accumulate advantages from all of them. Similar to [52], the largest networks processed by this paper has 3.3B edges.

8.3 Fast and Scalable Parallel Algorithms for Community Detection

We design fast parallel algorithms for detecting community in large networks. We identify that the *Louvain* algorithm [16] is a well-recognized and efficient sequential method mentioned in several other work [31]. We design our MPI-based parallel algorithm for community detection based on the Louvain algorithm. Parallelizing the Louvain algorithm for distributed-memory systems poses non-trivial challenges. We present explicit load balancing schemes and HPC-based optimization techniques to improve the performance of our parallel algorithm. We also design an MPI-based parallel algorithm for the Label Propagation algorithm and demonstrate how we can combine the benefits from both algorithms by an ensemble technique.

8.3.1 Sequential Louvain Algorithm

The Louvain method for community detection was first presented by Blondel et al. [16]. It can be classified as a locally greedy, bottom-up multilevel algorithm and uses modularity as the objective function. Figure 8.1 shows pseudocode for the sequential Louvain algorithm. We call the inner repeat-until loop (Line 5-14) *phase 1*, execution of Line 15-17 *phase 2* of computation, and the outer repeat-until (Line 3-18) loop a *pass* of the algorithm. In each pass, nodes are repeatedly moved to neighboring communities so that the locally maximal increase in modularity is achieved, until the communities are stable (phase 1). Then, the graph is coarsened according to the solution (phase 2) and the procedure continues recursively, forming communities of communities (another

1: **for** each $v \in V$ **do**
2: $C[v] \leftarrow v$ // singleton community
3: **repeat**
4: anychange $\leftarrow false$
5: **repeat**
6: done $\leftarrow true$
7: **for** each $v \in V$ **do**
8: $t \leftarrow \max_{u \in N_v} \triangle mod(v, C[v] \rightarrow C[u])$
9: $c \leftarrow C[\operatorname{argmax}_{u \in N_v} \triangle mod(v, C[v] \rightarrow C[u])]$
10: **if** $t > 0$ **then**
11: $C[v] \leftarrow c$
12: done$\leftarrow false$
13: anychange$\leftarrow true$
14: **until** done
15: **if** anychange **then**
16: $G' \leftarrow$ Contract(G, C)
17: $G \leftarrow G'$
18: **until** *not* anychange

Figure 8.1: Pseudocode of the sequential Louvain algorithm. $C[v]$ is the community label of node v. The quantity $\triangle mod(v, C[v] \rightarrow C[u])$ denotes the difference in modularity when node v is moved from $C[v]$ to a neighboring community $C[u]$.

pass). Finally, the communities in the coarsest graph determine those in the input graph by direct prolongation.

Next, we describe the overview of our parallel Louvain algorithm followed by a detailed description of different steps.

8.3.2 Overview of Our Parallel Algorithm

Let p be the number of processors used in our computation. Our algorithm partitions the input graph $G(V, E)$ as follows: the set of nodes V is partitioned into p disjoint subsets V_i^c, such that, for $0 \leq j, k \leq p-1$ and $j \neq k$, $V_j^c \cap V_k^c = \emptyset$ and $\bigcup_k V_k^c = V$. Edge set E_i^c, constructed as $E_i^c = \{(u, v) : u \in V_i^c, v \in \mathcal{N}_u\}$, constitutes the i-th partition. Processor P_i works on the i-th partition and is responsible for detecting community labels $C[v]$ of all nodes $v \in V_i^c$. Now, to detect $C[v]$ of all $v \in V_i^c$, processor P_i needs $C[u]$ of all $u \in \mathcal{N}_u$ (Lines 8-9, Figure 8.1). If $u \in V_i^c$, information of both $C[v]$ and $C[u]$ is available in the i-th partition. However, if $u \in V_j^c$, $j \neq i$, $C[u]$ resides in partition j. Processors P_i and P_j exchange message(s) for communicating $C[u]$. This exchanging of messages introduces a communication overhead, which is a crucial factor on the performance of the algorithm. Each processor locally executes one iteration of the sequential Louvain algorithm (Lines 7–9, Figure 8.1). Then the processor communicates with other processors for community

labels as discussed above. For parallelizing the phase 2 computation, we require to renumber the community labels obtained from phase 1 into new consecutive labels. This allows consistency in community labeling in different passes and enables the algorithm to reconstruct hierarchical communities of the original input network. However, in a distributed setting, parallelizing this renumbering operation should be done in an efficient way. We will describe this parallelization in Section 8.3.5. The remaining part of phase 2 deals with constructing a supergraph, computing a coarsened graph by merging nodes of the same communities into a supernode. Constructing a supergraph is also nontrivial, which we will describe in detail in Section 8.3.6.

8.3.3 Partitioning

For partitioning the input network $G(V, E)$, the set of nodes V is partitioned into p disjoint subsets V_i^c of consecutive nodes. Ideally, the set V should be partitioned in such a way that the cost for detecting communities of nodes in V_i^c is almost equal for all processors. Let, $f(v)$ be a *cost function* referring to the cost of detecting communities for each node $v \in V$. Similar to our fast parallel algorithm for counting triangles presented in Section 3.3.4, we need to compute p disjoint partitions of V such that for each partition V_i^c,

$$\sum_{v \in V_i^c} f(v) \approx \frac{1}{p} \sum_{v \in V} f(v). \tag{8.1}$$

We consider the following two load balancing schemes based on two different cost functions.

- Scheme $\mathcal{N}$: This scheme estimates cost function as $f(v) = 1$.
- Scheme $\mathcal{D}$: This scheme estimates cost function as $f(v) = d_v$.

The first scheme assumes equal cost for every node whereas the second scheme assumes that the cost depends on the degree of the node.

Given $f(v)$ for all $v \in V$, we compute V_i^c using the same parallel algorithm we used for computing balanced partition as described in Section 3.3.4.

8.3.4 Local Computing of Community Labels

Processor P_i is responsible for detecting community labels $C[v]$ of all nodes $v \in V_i^c$. Each processor locally executes one iteration of the sequential Louvain algorithm (Lines 7–9, Figure 8.1). However, as discussed in the overview of our algorithm, to detect $C[v]$ of all $v \in V_i^c$, processor P_i needs $C[u]$ of all $u \in \mathcal{N}_v$ (Lines 8-9, Figure 8.1). If $u \in V_j^c$, $j \neq i$, $C[u]$ resides in partition j. Processors P_i and P_j exchange message(s) for communicating $C[u]$.

One straightforward way to communicate all such labels is each processor broadcasting all labels. This approach is conceptually simple; however, such broadcasting is computationally expensive. A

better way for P_i is to request other processors for labels $C[u]$ of nodes $u \in \mathcal{N}_v \cap V_j^c$. This approach has a communication complexity of $O(2\ell)$, where ℓ is the number of cut edges. However, we observe that we can improve this approach further by eliminating the request messages altogether. Each processor can directly send community labels of nodes to other processors that might need them. Each processor can easily construct such messages by scanning neighbor lists N_v of nodes $v \in V_i^c$. In fact, P_i does not require any additional scanning; it can construct these message while executing phase 1 computation. Since no request messages are required for this approach, it saves 50% of message cost. Additionally, we bundle all messages sent to a particular processor to further reduce communication overhead.

8.3.5 Renumbering Community Labels

Renumbering is an operation that converts a set of community labels to another set of consecutive labels. Renumbering ensures consistency of graph representation and allows using the same data structure throughout passes. The operation also enables the retrieval of hierarchical community labels of the nodes of the input graph, when several passes of the algorithms are made.

In a sequential setting, an array with $O(n)$ size can be used to perform the renumbering: construct an array $A[.]$ of size n. Initial the array with zero values. If a node has community label i, increment the value of $A[i]$; do the same for all nodes. Now, scan the array, start community labels from zero, incrementally assign new labels to the labels i with nonzero values $A[i]$.

However, in a distributed setting, one way to construct the above array is to use MPI All_gather and reduce operation. This is conceptually simple but has a runtime complexity of $\Omega(n \lg p)$, which is worse than the sequential algorithm. Next, we devise a parallel algorithm for performing the renumbering operation in $O(n/p + \lg p)$ time in the worst case. The main steps of the algorithm are as follows.

- Assume S_i be the set of labels in processor P_i. P_i divides S_i into p (at most) disjoint subsets S_i^j and sends to processor j.
- We use a simple hash-based distribution of S_i into S_i^j.

$$S_i^j = \{x \in S_i : x \mod p = j\}. \tag{8.2}$$

- Processor P_j constructs S^j from all S_i^j received from other processors i.

$$S^j = \bigcup_i S_i^j \tag{8.3}$$

- Each processor P_i renumbers locally for each labels in S^i in consecutive numbers $0, \ldots, n_i - 1$, where n_i be the number of distinct labels in P_i.
- We compute parallel prefix sum using the algorithm by Aluru et al. (2011).
- Each processor adjusts its sequence of new labels by adding $\sum_{t=0}^{t=i-1} n_t$ to each labels.

8.3.6 Constructing Supergraph

The overview of constructing a supergraph from community labels is as follows.

- Contract nodes (member) of a particular community into a *supernode*.
- Compute *edges* between community supernodes based on their member nodes' connectivity.
- Compute *weights* between supernodes based on weights of all cutting edges between member nodes.
- Compute further iteration of Louvain algorithm on the *supergraph*.

In a sequential setting, a supergraph is computed by straightforward iteration over nodes of all communities. However, in a parallel setting, we distribute tasks with the following consideration:

- Reduction of communication cost
- Reusage of local data
- Load balancing in the current computation
- Possibly, providing a convenient initial partition for the next phase.

We devise a parallel scheme to compute the supergraph as follows. Processor P_i constructs a part G'_i of the super graph G' from the local information it has: V_i, community assignments, and neighbor information of all nodes $v \in V_i^c$. G' is computed by the equation $G' = \bigoplus_i G'_i$, where $\bigoplus$ is a *merging* operation that we describe shortly. We perform the operation $\bigoplus$ in parallel (with balanced load).

The overview of the merging operation $\bigoplus$ is given below.

- For a supernode v, let $\bar{N}_v$ be the set of neighbors of v and $\bar{W}_v$ be the set of weights of edges between v and $u \in \bar{N}_v$.
- Processor P_i constructs a part G'_i of the super graph G' from the local information it contains. That is, P_i construct partial (local) sets W_v^i and N_v^i of node v. We define S_i and T_i be the set of all partial sets N_v^i and W_v^i, respectively, constructed by P_i.
- Each processor P_i is responsible for performing the operation $\bigoplus$ for a subset V'_i of super nodes.
- V'_i is computed using our parallel load balancing scheme described in Chapter 3.
- Processor P_i divides S_i and T_i into p (the number of processors) disjoint subsets S_i^j and T_i^j, $0 \leq j \leq p-1$, as defined below.

$$S_i^j = \{N_v^i : v \in V'_j\}, \tag{8.4}$$

$$T_i^j = \{W_v^i : v \in V'_j\}. \tag{8.5}$$

- Processor P_i sends S_i^j and T_i^j to all other processors P_j.
- Once processor P_i gets T_j^i and S_j^i from all processors P_j, it constructs $\bar{N}_v$ and $\bar{W}_v$ for all $v \in V_i'$ by the following equations.

$$\bar{N}_v = \bigcup_{k:N_v^k \in S_k^i} N_v^k \quad (8.6)$$

$$\bar{W}_v = \bigcup_{k:W_v^k \in T_k^i} W_v^k \quad (8.7)$$

While performing the union operations as shown above, if duplicate items exist, the corresponding weights are summed together in $\bar{W}$, and only a single item is kept in $\bar{N}$.

The pseudocode for parallel Louvain algorithm is given in Figure 8.2.

8.4 Label Propagation Algorithm

Raghavan et al. [59] proposed the label propagation algorithm (LPA) for community detection. The advantage of a LPA is its simplicity and ease of parallelization. The high level overview of LPA is as follows: initially each node in the network is assigned a unique label. In each iteration every node updates its label to the label that is the most frequent in its neighborhood; ties are broken randomly. Densely connected set of nodes thus agree on a common community label. Usually after a small number of iteration, a global stable consensus of community labels is reached. Thus LPA has a near linear runtime complexity. At each iteration it requires $O(m)$ time. Further, it has been shown empirically that the algorithm reaches a stable solution in a small number of iteration. This algorithm does not require the computation of an objective function such as modularity. It maximizes any such functions only implicitly. Since the algorithm heavily involves the local update of community labels, it is well suited for parallel implementation. One obtains variants of LPA by varying how the initial label assignment is made, how ties are broken, and whether a node includes itself in computing the most frequent label in its neighborhood.

In this work, we parallelize a specific variation of LPA in which nodes are assigned initial labels the same as the node ID. Further, if there is a tie, it is broken in favor of the larger label. Finally, a node includes its own label in determining the most frequent label in its neighborhood. We also update labels in a synchronous fashion.

Our MPI-based parallel algorithm for label propagation is very similar to the phase 1 computation of the parallel Louvain algorithm. Instead of modularity, each update of community label considers only the labels of the neighbors of a node. We employ a similar partitioning, load balancing, and communication strategy, and therefore these are not repeated here.

```
1:  // Each processor P_i executes the following:
2:  V_i ←ComputeBalancedPartition(G, i)
3:  ReadGraph(G, V_i)
4:  CreateSingletonCommunity(V_i)
5:
6:  // Computation of Phase 1
7:  repeat
8:     anychange ← false
9:     repeat
10:       done ← true
11:       for each v ∈ V_i do
12:          t ← max_{u∈N_v} △mod(v, C[v] → C[u])
13:          c ← C[argmax_{u∈N_v} △mod(v, C[v] → C[u])]
14:          if t > 0 then
15:             C[v] ← c
16:             done← false
17:             anychange← true
18:       BroadcastAssignments(C, i)
19:    until done
20:
21:    // Community assignment at current iteration
22:    ParallelRenumbering(C, i, V_i)
23:    PrintCommunity(V_i, C)
24:    ComputeModularity(C, i, G_i)
25:    if anychange then
26:       ComputeSuperGraph(G, C)
27: until not anychange
```

Figure 8.2: Pseudocode for our parallel Louvain algorithm.

8.5 Evaluation of Our Parallel Algorithms

In this section, we present an experimental evaluation of the performance of our algorithms. We show the scalability and analyze various trade-offs. We use the same datasets and experimental setup as discussed in Chapter 2.

8.5.1 Load Balancing and Scalability

A parallel algorithm is completed when all of the processors complete their tasks. Thus, to reduce the running time of a parallel algorithm, it is desirable that no processor remains idle and all processors complete their executions almost at the same time.

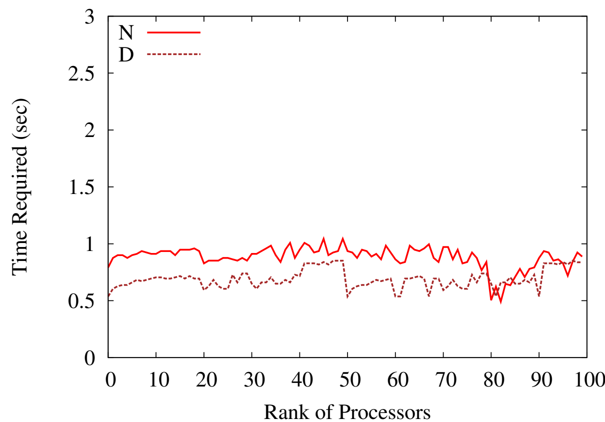


Figure 8.3: Laod distribution for Miami network with equal number of nodes and edges per processors.

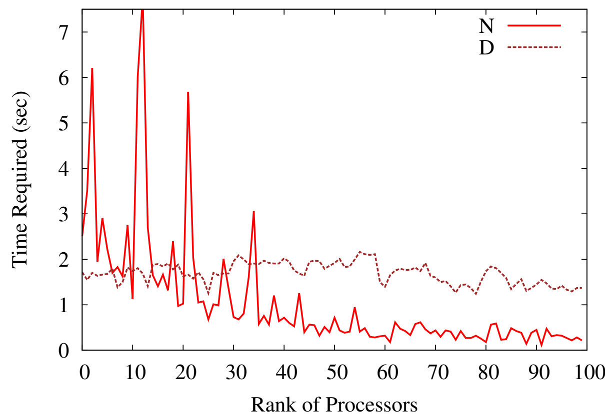


Figure 8.4: Laod distribution for LiveJournal network with equal number of nodes and edges per processors.

Figure 8.3 and 8.4 show load distribution of our parallel Louvain algorithm with Miami and LiveJournal networks. As described before, Miami is a graph with an almost even degree distribution,

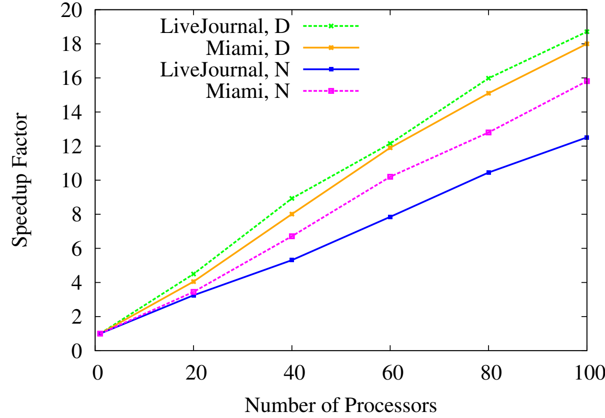


Figure 8.5: Speedups of our parallel Louvain algorithm on Miami and LiveJournal networks.

whereas LiveJournal has skewed degree distribution. Scheme $\mathcal{D}$ has better load distribution with both networks. Since the computational cost of our algorithm due to each node is proportional to its degree, scheme $\mathcal{D}$ provides more precise estimation of computing cost of our algorithm.

Figure 8.5 shows strong scaling (speedup) of our algorithm on Miami and LiveJournal networks with both load balancing schemes. Our algorithm demonstrates good speedups and scales almost linearly. Scheme $\mathcal{D}$ achieves better speedup than $\mathcal{N}$ for the reason discussed above.

8.5.2 Trading off the Quality and Speed of our Community Detection Algorithms

Louvain algorithm is one of the best sequential algorithm in the literature [31]. It has been used in the shared-memory parallel algorithm given in [71]. We presented the first MPI-based parallel algorithm for community detection based on the Louvain algorithm. We also implemented the Label Propagation Algorithm (LPA) with MPI. LPA has a near linear runtime complexity. However, the quality of detected communities depends on the number of iterations. It usually compromises the quality in favor of the speed of execution. Thus the Louvain algorithm and LPA provide a good trade-off between runtime and modularity value.

Table 8.1: Comparison of modularity and runtime between parallel LPA and Louvain Algorithm.

Networks	**LP**		**Louvain**		**Hybrid**	
	Mod.	Runtime	Mod.	Runtime	Mod.	Runtime
Miami	0.354	15.86	0.46	21.51	0.41	18.22
web-BerkStan	0.38	2.02	0.42	2.39	0.39	2.25
LiveJournal	0.42	18.95	0.445	23.76	0.43	21.52

Table 8.1 shows a comparison between our parallel LPA and Louvain algorithm in terms of execution time and modularity. The Louvain algorithm generates communities that are of better quality

than that of LPA. However, for the same number of iterations, the Louvain algorithm takes a larger time. We also design a hybrid algorithm (also referred to as *ensemble algorithm*) combining both parallel LPA and Louvain algorithm: in the first pass of the parallel Louvain algorithm, we update community labels using the update rule of LPA. Instead of each node updating its community labels based on modularity increase, it just takes the most frequent neighboring community. The last 2 columns of Table 8.1 show the modularity and runtime with this hybrid algorithm. This algorithm improves the runtime of the Louvain algorithm and the modularity of LPA and can be useful as a convenient tradeoff between runtime and modularity.

8.5.3 Parallel Sparsification Algorithm

We integrate sparsification techniques with our parallel algorithm. Sparsification of a network is a sampling technique where some randomly chosen edges are retained and the rest are deleted, and then computation is performed in the sparsified network. Sparsification of a network saves both computation time and memory space and provides an approximate result. There might be various criteria for selecting edges to retain. In this experiment, we consider the following three criteria inspired from the work in [64].

- Global sparsification: Each edge is preserved with a probability q. The pseudocode is shown in Figure 8.6.
- Local sparsification: Similar to global sparsification but it made sure each node has at least one edge incident on it. The pseudocode is shown in Figure 8.7.
- Jaccard-based sparsification: All edges with Jaccard index < 0.1 is discarded.

```
1: for v ∈ V_i do
2:    for (v, u) ∈ E do
3:       toss a biased coin with success prob. q
4:       if success then
5:          store u to N_v
```

Figure 8.6: Global sparsification of a network in parallel.

We show the performance of the above sparsification method in Table 8.2. Let ΔQ be the difference in modularity with the original and the sparsified graph. For this experiment, $q = 0.5$ for local and global sparsification. For Jaccard-based sparsification, we discard all edges having Jaccard index < 0.1. Global sparsification loses some quality (in terms of modularity) due to the prospect of disconnecting less dense communities or even isolating nodes with small degrees. Local sparsification performs better than global since no nodes are isolated. Jaccard based sparsification provides the best solution among them since this method discard edges that are less important in terms of community formation.

```
1: for v ∈ V_i do
2:     for (v, u) ∈ E do
3:         toss a biased coin with success prob. q
4:         if success then
5:             store u to N_v
6:     if N_v = ∅ then
7:         pick one u from {(v, u) ∈ E} and store to N_v
```

Figure 8.7: Local sparsification of a network in parallel.

Table 8.2: Modularity and runtime with various sparsification method on different networks.

Networks	**Local**		**Global**		**Jaccard**	
	ΔQ	Runtime	ΔQ	Runtime	ΔQ	Runtime
Miami	0.03	13.54	0.12	13.43	0.03	15.12
web-BerkStan	0.07	1.50	0.14	1.47	0.05	1.61
LiveJournal	0.11	15.87	0.18	15.52	0.02	16.37

8.5.4 Comparison with Other Algorithms

Previous parallel algorithms [52, 61, 70, 71] are based on MapReduce, shared-memory, and BSP framework. Our algorithms are MPI-based distributed memory algorithm. To the best of our knowledge, this is the first MPI-based parallelization of community detection methods. Previous algorithms have limited scalability. Largest networks processed by most of them are less than 100M edges, which take 10 minutes to an hour. A couple of algorithms [52, 71] can process some $3B$ edges in hours. Our algorithm can process $100M$ edges in 40 seconds. We can process $3B$ edges in minutes. Our algorithm scales almost linearly to a good number of processors. We also provide several analyses regarding quality and runtime trade-off and HPC-based optimization.

8.6 Conclusion

We design MPI-based parallel algorithms for detecting communities in large graphs. Our parallel algorithms are based on the sequential Louvain method. Parallelizing this method for distributed-memory systems poses non-trivial challenges. We propose efficient load balancing and communication approaches to address those issues. Our parallel algorithms work on large graphs and scale to a large number of processors. Further, we also combine variations of several known methods by a hybrid approach to compare speed and quality of the detection. We also adapt edge sparsification techniques with our parallel algorithms for providing even faster computation.

Part IV

Converting Edge List to Adjacency List

Chapter 9

Fast Parallel Conversion of Edge List to Adjacency List for Large-Scale Graphs

In the era of big data, we are deluged with large graph data emerging from numerous social and scientific applications. In most cases, graph data are generated as lists of edges (*edge list*), where an edge denotes a link between a pair of entities. However, most of the graph algorithms work efficiently when information of the adjacent nodes (*adjacency list*) for each node is readily available. Although the conversion from edge list to adjacency list can be trivially done on the fly for small graphs, such conversion becomes challenging for the emerging large-scale graphs consisting of billions of nodes and edges. These graphs do not fit into the main memory of a single computing machine and thus require distributed-memory parallel or external-memory algorithms.

In this chapter, we present efficient MPI-based distributed memory parallel algorithms for converting edge lists to adjacency lists. To the best of our knowledge, this is the first work on this problem. To address the critical load balancing issue, we present a parallel load balancing scheme that improves both time and space efficiency significantly. Our fast parallel algorithm works on massive graphs, achieves very good speedups, and scales to a large number of processors. The algorithm can convert an edge list of a graph with 20 billion edges to the adjacency list in less than 2 minutes using 1024 processors. Denoting the number of nodes, edges, and processors by n, m, and P, respectively, the time complexity of our algorithm is $O(\frac{m}{P} + n + P)$, which provides a speedup factor of at least $\Omega(\min\{P, d_{avg}\})$, where d_{avg} is the average degree of the nodes. The algorithm has a space complexity of $O(\frac{m}{P})$, which is optimal.

9.1 Introduction

We denote a graph by $G(V, E)$, where V and E are the set of vertices (nodes) and edges, respectively, with $m = |E|$ edges and $n = |V|$ vertices. In many cases, a graph is specified by simply listing the edges $(u, v), (v, w), \cdots \in E$, in an arbitrary order, which is called an *edge list*. A graph can also be specified by a collection of adjacency lists of the nodes, where the *adjacency list* of a node v is the list of nodes that are adjacent to v. Many important graph algorithms, such as

computing shortest path, breadth-first search, and depth-first search are executed by exploring the neighbors (adjacent nodes) of the nodes in the graph. As a result, these algorithms work efficiently when the input graph is given as adjacency lists. Although both edge list and adjacency list have a space requirement of $O(m)$, scanning all neighbors of node v in an edge list can take as much as $O(m)$ time compared to $O(d_v)$ time in an adjacency list, where d_v is the degree of node v.

An adjacency matrix is another data structure used for graphs. Much of the earlier work [4, 26] use an adjacency matrix $A[.,.]$ of order $n \times n$ for a graph with n nodes. Element $A[i, j]$ denotes whether node j is adjacent to node i. All adjacent nodes of i can be determined by scanning the i-th row, which takes $O(n)$ time compared to $O(d_i)$ time for adjacency list. Further, an adjacency matrix has a prohibitive space requirement of $O(n^2)$ compared to $O(m)$ for an adjacency list. In a real-world network, m can be much smaller than n^2 as the average degree of a node can be significantly smaller than n. Thus an adjacency matrix is not suitable for the analysis of emerging large-scale networks in the age of big data.

In most cases, a graph is generated as a list of edges, since it is easier to capture pairwise interactions among entities in a system in arbitrary order than to capture all interactions of a single entity at the same time. Examples include capturing person-person connection in social networks and protein-protein links in protein interaction networks. This is true even for generating large random graphs [2, 23], which is useful for modeling very large systems. As discussed by Leskovec et. al [43], some patterns only exist in large datasets and they are fundamentally different from those in smaller datasets. While generating such large random graphs, algorithms usually output edges one by one. Edges incident on a node v are not necessarily generated consecutively. Thus a conversion of edge list to adjacency list is necessary for analyzing these graphs efficiently.

Emerging large networks have millions to billions of nodes and edges [22]. These networks hardly fit in the memory of a single machine and thus require external memory or distributed memory parallel algorithms. Now external memory algorithms can be very I/O intensive leading to a large runtime. Efficient distributed memory parallel algorithms can solve both problems (runtime and space) by distributing computing tasks and data to multiple processors.

In a sequential setting, with the graphs being small enough to be stored in main memory, the problem of converting an edge list to an adjacency list representation is trivial as described in the next section. However, the problem in a distributed-memory setting with massive graphs poses many non-trivial challenges. The neighbors of a particular node v might reside in multiple processors, which need to be combined efficiently. Further, computation loads must be well-balanced among the processors to achieve a good performance of the parallel algorithm. Like many others, this problem demonstrates how a simple trivial problem can turn into a challenging problem when we are dealing with big data.

Contributions. In this chapter, we study the problem of converting an *edge list* to an *adjacency list* representation for large-scale graphs. We present MPI-based distributed-memory parallel algorithms which work for both directed and undirected graphs. We devise a parallel load balancing scheme that balances the computational load very well and improves the efficiency of the algorithms significantly, both in terms of runtime and space requirement. Furthermore, we present two efficient merging schemes for combining neighbors of a node from different processors, *message-based* and *external-memory* merging, which offer a convenient trade-off between space and run-

time. Our algorithms work on large graphs, demonstrate very good speedups on both real and artificial graphs, and scale to a large number of processors. The edge list of a graph with $20B$ edges can be converted to adjacency list in two minutes using 1024 processors. We also provide rigorous theoretical analysis of the time and space complexity of our algorithms. The time and space complexity of our algorithms are $O(\frac{m}{P} + n + P)$ and $O(\frac{m}{P})$, respectively, where n, m, and P are the number of the nodes, edges, and processors, respectively. The speedup factor is at least $\Omega(\min\{P, d_{avg}\})$, where d_{avg} is the average degree of the nodes.

9.2 Preliminaries and Background

In this section, we describe the basic definitions used in this chapter and then present a sequential algorithm for converting an edge list to an adjacency list representation.

9.2.1 Basic Definitions

We assume n nodes of the graph $G(V, E)$ are labeled as $0, 1, 2, \ldots, n-1$. If $(u, v) \in E$, we say u and v are neighbors of each other. The set of all adjacent nodes (neighbors) of $v \in V$ is denoted by N_v, i.e., $N_v = \{u \in V | (u, v) \in E\}$. The degree of v is $d_v = |N_v|$.

In an edge list representation, edges $(u, v) \in E$ are listed one after another without any particular order. Edges incident to a particular node v are not necessarily listed together. On the other hand, in an adjacency list representation, for all v, adjacent nodes of v, N_v, are listed together. An example of these representations is shown in Figure 9.1.

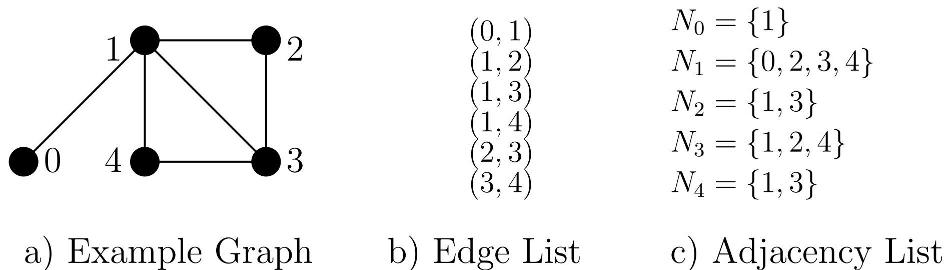


Figure 9.1: The edge list and adjacency list representations of an example graph with 5 nodes and 6 edges.

9.2.2 A Sequential Algorithm

The sequential algorithm for converting edge list to adjacency list works as follows. Create an empty list N_v for each node v, and then, for each edge $(u, v) \in E$, include u in N_v and v in N_u. The pseudocode of the sequential algorithm is given in Figure 3.1. For a directed graph, line 5 of the algorithm should be omitted since a directed edge (u, v) does not imply that there is also an

```
1: for each v ∈ V do
2:     N_v ← ∅
3: for each (u, v) ∈ E do
4:     N_u ← N_u ∪ {v}
5:     N_v ← N_v ∪ {u}
```

Figure 9.2: Sequential algorithm for converting edge list to adjacency list.

edge (v, u). In our subsequent discussion, we assume that the graph is undirected. However, the algorithm also works for the directed graph with the mentioned modification.

This sequential algorithm is optimal since it takes $O(m)$ time to process $O(m)$ edges and thus cannot be further improved. The algorithm has a space complexity of $O(m)$.

For small graphs that can be stored entirely in main memory, the conversion in a sequential setting is trivial. However, emerging large graphs pose many non-trivial challenges in terms of memory and execution efficiency. Such graphs might not fit in the local memory of a single computing node. Even if some of them fit in the main memory, the runtime might be prohibitively large. Efficient parallel algorithms can solve this problem by distributing computation and data among computing nodes. We present our parallel algorithm in the next section.

9.3 The Parallel Algorithm

First we present an overview of our parallel algorithm. A detailed description follows thereafter.

9.3.1 Overview of the Algorithm

Let P be the number of processor used in the computation and E be the list of edges given as input. Our algorithm has two phases of computation. In Phase 1, the edges E are partitioned into P *initial partitions* E_i, and each processor is assigned one such partition. Each processor then constructs neighbor lists from the edges of its own partition. However, edges incident to a particular node might reside in multiple processors, which creates multiple partial adjacency lists for the same node. In Phase 2 of our algorithms, such adjacency lists are merged together. Now, performing Phase 2 of the algorithm in a cost-effective way is very challenging. Further, computing loads among processors in both phases need to be balanced to achieve a significant runtime efficiency. The load balancing scheme should also make sure that the space requirement among processors is also balanced so that large graphs can be processed. We describe the phases of our parallel algorithm in detail as follows.

9.3.2 (Phase 1) Local Processing

The algorithm partitions the set of edges E into P partitions E_i such that $E_i \subseteq E$, $\bigcup_k E_k = E$ for $0 \leq k \leq P-1$. Each partition E_i has almost $\frac{m}{P}$ edges– to be exact, $\lceil \frac{m}{P} \rceil$ edges, except for the last partition which has slightly fewer $(m-(p-1)\lceil \frac{m}{P} \rceil)$. Processor i is assigned partition E_i. Processor i then constructs adjacency lists N_v^i for all nodes v such that $(., v) \in E_i$ or $(v, .) \in E_i$. Note that adjacency list N_v^i is only a partial adjacency list since other partitions E_j might have edges incident on v. We call N_v^i *local adjacency list* of v in partition i. The pseudocode for Phase 1 computation is presented in Figure 9.3.

```
1: Each processor i, in parallel, executes the following.
2: for (u, v) ∈ E_i do
3:     N_v^i ← N_v^i ∪ {u}
4:     N_u^i ← N_u^i ∪ {v}
```

Figure 9.3: Algorithm for performing Phase 1 computation.

This phase of computation has both the runtime and space complexity of $O(\frac{m}{P})$ as shown in Lemma 7 .

Lemma 7 *Phase 1 of our parallel algorithm has both the runtime and space complexity of* $O(\frac{m}{P})$.

Proof: Each initial partition i has $|E_i| = O(\frac{m}{P})$ edges. Executing Line 3-4 in Figure 9.3 for $O(\frac{m}{P})$ edges requires $O(\frac{m}{P})$ time. Now the total space required for storing local adjacency lists N_v^i in partition i is $2|E_i| = O(\frac{m}{P})$. $\square$

Thus the computing loads and space requirements in Phase 1 are well-balanced. The second phase of our algorithm constructs the final adjacency list N_v from local adjacency lists N_v^i from all processors i. Note that balancing load for Phase 1 does not make load well balanced for Phase 2 which requires a more involved load balancing scheme as described later in the following sections.

9.3.3 (Phase 2) Merging Local Adjacency Lists

Once all processors complete constructing local adjacency lists N_v^i, final adjacency lists N_v are created by merging N_v^i from all processors i as follows.

$$N_v = \bigcup_{i=0}^{P-1} N_v^i \tag{9.1}$$

The scheme used for merging local adjacency lists has significant impact on the performance of the algorithm. One might think of using a dedicated *merger* processor. For each node $v \in V_i$, the merger collects N_v^i from all other processors and merges them into N_v. This requires $O(d_v)$ time

for node v. Thus the runtime complexity for merging adjacency lists of all $v \in V$ is $O(\sum_{v \in V} d_v) = O(m)$, which is at most as good as the sequential algorithm.

To achieve parallelism in merging, multiple mergers can be employed instead of a single merger. Every merger can merge local adjacency list of two processors, in a binary tree style (Figure 9.4). For each node $v \in V$, the parallel merging with P processors with the binary tree scheme works as below.

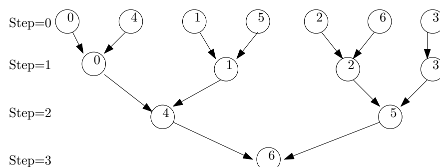


Figure 9.4: Parallel merging with the binary tree scheme ($P = 7$). Numbers in the circle denote rank of the processors.

i. Step 0 corresponds to the construction of local adjacency lists. In step 1, lower ranked $\lceil \frac{P}{2} \rceil$ processors k merges N_v^k and $N_v^{k+\lceil \frac{P}{2} \rceil}$. The rank of mergers k starts from 0 to $\lceil \frac{P}{2} \rceil - 1$. For P being an odd number, processor $k = \lceil \frac{P}{2} \rceil - 1$ simply passes its list N_v^k to the next step.

ii. In step $i > 1$, there are $\lceil \frac{P}{2^i} \rceil$ mergers working in parallel. The ranks k of merging processors range from $\lceil \frac{P}{2^{i-1}} \rceil$ to ($\lceil \frac{P}{2^{i-1}} \rceil$ + $\lceil \frac{P}{2^i} \rceil$ -1). The $j - th$ merger of step i merges the output of $2j$-th and $(2j + 1)$-th mergers of step $(i - 1)$.

It is easy to see that, the merger acting as the root of the tree constructs the final adjacency list $N_v = \bigcup_{i=0}^{P-1} N_v^i$. This scheme allows further improvement in efficiency by allowing pipelining: when a merger is done merging local lists for v, it sends it to the merger of the next step and start merging the next node $v+1$. Thus the scheme achieves a good parallelism in early steps. However, the cost for merging in the last step is $O(d_v)$ for node v, yielding a total cost of $O(\sum_{v \in V} d_v) = O(m)$ for all $v \in V$. This effectively diminishes the parallelism gained in previous steps. Next we present our efficient parallel merging scheme.

An Efficient Parallel Merging Scheme

To parallelize Phase 2 efficiently, our algorithm distributes the corresponding computation disjointly among processors. Each processor i is responsible for merging adjacency lists N_v for nodes v in $V_i \subset V$ such that for any i and j, $V_i \cap V_j = \emptyset$ and $\bigcup_i V_i = V$. Note that this partitioning of nodes is different from the initial partitioning of edges. How the nodes in V are distributed among processors crucially affects the load balancing and performance of the algorithm. Further, this partitioning and load balancing scheme should be parallel to ensure the efficiency of the algorithm. Later in this section, we discuss a parallel algorithm to partition set of nodes V which makes both space requirement and runtime well-balanced. Once the partitions V_i are given, the scheme for parallel merging works as follows.

- *Step 1:* Let S_i be the set of all local adjacency lists in partition i. Processor i divides S_i into P disjoint subsets S_i^j, $0 \leq j \leq P-1$, as defined below.

$$S_i^j = \{N_v^i : v \in V_j\}. \tag{9.2}$$

- *Step 2:* Processor i sends S_i^j to all other processors j. This step introduces non-trivial efficiency issues which we shall discuss shortly.

- *Step 3:* Once processor i gets S_j^i from all processors j, it constructs N_v for all $v \in V_i$ by the following equation.

$$N_v = \bigcup_{k : N_v^k \in S_k^i} N_v^k \tag{9.3}$$

We present two methods for performing Step 2 of the above scheme. The first method explicitly exchanges messages among processors to send and receive S_i^j by using message buffers (main memory). The other method uses disk space (external memory) to exchange S_i^j. We call the first method *message-based merging* and the second *external-memory merging*.

(1) Message-based Merging: Each processor i sends S_i^j directly to processor j via messages. Specifically, processor i sends N_v^i (with a message $< v, N_v^i >$) to processor j where $v \in V_j$. A processor might send multiple lists to another processor. In such cases, messages to a particular processor are bundled together to reduce communication overhead. Once a processor i receives messages $< v, N_v^j >$ from other processors, for $v \in V_i$, it computes $N_v = \cup_{j=0}^{P-1} N_v^j$. The pseudocode of this algorithm is given in Figure 9.5.

1: **for** *each* v s.t. $(., v) \in E_i \vee (v, .) \in E_i$ **do**
2:     *Send* $< v, N_v^i >$ to proc. j where $v \in V_j$
3: **for** *each* $v \in V_i$ **do**
4:     $N_v \leftarrow \emptyset$
5: **for** *each* $< v, N_v^j >$ received from any proc. j **do**
6:     $N_v \leftarrow N_v \cup N_v^j$

Figure 9.5: Parallel algorithm for merging local adjacency lists to construct final adjacency lists N_v. A message, denoted by $< v, N_v^i >$, refers to local adjacency lists of v in processor i.

(2) External-memory Merging: Each processor i writes S_i^j in intermediate disk files F_i^j, one for each processor j. Processor i reads all files F_j^i for partial adjacency lists N_v^j for each $v \in V_i$ and merges them to final adjacency lists using step 3 of the above scheme. However, processor i does not read in the whole file into its main memory. It only stores local adjacency lists N_v^j of a node v at a time, merges it to N_v, releases memory and then proceeds to merge the next node $v+1$. This works correctly since while writing S_i^j in F_i^j, local adjacency lists N_v^i are listed in the sorted order of v. External-memory merging thus has a space requirement of $O(\max_v d_v)$. However, the I/O operation leads to a higher runtime with this method than message-based merging, although the asymptotic runtime complexity remains the same. We demonstrate this space-runtime tradeoff between these two methods in our *performance analysis* section.

The runtime and space complexity of parallel merging depends on the partitioning of V. Next, we discuss the partitioning and load balancing scheme followed by the complexity analyses.

9.3.4 Partitioning and Load Balancing

The performance of the algorithm depends on how loads are distributed. In Phase 1, distributing the edges of the input graph evenly among processors provides an even load balancing both in terms of runtime and space leading to both space and runtime complexity of $O(\frac{m}{P})$. However, Phase 2 is computationally different than Phase 1 and requires a different partitioning and load balancing scheme.

In Phase 2 of our algorithm, the set of nodes V is divided into P subsets V_i where processor i merges adjacency lists N_v for all $v \in V_i$. The time for merging N_v of a node v (referred to as *merging cost* henceforth) is proportional to the degree $d_v = |N_v|$ of node v. Total cost for merging incurred on a processor i is $\Theta(\sum_{v \in V_i} d_v)$. Distributing equal number of nodes among processors may not make the computing load well-balanced in many cases. Some nodes may have large degrees and some very small. As shown in Figure 9.6, distribution of merging cost ($\sum_{v \in V_i} d_v$) across processors is very uneven with an equal number of nodes assigned to each processor. Thus the set V should be partitioned in such a way that the cost of merging is almost equal in all processors.

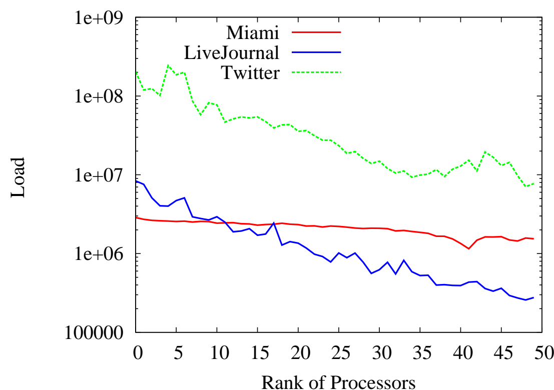


Figure 9.6: Load distribution among processors for LiveJournal, Miami and Twitter before applying the load balancing scheme.

Let, $f(v)$ be the cost associated with constructing N_v by receiving and merging local adjacency lists for a node $v \in V$. We need to compute P disjoint partitions of node set V such that for each partition V_i,

$$\sum_{v \in V_i} f(v) \approx \frac{1}{P} \sum_{v \in V} f(v).$$

Now, note that, for each node v, total size of the local adjacency lists received (in Line 5 in Figure 9.5) equals the number of adjacent nodes of v, i.e., $|N_v| = d_v$. Merging local adjacency lists N_v^j via set union operation (Line 6) also requires d_v time. Thus, $f(v) = d_v$. Now, since the

adjacent nodes of a node v can reside in multiple processors, computing $f(v) = |N_v| = d_v$ requires communication among multiple processors. For all v, computing $f(v)$ sequentially requires $O(m+n)$ time which diminishes the advantages gained by the parallel algorithm. Thus, we compute $f(v) = d_v$ for all v in parallel, in $O(\frac{n+m}{P} + c)$ time, where c is the communication cost. We will discuss the complexity shortly. This algorithm works as follows: for determining d_v for $v \in V$ in parallel, each processor i computes d_v for $\frac{n}{P}$ nodes v, where v starts from $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. Such nodes v satisfy the equation, $\left\lfloor \frac{v}{n/P} \right\rfloor = i$. Now, for each local adjacency list N_v^i constructed in Phase 1, processor i sends $d_v^i = |N_v^i|$ to processor $j = \frac{v}{n/P}$ with a message $< v, d_v^i >$. Once processor i receives messages $< v, d_v^j >$ from other processors, it computes $f(v) = d_v = \sum_{j=0}^{P-1} d_v^j$ for all nodes v such that $\left\lfloor \frac{v}{n/P} \right\rfloor = i$. The pseudocode of the parallel algorithm for computing $f(v) = d_v$ is given in Figure 9.7.

1: **for** *each* v s.t. $(., v) \in E_i \vee (v, .) \in E_i$ **do**
2:     $d_v^i \leftarrow |N_v^i|$
3:     $j \leftarrow \frac{v}{n/P}$
4:     *Send* $< v, d_v^i >$ to processor j
5: **for** *each* v s.t. $\left\lfloor \frac{v}{n/P} \right\rfloor = i$ **do**
6:     $d_v \leftarrow 0$
7: **for** *each* $< v, d_v^j >$ received from any proc. j **do**
8:     $d_v \leftarrow d_v + d_v^j$

Figure 9.7: Parallel algorithm executed by each processor i for computing $f(v) = d_v$.

Once $f(v)$ is computed for all $v \in V$, we compute cumulative sum $F(t) = \sum_{v=0}^{t} f(v)$ in parallel by using a parallel prefix sum algorithm [5]. Each processor i computes and stores $F(t)$ for nodes t, where t starts from $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. This computation takes $O(\frac{n}{P} + P)$ time. Then, we need to compute V_i such that computation loads are well-balanced among processors. Partitions V_i are disjoint subset of consecutive nodes, i.e., $V_i = \{n_i, n_i + 1 \ldots, n_{(i+1)} - 1\}$ for some node n_i. We call n_i *start* node or *boundary* node of partition i. Now, V_i is computed in such a way that the sum $\sum_{v \in V_i} f(v)$ becomes almost equal ($\frac{1}{P} \sum_{v \in V} f(v)$) for all partitions i. At the end of this execution, each processor i knows n_i and $n_{(i+1)}$. Algorithm presented in [8] compute V_i for the problem of triangle counting. The algorithm can also be applied for our problem to compute V_i using cost function $f(v) = d_v$. In summary, computing load balancing for Phase 2 has the following main steps.

- *Step 1:* Compute cost $f(v) = d_v$ for all v in parallel by the algorithm shown in Figure 9.7.
- *Step 2:* Compute cumulative sum $F(v)$ by a parallel prefix sum algorithm [5].
- *Step 3:* Compute boundary nodes n_i for every subset $V_i = \{n_i, \ldots, n_{(i+1)} - 1\}$ using the algorithms [3, 8].

Lemma 8 *The algorithm for balancing loads for Phase 2 has a runtime complexity of $O(\frac{n+m}{P} + P + \max_i M_i)$ and a space requirement of $O(\frac{n}{P})$, where M_i is the number of messages received by processor i in Step 1.*

Proof: For *Step 1* of the above load balancing scheme, executing Line 1-4 (Figure 9.7) requires $O(|E_i|)$=$O(\frac{m}{P})$ time. The cost for executing Line 5-6 is $O(\frac{n}{P})$ since there are $\frac{n}{P}$ nodes v such that $\left\lfloor \frac{v}{n/P} \right\rfloor = i$. Each processor i sends a total of $O(\frac{m}{P})$ messages since $|E_i| = \frac{m}{P}$. If the number of messages received by processor i is M_i, then Line 7-8 of the algorithm has a complexity of $O(M_i)$ (we compute bounds for M_i in Lemma 9). Computing *Step 2* has a computational cost of $O(\frac{n}{P} + P)$ [5]. *Step 3* of the load balancing scheme requires $O(\frac{n}{P} + P)$ time [3, 8]. Thus the runtime complexity of the load balancing scheme is $O(\frac{n+m}{P} + P + \max_i M_i)$. Storing $f(v)$ for $\frac{n}{P}$ nodes has a space requirement of $O(\frac{n}{P})$. □

Lemma 9 *Number of messages M_i received by processor i in Step 1 of load balancing scheme is bounded by $O(\min\{n, \sum_{in/P}^{(i+1)n/P-1} d_v\})$.*

Proof: Referring to Figure 9.7, each processor i computes d_v for $\frac{n}{P}$ nodes v, where v starts from $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. For each v, processor i may receive messages from at most $(P-1)$ other processors. Thus, the number of received messages is at most $\frac{n}{P} \times (p-1) = O(n)$. Now, notice that, when all neighbors $u \in N_v$ of v reside in different partitions E_j, processor i might receive as much as $|N_v| = d_v$ messages for node v. This gives another upper bound, $M_i = O(\sum_{in/P}^{(i+1)n/P-1} d_v)$. Thus we have $M_i = O(\min\{n, \sum_{in/P}^{(i+1)n/P-1} d_v\})$. □

In most of the practical cases, each processor receives a much smaller number of messages than that specified by the theoretical upper bound. Now, for each node v, processor i receives messages actually from fewer than $P-1$ processors. Let, for node v, processor i receives messages from $O(P.l_v)$ processors, where l_v is a real number $(0 \le l_v \le 1)$. Thus total number of message received, $M_i = O(\sum_{in/P}^{(i+1)n/P-1} Pl_v)$. To get a crude estimate of M_i, let $l_v = l$ for all v. The term l can be thought of as the average over all l_v. Then $M_i = O(\frac{n}{P}Pl) = O(nl)$. As shown in Table 9.1, the actual number of messages received M_i is up to $7\times$ smaller than the theoretical bound.

Table 9.1: Number of messages received in practice compared to the theoretical bounds. This results report $\max_i M_i$ with $P = 50$.

Network	**n**	$\sum_{\frac{in}{P}}^{\frac{(i+1)n}{P}-1} d_v$	M_i	l**(avg.)**
Miami	2.1M	2.17M	600K	0.27
LiveJournal	4.8M	2.4M	$560K$	0.14
PA($5M$, 20)	$5M$	$2.48M$	$1.4M$	0.28

Lemma 10 *Using the load balancing scheme discussed in this section, Phase 2 of our parallel algorithm has a runtime complexity of $O(\frac{m}{P})$. Further, the space required to construct all final adjacency lists N_v in a partition is $O(\frac{m}{P})$.*

Proof: Line 1-2 in the algorithm shown in Figure 9.5 requires $O(|E_i|)$=$O(\frac{m}{P})$ time for sending at most $|E_i|$ edges to other processors. Now, with load balancing, each processor receives and merges at most $O(\sum_{v \in V} d_v/P) = O(\frac{m}{P})$ edges (Line 5-6). Thus the cost for merging local lists N_v^j into final list N_v has a runtime of $O(\frac{m}{P})$. Since the total size of the local and final adjacent lists in a partition is $O(\frac{m}{P})$, the space requirement is $O(\frac{m}{P})$. □

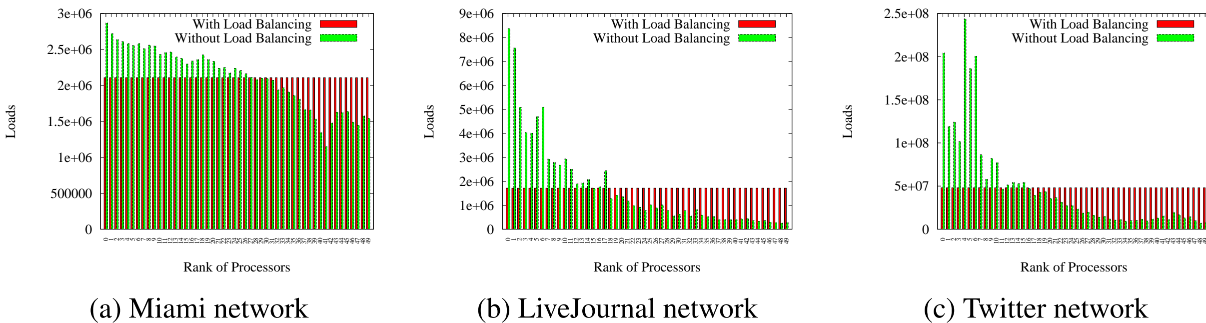
The runtime and space complexity of our complete parallel algorithm are formally presented in Theorem 11.

Theorem 11 *The runtime and space complexity of our parallel algorithm is* $O(\frac{m}{P} + P + n)$ *and* $O(\frac{m}{P})$*, respectively.*

Proof: The proof follows directly from Lemmas 1, 2, 3, and 4.□

The total space required by all processors to process m edges is $O(m)$. Thus the space complexity $O(\frac{m}{P})$ of our parallel algorithm is optimal.

Performance gain with load balancing: Cost for merging incurred on each processor i (pseudocode shown in Figure 9.5) is $\Theta(\sum_{v \in V_i} d_v)$. Without load balancing, this cost $\Theta(\sum_{v \in V_i} d_v)$ can be as much as $\Theta(m)$ (it is easy to construct such skewed graphs) leading the runtime complexity of the algorithm $\Theta(m)$. With load balancing scheme our algorithm achieves a runtime of $O(\frac{m}{P} + P + n) = O(\frac{m}{P} + \frac{m}{d_{avg}})$, for usual case $n > P$. Thus, by simple algebraic manipulation, it is easy to see, the algorithm with load balancing scheme achieves a $\Omega(\min\{P, d_{avg}\})$-factor gain in runtime efficiency over the algorithm without load balancing scheme. In other words, the algorithm gains a $\Omega(P)$-fold improvement in speedup when $d_{avg} \geq P$ and $\Omega(d_{avg})$-fold otherwise. We demonstrate this gain in speedup with experimental results in our *performance analysis* section.

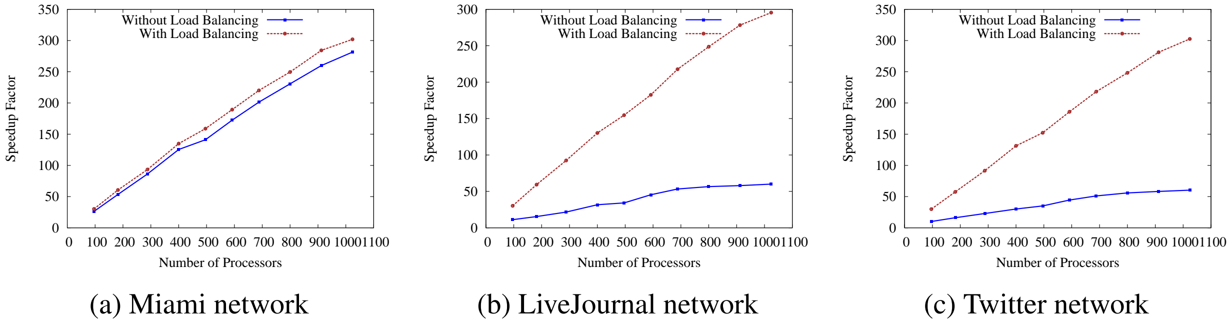


(a) Miami network (b) LiveJournal network (c) Twitter network

Figure 9.8: Load distribution among processors for LiveJournal, Miami and Twitter networks by different schemes.

9.4 Performance Analysis

In this section, we present the experimental results evaluating the performance of our algorithm.



(a) Miami network (b) LiveJournal network (c) Twitter network

Figure 9.9: Strong scaling of our algorithm on LiveJournal, Miami and Twitter networks with and without load balancing scheme. Computation of speedup factors includes the cost for load balancing.

9.4.1 Load Distribution

Load distribution among processors can be very uneven without applying our load balancing scheme, as discussed in the *partitioning and load balancing* section. We show a comparison of load distribution on various networks with and without load balancing scheme in Figure 9.8. Our scheme provides an almost equal load among the processors, even for graphs with very skewed degree distributions such as LiveJournal and Twitter. Loads are significantly uneven for such skewed networks without the load balancing scheme.

9.4.2 Strong Scaling

Figure 9.9 shows strong scaling (speedup) of our algorithm on the LiveJournal, Miami and Twitter networks with and without the load balancing scheme. Our algorithm demonstrates very good speedups, e.g., it achieves a speedup factor of ≈ 300 with 1024 processors for the Twitter network. Speedup factors increase almost linearly for all networks, and the algorithm scales to a large number of processors. Figure 9.9 also shows the speedup factors the algorithm achieves without the load balancing scheme. Speedup factors with load balancing scheme are significantly higher than those without load balancing scheme. For the Miami network, the differences in speedup factors are not very large since Miami has a relatively even degree distribution and loads are already fairly balanced without load balancing scheme. However, for real-world skewed networks, our load balancing scheme always improves the speedup quite significantly; for example, with 1024 processors, the algorithm achieves a speedup factor of 297 with the load balancing scheme compared to 60 without load balancing scheme for the LiveJournal network.

This experiment also demonstrates that our algorithm scales to a large number of processors. The speedup factors continue to grow almost linearly up to 1024 processors.

9.4.3 Comparison between Message-based and External-memory Merging

We compare the runtime and memory usage of our algorithm with both message-based and external-memory merging. Message-based merging is very fast and uses message buffers in main memory for communication. On the other hand, external-memory merging saves main memory by using disk space even though it requires large runtime for I/O operations. Thus these two methods provide desirable alternatives to a trade-off between space and runtime. However, as shown in Table 9.2, message-based merging is significantly faster (up to 20 times) than external-memory merging albeit taking a little larger space. Thus, message-based merging is the preferable method in our fast parallel algorithm.

Table 9.2: Comparison of external-memory (EXT) and message-based (MSG) merging (using 50 processors).

Network	**Memory (MB)**		**Runtime (s)**	
	EXT	**MSG**	**EXT**	**MSG**
Email-Enron	1.8	2.4	3.371	0.078
web-BerkStan	7.6	10.3	10.893	1.578
Miami	26.5	43.34	33.678	6.015
LiveJournal	28.7	42.4	31.075	5.112
Twitter	685.93	1062.7	1800.984	90.894
Gnp(500K, 20)	6.1	9.8	6.946	1.001
PA(5M, 20)	68.2	100.1	35.837	7.132
PA(1B, 20)	9830.5	12896.6	14401.5	1198.30

9.4.4 Weak Scaling

Weak scaling of a parallel algorithm shows the ability of the algorithm to maintain constant computation time when the problem size grows proportionally with the increasing number of processors. As shown in Figure 9.10, total computation time of our algorithm (including load balancing time) grows slowly with the addition of processors. This is expected since the communication overhead increases with additional processors. However, the growth of runtime of our algorithm is rather slow and remains almost constant, the weak scaling of the algorithm is very good.

9.5 Conclusion

We present a parallel algorithm for converting an *edge-list* of a graph to an *adjacency-list* representation. The algorithm scales well to a large number of processors and works on massive graphs. We devise a load balancing scheme that improves both space efficiency and runtime of the algorithm, even for networks with very skewed degree distributions. To the best of our knowledge, it is the first parallel algorithm to convert edge list to adjacency list for large-scale graphs. It also allows

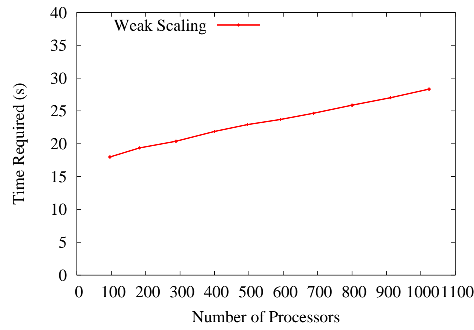


Figure 9.10: Weak scaling of our parallel algorithm. For this experiment we use networks PA($x/10 \times 1M, 20$) for x processors.

other graph algorithms to work on large graphs that emerge naturally as edge lists. Furthermore, this work demonstrates how a seemingly trivial problem becomes challenging when we are dealing with big data.

Chapter 10

General Conclusion

We present algorithms and analysis for mining large real-world networks. These networks are often characterized by an abundance of triangles and the existence of well-structured communities. Counting triangles is a very important problem in network mining and analysis. Many other graph problems, such as computation of clustering coefficients and transitivity, can be solved using an efficient enumeration of triangles. We devise fast and scalable parallel algorithms for counting and listing triangles in big networks. We provide parallel partitioning and load balancing schemes to design runtime efficient algorithms. Our algorithms are also space-efficient and thus allow us to work on big networks using widely available commodity machines. We also present how we can characterize networks by quantifying the number of common neighbors and demonstrate its relationship with other network properties. Such characterization will be proven useful in understanding interesting properties and structures of real-world networks. Another very important problem in network analysis and mining is community detection. Communities reveal useful organizational information of complex systems represented by networks. We devise distributed-memory parallel algorithms for detecting communities, which scale to big networks and achieve good parallel speedups. We also combine sparsification methods with our parallel algorithms to provide even faster detection of reasonable communities. Finally, we present fast parallel algorithms for converting edge list to adjacency list of big networks. Although such conversion is simple for small networks, the emerging networks with billions of nodes and edges pose non-trivial challenges. We present efficient high performance computing based techniques leading to fast and space-efficient algorithms. All the parallel algorithms presented in this dissertation scale to a large number of processors, can work on big networks, and demonstrate good speedups. We believe that these algorithms and HPC-based techniques will be proven useful in mining big data represented by networks. The novel analysis and characterization based on triangular statistics and communities will reveal important insights about big real-world networks.

Bibliography

[1] Twitter Data. http://an.kaist.ac.kr/~haewoon/release/twitter_social_graph, 2010. [Online].

[2] M Alam, M Khan, and M Marathe. Distributed-memory Parallel Algorithms for Generating Massive Scale-free Networks Using Preferential Attachment Model. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[3] M Alam, M Khan, and M Marathe. Parallel algorithms for generating random networks with given degree sequences. In *12th IFIP International Conference on Network and Parallel Computing (NPC)*, 2015.

[4] N Alon, R Yuster, and U Zwick. Finding and Counting Given length Cycles. *Algorithmica*, 17:209–223, 1997.

[5] S Aluru. Teaching Parallel Computing Through Parallel Prefix. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[6] C Apte, B Liu, E Pednault, and P Smyth. Business applications of data mining. *Commun. ACM*, 45(8):49–53, 2002.

[7] S Arifuzzaman and M Khan. Fast parallel conversion of edge list to adjacency list for large-scale graphs. In *23rd High Performance Computing Symposium*, 2015.

[8] S Arifuzzaman, M Khan, and M Marathe. PATRIC: A parallel algorithm for counting triangles in massive networks. In *22nd ACM International Conference on Information and Knowledge Management*, 2013.

[9] S Arifuzzaman, M Khan, and M Marathe. A Space-efficient Parallel Algorithm for Counting Exact Triangles in Massive Networks. In *17th IEEE International Conference on High Performance Computing and Communications*, 2015.

[10] S Arifuzzaman, M Khan, and M Marathe. A fast parallel algorithm for counting triangles in graphs using dynamic load balancing. In *2015 IEEE BigData Conference*, 2015.

[11] P Attewell and D Monaghan. *Data Mining for the Social Sciences: An Introduction*. University of California Press, 2015.

[12] Z Bar-Yosseff, R Kumar, and D Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[13] A Barabasi and R Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[14] C Barrett, R Beckman, M Khan, VS Anil Kumar, M Marathe, P Stretz, T Dutta, and B Lewis. Generation and analysis of large synthetic social contact networks. In *Winter Simulation Conference*, 2009.

[15] L Becchetti, P Boldi, C Castillo, and A Gionis. Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs. In *4th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2008.

[16] V Blondel, J Guillaume, R Lambiotte, and E Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10:10008, 2008.

[17] B Bollobas. *Random Graphs*. Cambridge Univ. Press, 2001.

[18] A Broder, R Kumar, F Maghoul, P Raghavan, S Rajagopalan, R Stata, A Tomkins, and J Wiener. Graph structure in the Web. *Computer Networks*, 33(1–6):309–320, 2000.

[19] L Buriol, G Frahling, S Leonardi, A Marchetti-Spaccamela, and C Sohler. Counting triangles in data streams. In *25th ACM Symposium on Principles of Database Systems*, 2006.

[20] J Chen and S Lonardi. *Biological Data Mining*. Chapman & Hall/CRC, 2009.

[21] N Chiba and T Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

[22] S Chu and J Cheng. Triangle Listing in Massive Networks and Its Applications. In *17th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2011.

[23] F Chung and L Lu. *Complex Graphs and Networks*. American Mathematical Society, 2006.

[24] M Ciglan, M Laclavik, and K Nørvåg. On Community Detection in Real-world Networks and the Importance of Degree Assortativity. In *19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013.

[25] A Clauset, M Newman, and C Moore. Finding community structure in very large networks. *Physical Review E*, 70(6):66111, 2004.

[26] D Coppersmith and S Winograd. Matrix Multiplication via Arithmetic Progressions. In *19th Annual ACM Symposium on Theory of Computing*, 1987.

[27] J Dean and S Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design and Implementation*, 2004.

[28] L Donetti and M Munoz. Detecting network communities: a new systematic and efficient algorithm. *Journal of Statistical Mechanics: Theory and Experiment*, 2004(10):P10012, 2004.

[29] S Dongen. Graph Clustering by Flow Simulation. *PhD Thesis, University of Utrecht, The Netherlands*, 2000.

[30] J Eckmann and E Moses. Curvature of co-links uncovers hidden thematic layers in the World Wide Web. *Proceedings of the National Academy of Sciences*, 99(9):5825–5829, 2002.

[31] S Fortunato and A Lancichinetti. Community detection algorithms: a comparative analysis. In *4th International ICST Conference on Performance Evaluation Methodologies and Tools*, 2009.

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

[33] D Gleich and C Seshadri. Vertex Neighborhoods, Low Conductance Cuts, and Good Seeds for Local Community Methods. In *18th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2012.

[34] O Green, P Yalamanchili, and L Munguía. Fast Triangle Counting on the GPU. In *4th Workshop on Irregular Applications: Architectures and Algorithms*, 2014.

[35] D Gruhl, R Guha, D Liben-Nowell, and A Tomkins. Information Diffusion Through Blogspace. In *13th International Conference on World Wide Web*, 2004.

[36] R Guimera and L Amaral. Functional cartography of complex metabolic networks. *Nature*, 2005.

[37] R Gupta, T Roughgarden, and C Seshadhri. Decompositions of Triangle-Dense Graphs. In *5th Conference on Innovations in Theoretical Computer Science*, 2014.

[38] M Jha, C. Seshadhri, and A Pinar. A Space Efficient Streaming Algorithm for Triangle Counting Using the Birthday Paradox. In *19th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2013.

[39] T Kolda, A Pinar, T Plantenga, and C Seshadhri. A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing*, 36(5), 2014.

[40] T Kolda, A Pinar, T Plantenga, C. Seshadhri, and C Task. Counting Triangles in Massive Graphs with MapReduce. *SIAM Journal on Scientific Computing*, 36(5):S48–S77, 2014.

[41] H Kwak, C Lee, and Others. What is Twitter, a social network or a news media? In *19th International Conference on World Wide Web*, pages 591–600, 2010.

[42] M Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407:458–473, 2008.

[43] J Leskovec. Dynamics of Large Networks. In *Ph.D. Thesis, Pittsburgh, PA, USA.*, 2008.

[44] J Leskovec, D Chakrabarti, J Kleinberg, C Faloutsos, and Z Ghahramani. Kronecker Graphs: An Approach to Modeling Networks. *eprint arXiv:0812.4905*, 0812.4905, 2008.

[45] J Leskovec, J Kleinberg, and C Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2005.

[46] J Leskovec, K Lang, and M Mahoney. Empirical comparison of algorithms for network community detection. In *19th International Conference on World Wide Web*, 2010.

[47] M McPherson, L Smith-Lovin, and J Cook. Birds of a Feather: Homophily in Social Networks. *Annual Rev. of Soc.*, 27(1):415–444, 2001.

[48] R Milo, S Shen-Orr, N Kashtan, D Chklovskii, and U Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[49] M Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.

[50] M Newman. Coauthorship networks and patterns of scientific collaboration. *Proceedings of the National Academy of Sciences*, 101(1):5200–5205, 2004.

[51] M Newman, S Strogatz, and D Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E*, 64, 2001.

[52] M Ovelgönne. Distributed Community Detection in Web-scale Networks. In *2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2013.

[53] R Pagh and C Tsourakakis. Colorful triangle counting and a MapReduce implementation. *Information Processing Letters*, 112(7):277–281, 2012.

[54] H Park and C Chung. An Efficient MapReduce Algorithm for Counting Triangles in a Very Large Graph. In *22nd ACM International Conference on Information & Knowledge Management*, 2013.

[55] H Park, F Silvestri, U Kang, and R Pagh. MapReduce Triangle Enumeration With Guarantees. In *23rd ACM International Conference on Information & Knowledge Management*, 2014.

[56] Y Perez, R Sosič, A Banerjee, R Puttagunta, M Raison, P Shah, and J Leskovec. Ringo: Interactive Graph Analytics on Big-Memory Machines. In *2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[57] A Prat-Pérez, D Dominguez-Sal, J. Brunat, and J Larriba-Pey. Put Three and Three Together: Triangle-Driven Community Detection. *ACM Trans. Knowl. Discov. Data*, 10(3):22:1—-22:42, 2016.

[58] F Radicchi, C Castellano, F Cecconi, V Loreto, and D Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences*, 101(9):2658–2663, 2004.

[59] U Raghavan, R Albert, and S Kumara. Near linear time algorithm to detect community structures in large-scale networks. *CoRR*, abs/0709.2938, 2007.

[60] M Rahman and M Hasan. Approximate triangle counting algorithms on multi-cores. In *2013 IEEE International Conference on Big Data*, 2013.

[61] E Riedy, H Meyerhenke, D Ediger, and D Bader. Parallel community detection for massive graphs. In *9th international conference on Parallel Processing and Applied Mathematics*, 2012.

[62] P Ronhovde and Z Nussinov. Multiresolution community detection for megascale networks by information-based replica correlations. In *Physical Review E*, 2009.

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

[64] V Satuluri, S Parthasarathy, and Y Ruan. Local Graph Sparsification for Scalable Clustering. In *2011 ACM SIGMOD International Conference on Management of Data*, 2011.

[65] T Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, University of Karlsruhe, 2007.

[66] T Schank and D Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms*, 2005.

[67] C Seshadhri, A Pinar, and T Kolda. Triadic measures on graphs: the power of wedge sampling. In *SIAM International Conference on Data Mining*, 2013.

[68] J Shun and K Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, 2015.

[69] SNAP. Stanford Network Analysis Project. `http://snap.stanford.edu`, 2012.

[70] J Soman and A Narang. Fast Community Detection Algorithm with GPUs and Multicore Architectures. In *International Parallel and Distributed Processing Symposium*, 2011.

[71] C Staudt and H Meyerhenke. Engineering High-Performance Community Detection Heuristics for Massive Graphs. In *International Conference on Parallel Processing*, 2013.

[72] S Suri and S Vassilvitskii. Counting triangles and the curse of the last reducer. In *20th international conference on World Wide Web*, 2011.

[73] K Tangwongsan, A Pavan, and S Tirthapura. Parallel Triangle Counting in Massive Streaming Graphs. In *22nd ACM International Conference on Information & Knowledge Management*, 2013.

[74] T Tian and K Burrage. Stochastic models for regulatory networks of the genetic toggle switch. *Proceedings of the National Academy of Sciences*, 103(22):8372–8377, 2006.

[75] C Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *IEEE International Conference on Data Mining*, 2008.

[76] C Tsourakakis, U Kang, G Miller, and C Faloutsos. DOULION: Counting Triangles in Massive Graphs with a Coin. In *15th International Conference on Knowledge Discovery and Data Mining*, 2009.

[77] L Wang, T Lou, J Tang, and J Hopcroft. Detecting Community Kernels in Large Social Networks. In *2011 IEEE 11th International Conference on Data Mining*, 2011.

[78] S Wasserman and K Faust. *Social Network Analysis. Methods and Applications.* Cambridge University Press, 1994.

[79] H. Wilf. Generatingfunctionology. `https://www.math.upenn.edu/~wilf/gfology2.pdf`, 1994.

[80] B Wu, K Yi, and Z Li. Counting Triangles in Large Graphs by Random Sampling. *IEEE Transactions on Knowledge and Data Engineering*, PP(99), 2016.

[81] Y Zhang, J Wang, Y Wang, and L Zhou. Parallel Community Detection on Large Networks with Propinquity Dynamics. In *15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

[82] Zoltan. Sandia National Laboratories. `http://www.cs.sandia.gov/zoltan/`.