HPC-based Parallel Algorithms for Generating Random Networks and Some Other Network Analysis Problems

Md Maksudul Alam

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 Marathe, Chair
Maleq Khan, Co-Chair
Anil Vullikanti
Lenwood S. Heath
Kalyan S. Perumalla

October 13, 2016
Blacksburg, Virginia

Keywords: Network Science, Parallel Algorithms, High Performance Computing, Random Networks
Copyright 2016, Md Maksudul Alam

HPC-based Parallel Algorithms for Generating Random Networks and Some Other Network Analysis Problems

Md Maksudul Alam

(ABSTRACT)

The advancement of modern technologies has resulted in an explosive growth of complex systems, such as the Internet, biological, social, and various infrastructure networks, which have, in turn, contributed to the rise of massive networks. During the past decade, analyzing and mining of these networks has become an emerging research area with many real-world applications. The most relevant problems in this area include: collecting and managing networks, modeling and generating random networks, and developing network mining algorithms. In the era of big data, speed is not an option anymore for the effective analysis of these massive systems, it is an absolute necessity. This motivates the need for parallel algorithms on modern high-performance computing (HPC) systems including multi-core, distributed, and graphics processor units (GPU) based systems. In this dissertation, we present distributed memory parallel algorithms for generating massive random networks and a novel GPU-based algorithm for index searching.

This dissertation is divided into two parts. In Part I, we present parallel algorithms for generating massive random networks using several widely-used models. We design and develop a novel parallel algorithm for generating random networks using the preferential-attachment model. This algorithm can generate networks with billions of edges in just a few minutes using a medium-sized computing cluster. We develop another parallel algorithm for generating random networks with a given sequence of expected degrees. We also design a new a time and space efficient algorithmic method to generate random networks with any degree distributions. This method has been applied to generate random networks using other popular network models, such as block two-level Erdős-Rényi and stochastic block models. Parallel algorithms for network generation pose many nontrivial challenges such as dependency on edges, avoiding duplicate edges, and load balancing. We applied novel techniques to deal with these challenges. All of our algorithms scale very well to a large number of processors and provide almost linear speed-up.

Dealing with a large number of networks collected from a variety of fields requires efficient management systems such as graph databases. Finding a record in those databases is very critical and typically is the main bottleneck for performance. In Part II of the dissertation, we develop a GPU-based parallel algorithm for index searching. Our algorithm achieves the fastest throughput ever reported in the literature for various benchmarks.

HPC-based Parallel Algorithms for Generating Random Networks and Some Other Network Analysis Problems

Md Maksudul Alam

(GENERAL AUDIENCE ABSTRACT)

The advancement of modern technologies has resulted in an explosive growth of complex systems, such as the Internet, biological, social, and various infrastructure networks, which have, in turn, contributed to the rise of massive networks. During the past decade, analyzing and mining of these networks has become an emerging research area with many real-world applications. The most relevant problems in this area include: collecting and managing networks, modeling and generating random networks, and developing network mining algorithms. As the networks are massive in size, we need faster algorithms for the quick and effective analysis of these systems. This motivates the need for parallel algorithms on modern high-performance computing (HPC) based systems. In this dissertation, we present HPC-based parallel algorithms for generating massive random networks and managing large scale network data.

This dissertation is divided into two parts. In Part I, we present parallel algorithms for generating massive random networks using several widely-used models, such as the preferential attachment model, the Chung-Lu model, the block two-level Erdős-Rényi model and the stochastic block model. Our algorithms can generate networks with billions of edges in just a few minutes using a medium-sized HPC-based cluster. We applied novel load balancing techniques to distribute workloads equally among the processors. As a result, all of our algorithms scale very well to a large number of processors and provide almost linear speed-up. In Part II of the dissertation, we develop a parallel algorithm for finding records by given keys. Dealing with a large number of network data collected from a variety of fields requires efficient database management systems such as graph databases. Finding a record in those databases is very critical and typically is the main bottleneck for performance. Our algorithm achieves the fastest data lookup throughput ever reported in the literature for various benchmarks.

Dedication

To my loving family, for their selfless love, support, and encouragement.

Acknowledgments

It was an incredible journey throughout my Ph.D. life at Virginia Tech. Behind the successful completion of this dissertation, there are contribution and support of many people whose names may not all be enumerated. I acknowledge and appreciate their support and express my sincere gratitude.

I would like to acknowledge the supervision of my advisors Dr. Madhav Marathe and Dr. Maleq Khan, for their enthusiasm, constant encouragement, and continuous efforts for guiding me for the past five years. I am particularly thankful to Dr. Madhav Marathe for helping me developing my presentation and writing skills, giving career advice, and providing constructive suggestions regarding my research problems. When I experienced the hardest time during my Ph.D. life, overwhelmed with frustrations and stresses, he offered keen support and proper guidance. He has truly been an inspiration along the journey.

I would like to thank Dr. Maleq Khan for his keen research insight, constant encouragement, and active guidance towards my dissertation. I have learned a lot from his remarkable expertise in distributed systems and rigorous theoretical analytical capability. He spent a substantial amount of time guiding me thoroughly to investigate new ideas, develop efficient solutions, and grasp the analytical abilities for quality writings. This dissertation could not have been completed in time without his continuous support. He also helped me to develop my personal and professional skills. I am honored to have an excellent mentor like him.

I would like to thank the rest of my committee members Dr. Anil Vullikanti, Dr. Lenwood Heath, and Dr. Kalyan Perumalla, for their invaluable feedback and suggestions. I would like to especially thank Dr. Kalyan Perumalla, for taking me as a research intern at the Oak Ridge National Lab and introducing me to GPU programming that resulted into a chapter of my dissertation. I would like to thank Dr. Lenwood Heath for thoroughly reviewing the draft of the dissertation. His insightful comments and suggestions helped in improving the dissertation. I sincerely thank Dr. Anil Vullikanti for offering his thoughtful ideas on

our work, which played a significant role in one of our papers being nominated for the best paper award at the prestigious 2016 Supercomputing conference.

I also acknowledge the support of my colleagues in the Network Dynamics and Simulation Science Laboratory (NDSSL). I am grateful to Dr. Madhav Marathe for providing me the fascinating research opportunities at NDSSL and for funding me as a Graduate Research Assistant throughout my years at Virginia Tech. I thank my lab mates for the wonderful interactions and discussions. Their warm encouragement and support made me never feel alone.

I also acknowledge the contribution of each of my co-authors in this dissertation. Apart from my advisors, Dr. Srikanth Yoginath helped me tremendously to develop my work on GPUs. I also thank the anonymous referees of the papers that were published from this dissertation.

I thank the Bangladeshi community at Virginia Tech for their support, friendship, and for making my stay here a pleasant one. They were always beside me, in good and bad times. I also thank all my friends, who always believed in me and provided constant support.

Last but not the least, I would like to thank my wife for her encouragement and innumerable sacrifices throughout my Ph.D. years. I express my humble gratitude to her for being such a wonderful wife. It would not have been possible without her continuous support, love, and patience. My deepest gratitude to my parents for believing in me and supporting me throughout my whole life. Without their endless blessings, I would not have been able to embark this place.

Contents

Chapter 1: Introduction **1**

1.1 Overview 1

1.2 Large-Scale Random Network Generation 3

1.3 GPU-Based Network Mining Algorithms 5

1.4 Organization of the Dissertation 6

Chapter 2: Preliminaries **7**

2.1 Basic Terminology 7

2.1.1 Networks 7

2.1.2 Simple Network 7

2.1.3 Complete Network 8

2.1.4 Multipartite Network 8

2.1.5 Evolving Network Model 9

2.2 Network Measures 10

2.2.1 Degree Distribution 10

2.2.2 Clustering Coefficient 10

I Distributed Memory Parallel Algorithms for Random Network Generation **11**

Chapter 3: Distributed-Memory Parallel Algorithms for Generating Massive Scale-free Networks Using Preferential Attachment Model **12**

3.1 Introduction 12

3.2 Preliminaries and Notations 13

3.3 Preferential Attachment Model 13

3.3.1 Sequential Algorithms for Preferential Attachment Model 14

3.3.2 Parallel Algorithm for Preferential Attachment Model with $x = 1$ . . 16

3.3.3 Parallel Algorithm with $x \geq 1$ 18

3.3.4 Dependency Chains 19

3.3.5 Waiting Queue 25

3.4 Partitioning and Load Balancing 29

3.4.1 Consecutive Partitioning 30

3.4.2 Round-Robin Partitioning 34

3.4.3 Block Partitioning 35

3.5 Experimental Results 37

3.5.1 Degree Distribution 37

3.5.2 Partitioning and Load Balancing 38

3.5.3 Scalability 41

3.6 Conclusion 42

Chapter 4: Parallel Algorithm for Generating Massive Random Networks with a Given Sequence of Expected Degrees **44**

4.1 Introduction 44

4.2 The Chung-Lu Model and Efficient Sequential Algorithm 45

4.3 A Time Efficient Parallel Algorithm for the CL Model 47

4.3.1 Consecutive Partitioning (CP) 49
4.3.2 Round-Robin Partitioning (RRP) 57
4.4 Space-Efficient Parallel Algorithm for the CL Model 59
4.5 Experimental Results 62
4.5.1 Memory Requirement 66
4.6 Conclusion 69

Chapter 5: An Efficient and Scalable Algorithmic Method for Generating Large-Scale Random Graphs 70

5.1 Introduction 71
5.2 Generating random graphs with a desired degree distribution 72
5.2.1 The Chung-Lu Model 73
5.2.2 DG: A New Time and Memory Efficient Algorithm 74
5.3 Parallelization of the DG Algorithm 80
5.3.1 Task Distribution and Load Balancing 80
5.3.2 Parallel Implementation 84
5.3.3 Experimental Results 87
5.4 Block Two-level Erdős-Rényi 91
5.4.1 A DG-based Algorithm for the BTER Model 93
5.4.2 Experimental Evaluation 93
5.5 Stochastic Block Model 94
5.6 Conclusion 97

II GPU-Based Algorithms for Network Analysis and Data Mining Problems 99

Chapter 6: GPU-based Efficient Index Searching 100

6.1 Introduction 100

6.1.1 In-Memory Databases 100
6.1.2 Index Search Approaches 101
6.1.3 Organization 102
6.2 GPU-based Radix Tree for Index Searching 102
6.2.1 Radix Tree Nodes 103
6.2.2 Benefits and Challenges of Using GPUs 104
6.2.3 Tree Interface and Serialization 104
6.2.4 Query Types 106
6.3 Benchmarks 106
6.3.1 Overview 107
6.3.2 Lookup Benchmarks 108
6.4 Performance Study 109
6.4.1 GPU Overhead 110
6.4.2 Bulk Lookup 111
6.4.3 Bulk Lookup with Real-World Keys 111
6.4.4 Singular and Hierarchical Lookup 112
6.4.5 Range Lookup 113
6.4.6 Update Operations 114
6.4.7 Comparing GRT with Other Index Searching Systems 114
6.5 Conclusion and Future Work 116

Chapter 7: Concluding Remarks **118**

Bibliography **119**

List of Figures

2.1 Example of two networks with 13 vertices each. 8

2.2 A complete network with five vertices. 8

2.3 A bipartite network. 9

2.4 An evolving network G with time $t = 0, 1, 2, 3, \ldots, T$ 9

3.1 A network with 7 vertices generated by Algorithm 1: a) an intermediate instance of the network in the middle of the execution of the algorithm, b) the final network. Solid lines show final resolved edges, and dashed lines show waiting of the vertices. For example, for vertex $t = 4$, k is chosen to be 2, $F(4)$ is chosen to be set to $k = 2$ (in Line 2-5), and thus edge $(4, 2)$ is finalized immediately. For vertex $t = 5$, k is 3 and $F(5)$ is set to be $F(3)$ (in Line 7); as a result, determination of $F(5)$ is waited until $F(3)$ is known. At the end, we have $F(5) = F(3) = F(2) = 1$. 17

3.2 Selection chain and dependency chain. The entire chain, which is marked by the solid lines, is a selection chain $\langle t, k, j, i, 2, 1, 0\rangle$, and the sub-chain marked by the thick solid lines is a dependency chain $\langle t, k, j, i\rangle$. 21

3.3 Experimental result shows that the maximum length of a dependency chain is $O(\log n)$. The horizontal axis (in log scale) represents the number of vertices and the vertical axis represents the length of a dependency chain. Filled circles show the maximum length of a dependency chain for each pair of n and x. The solid line represents a logarithmic fit of the function $y = a \log n + c$. 25

3.4 The maximum size of a waiting queue varies logarithmically with n 28

3.5 The maximum size of a waiting queue varies linearly with x 29

3.6 The maximum size of the waiting queues changes linearly with p 29

3.7 Distribution of the vertices among processors for actual solutions of Equation 3.12 and its linear approximation. 32
3.8 Blocked partitioning with $P = 3$ processors 36
3.9 The degree distribution (in log − log scale) of the network generated by our parallel algorithms. The network is generated with $n = 10^9$ and $x = 4$. . . . 38
3.10 Vertex and message distribution for the partitioning schemes 39
3.11 Change of run time based on block size 40
3.12 Change of run time based on block size 40
3.13 Effect of blocked partitioning on block size 40
3.14 The strong scaling of our parallel algorithms for the problem size $n = 100M$ and $x = 60$. 41
3.15 Weak scaling of our parallel PA algorithm. 42

4.1 Computational cost and run time in naïve CP scheme 50
4.2 Uniform cost partitioning (UCP) scheme 51
4.3 Steps for determining cumulative cost in UCP 52
4.4 The maximum number of partition boundaries in a single processor 55
4.5 Distribution of vertices, edges and messages for space-efficient parallel CL algorithm 61
4.6 Space-efficient parallel CL algorithm with $2P$ subsets 62
4.7 Degree distributions of input and generated degree sequences 64
4.8 Comparison of partitioning schemes 65
4.9 Strong and weak scaling of the parallel algorithms 67
4.10 Strong scaling of the space efficient parallel algorithm 68

5.1 Selecting edges from a sequence of potential edges. The black circles represents the selected edges. 75
5.2 Group V_i using $\mathcal{G}(n, p)$ model with $n_i = 5$ and $p = \frac{d_i^2}{S}$. 76
5.3 Inter edges between V_i and V_j ($n_i = 4$ and $n_j = 2$). 76

5.4 Performance of the sequential algorithms 79
5.5 Relabeling the tasks. The red and blue texts represent the original and new labels of the tasks respectively. 81
5.6 Dividing a task into multiple sub-tasks 83
5.7 Dividing the boundary tasks. The blue and green texts represent the cost boundaries among the processors and the subtask partitions within the task respectively. 83
5.8 Degree distributions of input and generated graph 88
5.9 Strong and weak scaling of the parallel algorithms 89
5.10 Load balancing of the parallel DG algorithm 90
5.11 Degree distributions and average clustering coefficient per degree of input and generated networks using the BTER model 95
5.12 Load Balancing of Parallel BTER. P1 and P2 denotes Phase 1 and Phase 2 respectively. 96
5.13 Strong scaling of the parallel algorithms for BTER 96
5.14 Strong scaling of the parallel algorithm for the SBM 97

6.1 Serializing the radix tree for GPU 105
6.2 Replicating Radix Tree for the GPU 110
6.3 Bulk, Singular, and Hierarchical lookup throughputs for 4-byte keys 111
6.4 Bulk lookup for ISBN (8 Bytes) and Music Id keys (16 Bytes) 112
6.5 Performance of range queries 113
6.6 Comparing GRT with other systems 115

List of Tables

4.1 Networks used in the experiments 63
4.2 KL Divergence Test for the Input and Output Degree Distributions 65
4.3 Required memory (in MB) for parallel algorithm 66
4.4 Effect of average degree on run time 69

5.1 A comparison of the algorithms for the CL model 74
5.2 Number of distinct degrees in real-world graphs 79
5.3 Performance of the sequential algorithm for BTER 94

6.1 Datasets used in the evaluation 107
6.2 List of Benchmarks 108
6.3 Machine configurations of existing index searching systems 115

Chapter 1

Introduction

1.1 Overview

During the past decade, the field of network analysis and mining has become an emerging research area. Networks have become increasingly important in modeling complex structures, such as circuits [28], images [21, 60], chemical compounds [44, 148], protein structures [25], biological networks [13, 63, 127], social networks [86, 92], the Web [54, 138], and XML documents [4, 89]. Various social network theories, network metrics, topology, and mathematical models have been proposed to understand the underlying properties and relationships of those systems. Three of the most crucial issues in this area are: the collection and management of network data [23, 70], the modeling and generation of random networks [15, 19, 30, 31, 39, 51, 71, 93, 94, 123, 153], and the development of network mining algorithms, such as frequent pattern mining [2, 69, 85], clustering and classification [58, 89], centrality analysis [32, 96, 121], triangle counting [87], clustering coefficient [153], eigenvalue [27, 108, 150], community detection [1, 53, 127, 142], information diffusion [67, 81], and epidemic spread [96, 103].

We are observing an explosive growth of complex systems, such as the Internet, biological networks, social networks, and various infrastructure networks due to the advance of modern technologies. The growth of these systems has contributed to the rise of massive networks. It is estimated that, every day, 2.5 quintillion (2.5×10^{18}) bytes of data are generated by approximately 2.5 billion Internet users [146]. We are currently observing the emergence of big data. Big data plays a crucial role in data analytics, computational genomics, network science, and business intelligence and analytics [36]. With the increasing quantities of data, network mining has been an active and important topic in data mining and analytics.

Although many algorithms have been developed for network mining and analysis in the past, most of them are not suitable for very large networks, due to their computational cost and space requirement. For effective analysis of these systems, speed is not an option anymore, it is an absolute necessity. This naturally motivates processing such large data using parallel algorithms. However, it is well known that network algorithms are difficult to scale to a large number of processors. Achieving scalability for these parallel algorithms has emerged as a new frontier of research in network science.

Most network-centric algorithms studied in the literature implicitly assume that the network is stored in main memory. However, many networks are massive in size (i.e., social and Web networks). These are impossible to explicitly store in the main memory of a single machine and sometimes even on the disk. Even if it was possible to use the disk, the algorithm would perform poorly because disk I/O is extremely inefficient in practice compared to main memory access. For this reason, high-performance computing cluster (HPC) based systems, such as multi-core, distributed, and graphics processor units (GPU) based systems, should be used for network algorithms.

In multi-core shared memory systems, all processors have access to a large global shared memory and share the same logical address space to access data directly from memory. As the whole network data is available from a single place, programming is relatively simpler. Also, the challenges associated with network algorithms, such as data partitioning, migration, and load balancing, are reduced in this paradigm. However, shared memory systems are difficult to scale to a large number of processors due to memory contention. Besides, there is a limited availability of shared memory systems with a large number of processors due to difficulties in managing such systems.

In contrast, in distributed memory systems, each processor has its own local memory. Therefore, the memory contention issue does not arise for these systems, and algorithms tend to scale to a large number of processors. Due to the ease of management and maintenance, distributed memory systems with a large number of processors are widely available. However, as data may no longer fit into the memory of a single processor, the algorithms require data partitioning, migration, and communications, all of which are challenging. Load balancing is another problem associated with network data due to the unstructured nature of workloads, which is even more challenging in distributed memory systems. Therefore, efficient parallel algorithms on distributed memory systems have to deal with communication overheads and load balancing issues. Novel techniques are required to deal with these challenges. There are many distributed memory systems available, such as message passing interface (MPI), Hadoop, Map-Reduce, and Giraph. In this dissertation, we use MPI-based systems due to their superior performance.

Graphics processors (GPUs) are highly parallel, multi-threaded, many-core processors and are widely used for general purpose computing. The use of GPUs is prevalent in scientific computation and complex simulations [74, 100, 114, 115]. GPUs are also being increasingly used in big data analytics, machine learning, and data mining [52, 72, 132, 141]. A GPU has some streaming multiprocessors (SMs). An SM is a group of core processors. Each core processor executes only one thread at a time. All core processors can execute their corresponding threads simultaneously. If some threads perform operations with high latencies, those are put into the waiting state and other pending threads are executed. Therefore, GPUs increase throughput by keeping the processors busy. All thread management, including the creation and scheduling of threads, is performed entirely in hardware with virtually zero overhead and requires negligible time. For these advantages, modern supercomputers, such as Titan [112], the largest supercomputer in the USA, are using GPUs in addition to conventional central processing units (CPUs).

In this dissertation, we present parallel algorithms on the problems of generating random networks and a few network management and mining problems such as index searching. For the organization, this dissertation is divided into two parts: Part I focuses on distributed memory parallel algorithms for generating random networks, and Part II focuses on parallel algorithms for network mining problems using GPUs.

1.2 Large-Scale Random Network Generation

Advances in hardware technology, as well as developments in software and algorithms, have enabled the detailed study of complex networks. Complex networks, such as the Internet [54, 138], biological networks [63], social networks [86, 92], and various infrastructure networks [17, 33, 88], are abstracted as random networks for the purposes of obtaining rigorous mathematical results [33]. The study of these complex systems have significantly increased the interest in various random network models such as the Erdős-Rényi (ER) [51], the small-world [153], the Barabási-Albert (BA) [15], the Chung-Lu (CL) [39], the HOT [30], the exponential random graph (ERGM) [123], the recursive matrix (R-MAT) [31], and the stochastic Kronecker graph (SKG) [93, 94] models.

Among the proposed models, the first and the most studied model is the Erdős-Rényi model [51]. However, the Erdős-Rényi model does not exhibit the characteristics observed in many real-world complex systems [19]. Barabási and Albert [15] discovered a class of inhomogeneous networks, called scale-free networks, characterized by a power-law degree distribution $P(k) \propto k^{-\gamma}$, where k represents the degree of a vertex and γ is a constant. While high degree vertices are improbable in Erdős-Rényi networks, they do occur with

statistically significant probability in scale-free networks. Furthermore, the work of Albert, Jeong, and Barabási [9] suggests these high degree vertices appear to play an important role in the behavior of scale-free systems, particularly with respect to their resilience [33].

Watts and Strogatz [153] described small-world networks, which also lead to a relatively homogeneous topology [33]. This model transforms a regular one-dimensional lattice (with vertex degree of four or higher) by rewiring each edge to a randomly chosen vertex, with a certain probability. It has been found that, even with small rewiring probability, the average shortest-path length of the resulting networks is in the order of Erdős-Rényi random networks, and generates networks with fat-tailed degree distributions [163].

The stochastic Kronecker graph (SKG) is a generalization of the recursive matrix (R-MAT) model, used in many network mining applications [31, 93, 94]. The network is generated by the Kronecker multiplication of a small square matrix. The SKG is one of the few models that can generate networks fully in parallel and have interesting properties of real-world networks. For these reasons, it has been included in the Graph500 supercomputer benchmark [65]. The Chung-Lu (CL) model generates random networks from a given sequence of expected degrees [39]. The CL model exhibits properties similar to the SKG model and further has the ability to generate a wider range of degree distributions [117].

In the era of big data, the advance of modern technologies is causing a rapid growth of complex systems. As some of the complex networks grow, it has become necessary to generate massive random networks efficiently. As discussed in [95], a smaller network may not exhibit the same behavior even if both networks are generated using the same model. In [95], by experimental analysis, it was shown that the structure of larger networks is fundamentally different from that of small networks, and many patterns emerge only in massive networks. In the areas of network science and data mining as well as social sciences and physics, large-scale network analysis is becoming a significant research area [12].

The advent of large random networks necessitates efficient algorithms to generate such networks, both in terms of running time and memory consumption. Although various random network models have been used and studied over the last several decades, even efficient sequential algorithms for generating such networks were nonexistent until recently. Batagelj and Brandes [19] justifiably said:

> *"To our surprise we have found that the algorithms used for these generators in software such as BRITE, GT-ITM, JUNG, or LEDA are rather inefficient. . . . superlinear algorithms are sometimes tolerable in the analysis of networks with tens of thousands of vertices, but they are clearly unacceptable for generating large numbers of such graphs."*

As a step towards meeting this goal, recently, efficient sequential algorithms have been developed to generate certain classes of random networks: the Erdős-Rényi [19, 109], small world [19], preferential attachment [19], and Chung-Lu [105] models.

However, although efficient sequential algorithms are able to generate networks with millions of vertices quickly, generating networks with billions of vertices can take prohibitively large amounts of time. Furthermore, a large memory requirement often makes the generation of such large networks using these sequential algorithms infeasible. We have developed distributed memory parallel algorithms for generating massive random networks using the preferential attachment [8], Chung-Lu [7], and small-world models.

The design of parallel distributed memory algorithms poses two main challenges in the context of generating random networks. First, the dependencies among the edges, especially in the preferential attachment model, impede independent operations of the processors. Second, different processors can create duplicate edges, which must be avoided. Dealing with both of these problems requires complex synchronization and communications among the processors, and thus gaining satisfactory speed up by parallelization is a challenging problem. Even for the Erdős-Rényi model, where the existence of edges is independent of each other, the parallelization of a non-naïve efficient algorithm, such as the algorithm by Batagelj and Brandes [19], is a non-trivial problem. A parallelization of Batagelj and Brandes's algorithm was recently proposed in [109].

Our preferential attachment based algorithm has generated random networks with 400 billion edges in five minutes using a medium sized computing cluster [135]. The algorithm using the Chung-Lu model has generated a network with 250 billion edges in just one minute.

1.3 GPU-Based Network Mining Algorithms

Recently, there has been a keen interest in using GPUs for many data and network analytics problems. Recent works include computing eigenvalues [14, 41, 126, 139, 143, 144, 147], computing centrality measures [47, 104, 130], counting triangles [34, 47, 66], subgraph mining [77, 98, 145, 160], graph traversal [75, 130], clustering [155], PageRank [37, 50, 84, 125, 131, 164], and simulation [114, 115]. Several GPU-based graph processing libraries have been developed [5, 61, 133, 137, 151, 165].

Dealing with a large number of networks collected from a variety of fields requires efficient management systems such as graph databases [3, 45]. Many graph databases have been developed recently [11, 24, 154]. In the age of big data, modern systems have to support

high throughput, low latency data lookup. Traditional disk-based systems do not meet these requirements. The development of memory capacities has prompted the advance of in-memory database systems that can support these requirements. Many commercial in-memory systems have been developed recently, such as HyPer [76], SAP HANA [56], and Microsoft Hekathon [46]. These systems are used heavily in data intensive tasks in the fields of information retrieval, machine learning, data mining and analysis [72, 132, 141]. One of the most important problems of these systems is to find the record index in the database from a given key. This problem is the main bottleneck for the performance of many database systems [157, 164]. It has been shown that the majority of the time for a GET query operation in a traditional in-memory database system is spent on searching the index [164]. Therefore, the best performance of such systems depends on efficient index searching. In this dissertation, we present an efficient parallel index searching algorithm using GPUs.

1.4 Organization of the Dissertation

The remainder of the dissertation proposal is organized as follows. In Chapter 3, we present a distributed-memory parallel algorithm for generating massive scale-free networks using the preferential attachment model. In Chapter 4, we present another parallel algorithm for generating massive random networks with a given sequence of expected degrees. We also present a novel parallel partitioning algorithm to achieve good load-balancing. In Chapter 5, we present an efficient and scalable algorithmic approach for generating large-scale random networks for popular network models such as the Chung-Lu [39], block two-level Erdős-Rényi [134], stochastic block [73], and joint degree distribution [140] models efficiently. In Chapter 6, we present an efficient index searching system.

Chapter 2

Preliminaries

In this chapter, we define some basic terminology of network theory. Definitions not included in this chapter will be introduced later as they are needed. We start, in Section 2.1, by giving some definitions of standard network-theoretical terms used throughout the remainder of this dissertation.

2.1 Basic Terminology

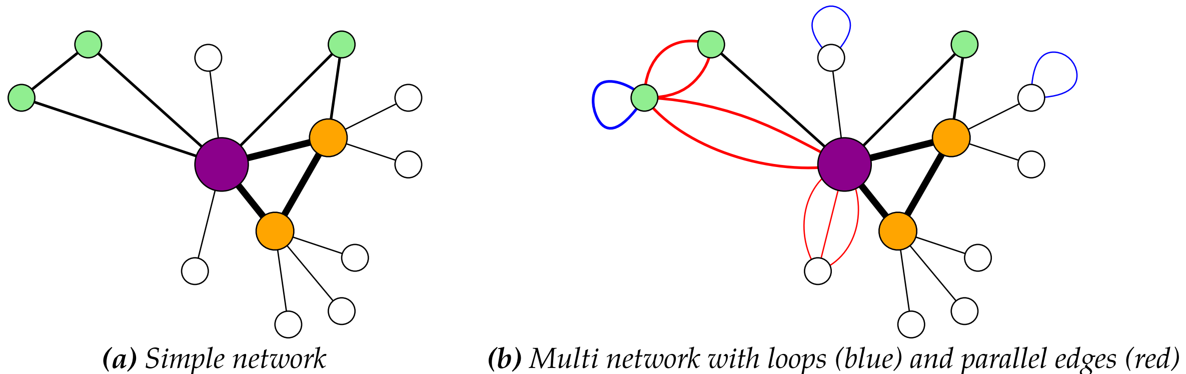
In this section, we define some technical terms used throughout the remainder of this dissertation. Interested readers are referred to detailed texts of the literature [156, 158].

2.1.1 Networks

A *network* is an ordered pair $G(V, E)$ that consists of a finite set of *vertices* V and a finite set of *edges* E. An *edge* is an unordered pair of vertices $\{u, v\}$, where u and v are vertices in V. An edge connecting vertices u and v in V is denoted by (u, v). If $(u, v) \in E$, then two vertices u and v are said to be *adjacent* in G; edge (u, v) is then said to be *incident* to vertices u and v.

2.1.2 Simple Network

A *loop* (also called a *self-loop*) is an edge that connects a vertex to itself. In a graph, *parallel edges* (also called a *multi-edge*) are two or more edges that are incident to the same two



(a) *Simple network* **(b)** *Multi network with loops (blue) and parallel edges (red)*

Figure 2.1: *Example of two networks with* 13 *vertices each.*

vertices. A network G is called a *simple network* if there is no loop or parallel edges between any two vertices in G. Networks with loops or parallel edges are called *multi networks*. In this dissertation, all networks are considered to be simple networks. Figure 2.1 depicts examples of simple and multi networks.

2.1.3 Complete Network

A *complete network* is a simple network whose vertices are pairwise adjacent. The complete network with n vertices is denoted by K_n. Figure 2.2 is an example of a complete network with five vertices.

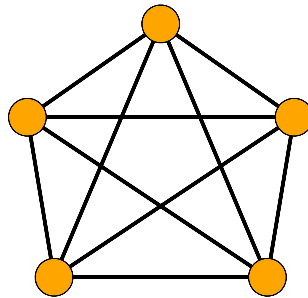


Figure 2.2: *A complete network with five vertices.*

2.1.4 Multipartite Network

A network $G(V, E)$ is k-partite if V can be partitioned in k disjoint subsets $V_1, V_2, \ldots V_k$ such that two vertices from the same subset are not adjacent. That is, for a subset of vertices

V_i where $1 \leq i \leq k$, for every two vertices in V_i, there is no edge connecting the two. When $k = 2$, it is called a bipartite network. A *complete k-partite network* is a simple network such that two vertices are adjacent if and only if they are in different partite sets. Figure 2.3 is an example of two partite (bipartite) network.

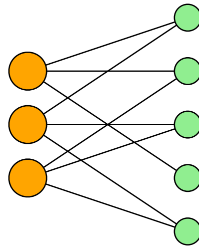


Figure 2.3: *A bipartite network.*

2.1.5 Evolving Network Model

Let $G(V, E)$ denote a network with vertex set V and edge set E. A network is called an *evolving network* if it changes as a function of time [83]. The changes can be done by adding or removing vertices or edges over time. Almost all real world networks evolve over time. Let the network evolves over discrete time steps $t = 0, 1, 2, \ldots, T$. Also, let the partial network at time step t be $G_t(V_t, E_t)$ where V_t is the set of vertices and E_t is the set of edges at time step t. Thus, $\{G_0, G_1, \ldots G_T\}$ is a sequence of networks indexed by time. Since each partial network in the sequence $\{G_0, G_1, \ldots G_T\}$ represents the same network G at different time steps, we have $V_t \subset V$ and $E_t \subset E$.

For simplicity, we assume that with each time step a number of vertices is added to the network. Let $f_v(V_t, t)$ be a function that returns the number of vertices to be added at time $t + 1$; therefore, $|V_{t+1}| = |V_t| + f_v(V_t, t)$. The new vertices make edges by connecting to the existing vertices. Let $f_e\left(f_v, G_t, t\right)$ be a stochastic process that returns a set of edges to be added at time $t + 1$. Therefore, $E_{t+1} = E_t \cup f_e\left(f_v, G_t, t\right)$. An evolving network model is

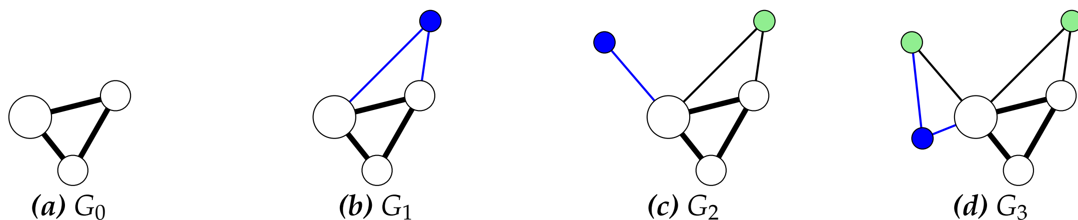


Figure 2.4: *An evolving network G with time $t = 0, 1, 2, 3, \ldots, T$*

sufficiently described by the functions $\langle f_v, f_e \rangle$. Figure 2.4 illustrates an evolving network with time.

2.2 Network Measures

While generating random networks, we are mainly concerned with the following two measures (or properties) of the networks: degree distribution and clustering coefficient.

2.2.1 Degree Distribution

Let $G(V, E)$ be a network where V is the set of vertices and E is the set of edges. If $(u, v) \in E$, then vertices u and v are called *neighbors* of each other. Let $\mathcal{N}_u$ be the set of all neighbors of vertex $u \in V$, i.e., $\mathcal{N}_u = \{v \in V | (u, v) \in E\}$. The *degree* of a vertex in G is the number of edges incident to it in G. In other words, the degree of a vertex is the number of it's neighbors. Let d_u denote the degree of vertex u. Therefore, $d_u = |\mathcal{N}_u|$.

For any integer $k \geq 0$, let n_k be the number of vertices having degree k. Then, the *degree distribution* of a network is defined by the set $\{n_k | k \geq 0\}$. Note that a randomly chosen vertex in the network has degree k with probability $\frac{n_k}{|V|}$.

2.2.2 Clustering Coefficient

The *clustering coefficient* C_u of a vertex u is defined as the ratio of the number of edges among its neighbors to the maximum number of all possible such edges. More formally, the clustering of a vertex u is given by

$$C_u = \frac{\left|\{(v, w) \in E | v, w \in \mathcal{N}_u\}\right|}{\binom{d_u}{2}}$$

where $\mathcal{N}_u$ is the set of neighbors, and d_u is the degree of u.

The *average clustering coefficient* of a network is the average of the local clustering coefficients of all the vertices. Let $\bar{C}$ be the average clustering coefficient of a network $G(V, E)$. Then, we have

$$\bar{C} = \frac{1}{n} \sum_{i=1}^{n} C_i.$$

Part I

Distributed Memory Parallel Algorithms for Random Network Generation

Chapter 3

Distributed-Memory Parallel Algorithms for Generating Massive Scale-free Networks Using Preferential Attachment Model

3.1 Introduction

Preferential attachment is a model that generates random scale-free networks, where a new vertex makes connections to some existing vertices that are chosen preferentially based on some of the properties of those vertices. For the preferential attachment model, the only previously known distributed-memory parallel algorithm is given by Yoo and Henderson [163]. Although useful, the algorithm has two weaknesses: (*i*) to deal with dependencies and the required complex synchronization, they came up with an approximation algorithm rather than an exact algorithm; and (*ii*) the accuracy of their algorithm depends on several control parameters, which are manually adjusted by running the algorithm repeatedly. Several other studies were done on the preferential attachment based models. Machta and Machta [101] described how an evolving network can be generated in parallel. Dorogovtsev, Mendes, and Samukhin [49] proposed a model that can generate networks with fat-tailed degree distributions, i.e., the networks contain some high degree vertices. In this model, starting with a random network, edges are randomly rewired according to some preferential choices.

In this chapter, we study the problem of designing a distributed memory parallel algorithm

for generating massive scale-free networks based on the preferential attachment (PA) model. To the best of our knowledge, our algorithms are the first distributed-memory parallel algorithms for generating random networks while exactly following the preferential attachment model.

The rest of the chapter is organized as follows. Preliminaries, notations, and a description of the parallel computation model are given in Section 3.2. In Section 3.3, we describe the problem and algorithms. Some sequential algorithms are discussed in Section 3.3.1. In Section 3.3.2, we present our parallel algorithm for distributed memory architecture for the case where each vertex connects a single edge to the existing network. In Section 3.3.3, we extend the algorithm for the general case where each vertex contributes x edges to the existing network. Experimental results showing the performance of our parallel algorithms are presented in Section 3.5. Finally, we conclude in Section 3.6.

3.2 Preliminaries and Notations

In the rest of the chapter, we use the following notations. We denote a network $G(V, E)$, where V and E are the sets of vertices and edges, respectively, with $m = |E|$ edges and $n = |V|$ vertices 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 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 $d_v = |N(v)|$. If u and v are neighbors, sometime we say that u is *connected* to v and vice versa.

We develop parallel algorithms for the message passing interface (MPI) based distributed memory system, where the processors do not have any shared memory and each processor has its own local memory. The processors can exchange data and communicate with each other by exchanging messages. The processors have a shared distributed file system from which they read and write data files. However, such reading and writing of the files are done independently and concurrently in each processor.

We use K, M, and B to denote thousands, millions and billions, respectively; e.g., 2B stands for two billion.

3.3 Preferential Attachment Model

The preferential attachment model is a model for generating random evolving scale-free networks using a preferential attachment mechanism. In a preferential attachment

mechanism, a new vertex is added to the network and connected to some existing vertices that are chosen preferentially based on some properties of the vertices. In the most common application, preference is given to vertices with larger degrees: the higher the degree of a vertex, the higher the probability of choosing it. In this chapter, we study only degree-based preferential attachment and, in the rest of the chapter, by preferential attachment (PA), we mean degree-based preferential attachment.

Before presenting our parallel algorithms for generating PA networks, we briefly discuss the sequential algorithms for the same.

3.3.1 Sequential Algorithms for Preferential Attachment Model

Barabási-Albert Model. One way to generate a random PA network is to use the Barabási-Albert (BA) model. Many real-world networks have two important characteristics: (*i*) they are evolving in nature and (*ii*) the networks tend to be scale free [15]. They provided a model, known as the Barabási-Albert (BA) model, where a new vertex is connected to an existing vertex that is chosen with probability directly proportional to its current degree.

The BA model works as follows. Starting with a small clique of $\hat{x}$ vertices, in every time step, a new vertex t is added to the network and connected to $x \le \hat{x}$ randomly chosen existing vertices: $F_k(t)$ for $1 \le k \le x$ with $F_k(t) < t$; that is, $F_k(t)$ denotes the k-th vertex which t is connected to. Thus each time step adds x new edges $(t, F_1(t)), (t, F_2(t)), \ldots, (t, F_x(t))$ to the network, which exhibits the evolving nature of the model. For each of the x new edges, vertices $F_1(t), F_2(t), \ldots, F_x(t)$ are randomly selected based on the degrees of the vertices in the current network. In particular, the probability $P_i(t)$ that vertex t is connected to vertex $i < t$ is given by $P_i(t) = \frac{d_i}{\sum_j d_j}$, where d_j represents the degree of vertex j.

The networks generated by the BA model are called BA networks, which bear those two characteristics of a real-world network. BA networks have a power law degree distribution. A degree distribution follows a power law distribution if the probability that a vertex has degree d is given by $\Pr[d] \propto d^{-\gamma}$, where γ is a positive constant. Barabási and Albert showed this preferential attachment method of selecting vertices results in a power-law degree distribution [15].

First, we assume $x = 1$, and for this case, we use $F(t)$ for $F_1(t)$. We discuss the general case $x \ge 1$ later. A naïve implementation of the above algorithm can take $\Omega(n^2)$ time. One naïve approach is to maintain a list of the degrees of the vertices, and in each time time step t, generate a uniform random number in $\left[1, \sum_{i=0}^{t-1} d_i\right]$ and scan the list of the degrees sequentially to find $F(t)$. In this case, time step t takes $\Theta(t)$ time, and the total time is

$\Omega(n^2)$. Batagelj and Brandes give an efficient algorithm with running time $O(m)$ [19]. This algorithm maintains a list of vertices such that each vertex i appears in this list exactly d_i times. The list can easily be updated dynamically by simply appending u and v to the list whenever a new edge (u, v) is added to the network. Now to find $F(t)$, a vertex is chosen from the list uniformly at random. Since each vertex i occurs exactly d_i times in the list, we have $\Pr[F(t) = i] = \frac{d_i}{\sum_j d_j}$. A sequential implementation of this algorithm is given in the network algorithm library NetworkX [68].

Copy Model. As it turns out, none of the above algorithms lead to an efficient parallelization. Another algorithm, called the *copy model*, proposed in [79, 83] also leads to preferential attachment and a power law degree distribution. The algorithm works as follows. In each time step t,

Step 1: first a random vertex $k \in [1, t-1]$ is chosen with uniform probability.

Step 2: then $F(t)$ is determined as follows:

$$F(t) = k \text{ with probability } p \qquad \text{(Direct Edge)} \tag{3.1}$$

$$= F(k) \text{ with probability } (1-p) \qquad \text{(Copy Edge)} \tag{3.2}$$

It can be easily shown that $\Pr[F(t) = i] = \frac{d_i}{\sum_j d_j}$ when $p = \frac{1}{2}$. Thus when $p = \frac{1}{2}$, this algorithm follows the Barabási-Albert model as shown in Theorem 1.

Theorem 1. *The Barabási-Albert model is a special case of the copy model when $p = \frac{1}{2}$.*

Proof. It can be easily shown that $\Pr[F(t) = i] = \frac{d_i}{\sum_j d_j}$ when $p = \frac{1}{2}$. $F(t)$ can be equal to i in two mutually exclusive ways: i) i is chosen in the first step and assigned to $F(t)$ in the second step (Equation 3.1); this event occurs with probability $\frac{1}{t-1} \cdot p$; or ii) a neighbor of i, $v \in \{u | F(u) = i\}$ is chosen in the first step, and $F(v)$ is assigned to $F(t)$ in the second step (Equation 3.2); this event occurs with probability $\frac{d_i - 1}{t-1} \cdot (1-p)$. Thus we have

$$\begin{aligned} \Pr[F(t) = i] &= \frac{1}{t-1} \cdot p + \frac{d_i - 1}{t-1} \cdot (1-p) \\ &= \frac{p + (d_i - 1)(1-p)}{\frac{1}{2} \sum_j d_j} \end{aligned} \tag{3.3}$$

When $p = \frac{1}{2}$, we have $\Pr[F(t) = i] = \frac{d_i}{\sum_j d_j}$. □

Thus, the copy model is more general than the BA model. In [83], it has been shown that the copy model produces networks with degree distribution following a power law $d^{-\gamma}$,

where the value of the exponent γ depends on the choice of p. Further, it is easy to see the running time of the copy model is $O(m)$, and we found that the copy model leads to more efficient parallel algorithms for generating preferential attachment networks. We develop our parallel algorithm based on the copy model.

3.3.2 Parallel Algorithm for Preferential Attachment Model with $x = 1$

The dependencies among the edges pose a major challenge in parallelizing preferential attachment algorithms. In phase t, to determine $F(t)$, it requires that F_i is known for each $i < t$. As a result, any algorithm for preferential attachment seems to be highly sequential in nature: phase t cannot be executed until all previous phases are completed. However, a careful observation reveals that $F(t)$ can be partially, or sometimes completely, determined even before completing the previous phases. The copy model helps us exploit this observation in designing a parallel algorithm. However, it requires complex synchronization and communication among the processors. To keep the algorithm efficient, such synchronization and communication must be done carefully. In this section, we present a parallel algorithm based on the copy model. For ease of discussion, we first present our algorithm for the case $x = 1$. We present the general case $x \geq 1$ in Section 3.3.3.

Let P be the number of processors. The set of vertices V is partitioned into P disjoint subsets of vertices $V_0, V_1, \ldots, V_{P-1}$; that is, $V_i \subset V$, such that for any i and j, $V_i \cap V_j = \emptyset$ and $\bigcup_i V_i = V$. Processor $\mathcal{P}_i$ is responsible for computing and storing $F(t)$ for all $t \in V_i$. The load balancing and performance of the algorithm crucially depend on how V is partitioned. The details of vertex partitioning are presented in Section 3.4.

The basic principle behind our parallel algorithm is as follows. Recall the sequential algorithm for the copy model. Each processor $\mathcal{P}_i$ can independently compute step 1 for each $t \in V_i$, as a random $k \in [1, t-1]$ is chosen with uniform probability (independent of the vertex degrees). Also, in step 2, if $F(t)$ is chosen to be k, $F(t)$ is determined immediately. If $F(t)$ is chosen to be $F(k)$, determination of $F(t)$ needs to wait until $F(k)$ is known. If $k \in V_j$ where $i \neq j$, processor $\mathcal{P}_i$ sends a *request* message to processor $\mathcal{P}_j$ to find $F(k)$. Note that, at the time when processor $\mathcal{P}_j$ receives this message, $F(k)$ can still be unknown. If so, $\mathcal{P}_j$ keeps this message in a queue until $F(k)$ is known. Once $F(k)$ is known, $\mathcal{P}_j$ sends back a *resolved* message to $\mathcal{P}_i$. The basic method executed by a processor $\mathcal{P}_i$ is given in Algorithm 1. An example instance of the execution of this algorithm with seven vertices is depicted in Figure 3.1.

Algorithm 1: Parallel PA with $x = 1$

```
/* Each processor P_i executes the following in parallel:              */
1  foreach t ∈ V_i do
2      k ← a uniform random vertex in [1, t − 1]
3      c ← a uniform random number in [0, 1]
4      if c < p then                                // i.e., with probability p
5          F(t) ← k
6      else
7          F(t) ← NIL                                // to be set later to F(k)
8          send message ⟨request, t, k⟩ to P_j, where k ∈ V_j

/* Next, processor P_i receives messages and processes them as follows:  */
9  Upon receipt of message ⟨request, t′, k′⟩ from P′_j:    // note that k′ ∈ V_i
10 if F(k′) ≠ NIL then
11     send message ⟨resolved, t′, F(k′)⟩ to P′_j
12 else
13     store t′ in queue Q_k′
14 Upon receipt of message ⟨resolved, t, v⟩:
15 F(t) ← v
16 foreach t′ ∈ Q_t do
17     send message ⟨resolved, t′, v⟩ to P_j where t′ ∈ V_j
```

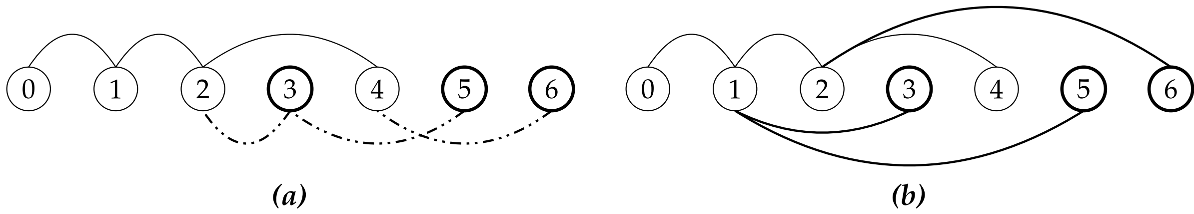


Figure 3.1: *A network with 7 vertices generated by Algorithm 1: a) an intermediate instance of the network in the middle of the execution of the algorithm, b) the final network. Solid lines show final resolved edges, and dashed lines show waiting of the vertices. For example, for vertex $t = 4$, k is chosen to be 2, $F(4)$ is chosen to be set to $k = 2$ (in Line 2-5), and thus edge $(4, 2)$ is finalized immediately. For vertex $t = 5$, k is 3 and $F(5)$ is set to be $F(3)$ (in Line 7); as a result, determination of $F(5)$ is waited until $F(3)$ is known. At the end, we have $F(5) = F(3) = F(2) = 1$.*

3.3.3 Parallel Algorithm with $x \geq 1$

In Section 3.3.2, we presented the algorithm for the simpler case $x = 1$. In this section, we modify this algorithm for the general case where each vertex creates $x \geq 1$ edges. The pseudocode of the algorithm is given in Algorithm 2. The basic structure of the algorithm for the general case is the same as that of the special case $x = 1$. We focus our discussion only on the modifications required and the differences between the two cases. The main difference is that, for each vertex t, instead of computing one edge $(t, F(t))$, we need to compute x edges $(t, F_1(t)), (t, F_2(t)), \ldots, (t, F_x(t))$, and make sure such edges are distinct and do not create any parallel edges. For this general case, the set of vertices $\{F_1(t), F_2(t), \ldots, F_x(t)\}$ is denoted by $\mathbb{F}(t)$.

The algorithm starts with an initial network, which is a clique of the first x vertices labeled $0, 1, 2, \ldots, x-1$. Each of the other vertices from x to $n-1$ generates x new edges. There are fundamentally two important issues that need to be handled for the general case: i) how we select $F_\ell(t)$ for vertex t where $1 \leq \ell \leq x$, and ii) how we avoid duplicate edge creation. Multiple edges for a vertex t are created by repeating the same procedure x times (Line 2), and duplicate edges are avoided by simply checking if such an edge already exists - such checking is done whenever a new edge is created.

For the ℓ-th edge of a vertex t, another vertex k is chosen from $[x, t-1]$ uniformly at random (Line 3). Edge (t, k) is created with probability p (Line 5). However, before creating such an edge (t, k) in Line 7, the existence of such an edge is checked immediately before creating them in Line 6. If the edge already exists at that time, the process is repeated again (Line 9). With the remaining $1-p$ probability, t is connected to some vertex in $\mathbb{F}(k)$; that is, we make an edge $(t, F_\ell(k))$, such that ℓ is chosen from $[1, x]$ uniformly at random. Similar to the special case $x = 1$, if k is in another processor, a request message is sent to that processor to find $F_\ell(k)$ (Line 14). The request and response messages are also processed in the same way.

Duplicate edges can also be created during the execution of Line 19. For example, suppose vertex t creates two edges $(t, F_\ell(k))$ and $(t, F_{\ell'}(k'))$. Also, assume both k and k' are not in the same processor as t. Hence, request messages are sent to the processors containing k and k' to resolve $F_\ell(k)$ and $F_{\ell'}(k')$. If the ℓ-th edge of k and ℓ'-th edge of k' both connect to the same vertex u, then $F_\ell(k) = F_{\ell'}(k') = u$. Hence, t may create a duplicate edge (t, u) which could not be detected early. To deal with such duplicate edges, after receiving a resolved message $\langle \texttt{resolved}, F_\ell(t), v \rangle$, the adjacency list of t is checked to find whether edge (t, v) already exists (Line 20). If the edge does not exist, it is created. Otherwise, new k and ℓ are selected (Line 25-26), and a new request message is sent (Line 27).

Algorithm 2: Parallel PA with $x \geq 1$

```
/* Each processor P_i executes the following in parallel:                */
1  foreach t ∈ V_i do
2      for ℓ = 1 to x do
3          k ← a uniform random vertex in [1, t − 1]
4          c ← a uniform random number in [0, 1]
5          if c < p then                               // i.e., with probability p
6              if k ∉ 𝔽(t) then
7                  F_ℓ(t) ← k
8              else
9                  go to line 3
10         else
11             l ← a uniform random number in [1, x]
12             F_ℓ(t) ← NIL                            // to be set later to F_l(k)
13             send message ⟨request, F_ℓ(t), F_l(k)⟩ to P_j, where k ∈ V_j

/* Next, processor P_i receives messages and processes them as follows:  */
14 Upon receipt of message ⟨request, F_ℓ'(t'), F_l'(k')⟩ from P_j':   // note that k' ∈ V_i
15 if F_l'(k') ≠ NIL then
16     send message ⟨resolved, F_ℓ'(t'), F_l'(k')⟩ to P_j'
17 else
18     store ⟨F_ℓ'(t'), F_l'(k')⟩ in queue Q_k'
19 Upon receipt of message ⟨resolved, F_ℓ(t), v⟩:
20 if v ∉ 𝔽(t) then
21     F_ℓ(t) ← v
22     foreach ⟨F_ℓ'(t'), F_ℓ(t)⟩ ∈ Q_t do
23         send message ⟨resolved, F_ℓ'(t'), v⟩ to P_j where t' ∈ V_j
24 else
25     k ← a uniform random vertex in [x, t − 1]
26     l ← a uniform random number in [1, x]
27     re-send message ⟨request, F_ℓ(t), F_l(k)⟩ to P_j, where k ∈ V_j
```

3.3.4 Dependency Chains

In our parallel algorithm, it is possible that computation of $F(t)$ for some vertex t can wait until $F(k)$ for some other vertex k is known. Such waiting can form a chain, namely

a *dependency chain*. For example, as demonstrated in Figure 3.1, computation of $F(5)$ is waiting for $F(3)$, which in turn is waiting for $F(2)$, and thus we have a chain of dependency $\langle 5, 3, 2\rangle$. If the lengths of these chains are large, the waiting period for some vertices can be quite long, leading to poor performance of the parallel algorithm. Fortunately, the length of a dependency chain is small, and the performance of the algorithm is hardly affected by such waiting.

Dependency chain for $x = 1$. For the ease of analysis, first we formally define a dependency chain for $x = 1$ and provide a rigorous analysis showing that the maximum length of a dependency chain is at most $O(\log n)$ with high probability (w.h.p.). For large n, $O(\log n)$ is small compared to n. Moreover, while $O(\log n)$ is the maximum length, most of the chains have much smaller length. It is easy to see that for a constant p, the average length of a dependency chain is also constant, which is at most $\frac{1}{p}$. For an arbitrary p, the average length is still bounded by $\log n$ as shown in Theorem 4. Thus, while for some vertices a processor may need to wait for $O(\log n)$ steps, the processor hardly remains idle as it has other vertices to work with.

For the purpose of analysis, first we introduce another chain named a *selection chain*. In the first step (Line 2 of Algorithm 1), for each vertex t, another vertex $k \in [1, t-1]$ is selected. In turn for vertex k, another vertex in $[1, k-1]$ is selected. We can think that such a selection process creates a chain called a *selection chain*. Formally, we define a selection chain S_t starting at vertex t to be a sequence of vertices $\langle u_0, u_1, u_2, \ldots, u_i, \ldots u_x\rangle$ such that $u_0 = t$, $u_x = 1$, and u_{i+1} is selected for vertex u_i for $0 \le i < x$. Notice that a selection chain must end at vertex 1. The length of a selection chain S_t denoted by $|S_t|$ is the number of vertices in S_t.

In the next step (see Equation 3.2 and Line 2-5 of Algorithm 1), $F(t)$ is computed by assigning k or $F(k)$ to it. If $F(k)$ is selected to be assigned to $F(t)$, $F(t)$ cannot be determined until $F(k)$ is known; that is, the computation of $F(t)$ for vertex t depends on vertex k. In such a case, we say vertex t is dependent on k; otherwise, we say vertex t is independent. In turn, vertex k can depend on some other vertex, and eventually such successive dependencies can form a dependency chain. Formally, a *dependency chain D_t* starting at vertex t is a sequence of vertices $\langle v_0, v_1, v_2, \ldots, v_i, \ldots v_y\rangle$ such that $v_0 = t$, v_i depends on v_{i+1} for $0 \le i < y$, and v_y is independent. Notice that if $v_i \in D_t$, D_{v_i} is a subsequence and a suffix of D_t. Also it is easy to see that D_t is a subsequence and a prefix of S_t, and we have $|D_t| \le |S_t|$. Examples of a selection chain and a dependency chain are shown in Figure 3.2. Bounds on the length of dependency chains are given in Theorem 4. The following lemmas, Lemma 2 and 3, are needed to prove Theorem 4.

Lemma 2. *Let $P_t(i)$ be the probability that vertex i is in selection chain S_t starting at vertex t.*

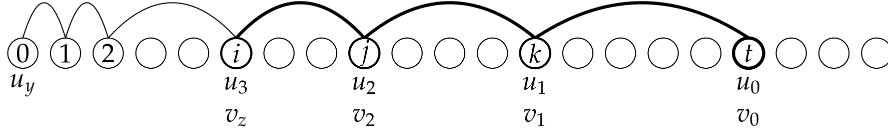


Figure 3.2: *Selection chain and dependency chain. The entire chain, which is marked by the solid lines, is a selection chain* $\langle t, k, j, i, 2, 1, 0\rangle$*, and the sub-chain marked by the thick solid lines is a dependency chain* $\langle t, k, j, i\rangle$*.*

Then for any $1 \le i < t$*,* $P_t(i) = \frac{1}{i}$*.*

Proof. Vertex i can be in S_t in two ways: a) vertex i is selected for t (in Line 2 of Algorithm 1); the probability of such an event is $\frac{1}{(t-1)}$; b) vertex k is selected for t, where $i < k < t$, with probability $\frac{1}{(t-1)}$, and i is in S_k. Hence, for $1 \le i < t$, we have

$$P_t(i) = \frac{1}{t-1} + \sum_{k=i+1}^{t-1} \frac{1}{t-1} \Pr[i \in S_k]$$

$$(t-1)P_t(i) = 1 + \sum_{k=i+1}^{t-1} P_k(i) \tag{3.4}$$

Substituting t with $t+1$, for any i with $1 \le i < t+1$, we have

$$tP_{t+1}(i) = 1 + \sum_{k=i+1}^{t} P_k(i) \tag{3.5}$$

By subtracting Equation 3.4 from Equation 3.5,

$$tP_{t+1}(i) - (t-1)P_t(i) = P_t(i)$$

$$P_{t+1}(i) = P_t(i) \tag{3.6}$$

From Equation 3.6 by induction, we have $P_k(i) = P_t(i)$ for any k and t such that $1 \le i < \min\{k, t\}$. Now consider $k = i+1$. Notice that i is in S_{i+1} if and only if i is selected for vertex $i+1$; that is, $P_{t+1}(i) = \frac{1}{i}$. Hence, for any $t > i$, we have

$$P_t(i) = \frac{1}{i}.$$

□

Lemma 3. *Let* A_i *denote the event that* $i \in S_t$*. Then the events* A_i *for all* i*, where* $1 \le i < t$*, are mutually independent.*

Proof. Consider a subset $\{A_{i_1}, A_{i_2}, \ldots, A_{i_\ell}\}$ of any ℓ such events where $i_1 < i_2 < \ldots < i_\ell$. To prove the lemma, it is necessary and sufficient to show that for any ℓ with $2 \leq \ell < t$,

$$\Pr\left[\bigcap_{k=1}^{\ell} A_{i_k}\right] = \prod_{k=1}^{\ell} \Pr\left[A_{i_k}\right]. \tag{3.7}$$

We know

$$\Pr\left[\bigcap_{k=1}^{\ell} A_{i_k}\right] = \Pr\left[A_{i_1}\middle|\bigcap_{k=2}^{\ell} A_{i_k}\right] \cdot \Pr\left[\bigcap_{k=2}^{\ell} A_{i_k}\right]$$

When it is given that $\bigcap_{k=2}^{\ell} A_{i_k}$, i.e., $i_2, \ldots, i_\ell \in S_t$, by the constructions of selection chains S_{i_2} and S_t and since $i_1 < i_2$, we have $i_1 \in S_t$ if and only if $i_1 \in S_{i_2}$. Then

$$\Pr\left[A_{i_1}\middle|\bigcap_{k=2}^{\ell} A_{i_k}\right] = \Pr\left[i_1 \in S_{i_2}\middle|\bigcap_{k=2}^{\ell} A_{i_k}\right].$$

Let R_i be a random variable that denotes the random vertex selected for vertex i. Now observe that the occurrence of event $i_1 \in S_{i_2}$ can be fully determined by the variables in $\{R_j \mid i_1 < j \leq i_2\}$; that is, event $i_1 \in S_{i_2}$ does not depend on any random variables other than the variables in $\{R_j \mid i_1 < j \leq i_2\}$. Similarly, the events $i_2, \ldots, i_\ell \in S_t$ do not depend on any random variables other than the variables in $\{R_j \mid i_2 < j \leq t\}$. Since the random variables R_is are chosen independently at random and the sets $\{R_j \mid i_1 < j \leq i_2\}$ and $\{R_j \mid i_2 < j \leq t\}$ are disjoint, the events $i_1 \in S_{i_2}$ and $\bigcap_{k=2}^{\ell} A_{i_k}$ are independent; that is,

$$\Pr\left[i_1 \in S_{i_2}\middle|\bigcap_{k=2}^{\ell} A_{i_k}\right] = \Pr\left[i_1 \in S_{i_2}\right].$$

By Lemma 2, we have $\Pr\left[i_1 \in S_{i_2}\right] = \frac{1}{i_1} = \Pr\left[i_1 \in S_t\right] = \Pr\left[A_{i_1}\right]$ and thus,

$$\Pr\left[\bigcap_{k=1}^{\ell} A_{i_k}\right] = \Pr\left[A_{i_1}\right] \cdot \Pr\left[\bigcap_{k=2}^{\ell} A_{i_k}\right]. \tag{3.8}$$

Next, by using Equation 3.8 and applying induction on ℓ, we prove Equation 3.7. The base case, $\ell = 2$, follows immediately from Equation 3.8:

$$\Pr\left[\bigcap_{k=1}^{2} A_{i_k}\right] = \Pr\left[A_{i_1}\right] \cdot \Pr\left[A_{i_2}\right].$$

By induction hypothesis, for $\ell - 1$ events $A_{i_k}, 2 \le k \le \ell$, we have $\Pr\left[\bigcap_{k=2}^{\ell} A_{i_k}\right] = \prod_{k=2}^{\ell} \Pr\left[A_{i_k}\right]$. Then using Equation 3.8 for case $2 < \ell < t$, we have

$$\Pr\left[\bigcap_{k=1}^{\ell} A_{i_k}\right] = \Pr\left[A_{i_1}\right] \cdot \prod_{k=2}^{\ell} \Pr\left[A_{i_k}\right] = \prod_{k=1}^{\ell} \Pr\left[A_{i_k}\right].$$

□

Theorem 4. *Let L_t be the length of the dependency chain starting at vertex t and $L_{\max} = \max_t L_t$. Then the expected length $E[L_t] \le \log n$ and $L_{\max} = O(\log n)$ w.h.p., where n is the number of vertices.*

Proof. Let S_t and D_t be the selection chain and dependency chain starting at vertex t, respectively, and $X_t(i)$ be an indicator random variable such that $X_t(i) = 1$ if $i \in S_t$ and $X_t(i) = 0$ otherwise. Then we have

$$L_t = |D_t| \le |S_t| = \sum_{i=1}^{t-1} X_i(t).$$

Let $P_t(i)$ be the probability that $i \in S_t$; that is, $P_t(i) = \Pr[X_t(i) = 1]$ and $E[X_t(i)] = P_t(i) = \frac{1}{i}$. By linearity of expectation, we have

$$E[L_t] = \sum_{i=1}^{t-1} E[X_i] = \sum_{i=1}^{t-1} \frac{1}{i} = H_{t-1} \le \log t \le \log n$$

By Lemma 3, the random variables $X_t(i)$, for $1 \le i < t$, are mutually independent. Applying the Chernoff bound on independent Poisson trials, we have

$$\Pr\left[\sum_t X_t(i) \ge (1+\delta)\mu\right] \le \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

In the Chernoff bound, we set $\delta = \frac{6 \log n}{\mu} - 1$. Since $\mu \le \log n$, we have $\delta > 0$. Then,

$$\begin{aligned}
\Pr\left[L \ge 6 \log n\right] &= \Pr\left[L \ge (1+\delta)\mu\right] \\
&\le \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \\
&\le \left(\frac{e}{1+\delta}\right)^{\mu(1+\delta)}
\end{aligned}$$

$$\leq \left(\frac{e\mu}{6\log n}\right)^{6\log n}$$
$$\leq \left(\frac{e\log n}{6\log n}\right)^{\log n^6}$$
$$\leq \frac{1}{\left(\frac{6}{e}\right)^{\log n^6}}$$
$$\leq \frac{1}{n^{6\log\frac{6}{e}}} \qquad [a^{\log b} = b^{\log a}]$$
$$\leq \frac{1}{n^4}$$

Thus, with probability at least $1 - \frac{1}{n^4}$, the length of the dependency chain is $O(\log n)$. Using the union bound, it holds simultaneously for all n vertices with probability at least $1 - \frac{1}{n^3}$. Hence, we can say, the length of the dependency chain is $O(\log n)$ w.h.p. □

Dependency chain for the general case ($x \geq 1$). For the general case $x \geq 1$, each new vertex creates x new edges. Similar to the earlier case, each of these edges forms a selection and a dependency chain. Notice that all of the x selection chains originating from a new vertex are independent of each other because they independently execute the copy model (irrespective of other outgoing edges from the same vertex) and follow the exact same procedures with the same probabilities as shown in the Lemma 2 and Lemma 3. We already showed that the maximum length of a selection chain is at most $6\log n$ with probability $1 - \frac{1}{n^4}$ in Theorem 4. For the general case, there are $O(nx)$ such chains. Using the union bound, the probability that the maximum length is $6\log n$ for any of the $O(nx)$ selection chains is at least $O\left(1 - \frac{x}{n^3}\right)$. As $x \leq n$, we can say that the length of the dependency chain is still $O(\log n)$ w.h.p.

Experimental Validation. We also experimentally evaluated the maximum length of a dependency chain using our general algorithm (Algorithm 2). In this experiment, we varied the number of vertices n from $1K$ to $64M$. For each n, we also varied x from 1 to 128. For each possible combination of values of n and x, we calculated the maximum length of a dependency chain by repeating the algorithm several times. Figure 3.3 shows the maximum length of a dependency chain for each combination of n and x. We also plotted a fitted line of the function $y = a\log n + c$ using logarithmic regression. The fitted line has a correlation of 0.97. Therefore, the figure clearly suggests that the maximum length of a dependency chain varies logarithmically with n and is independent of x.

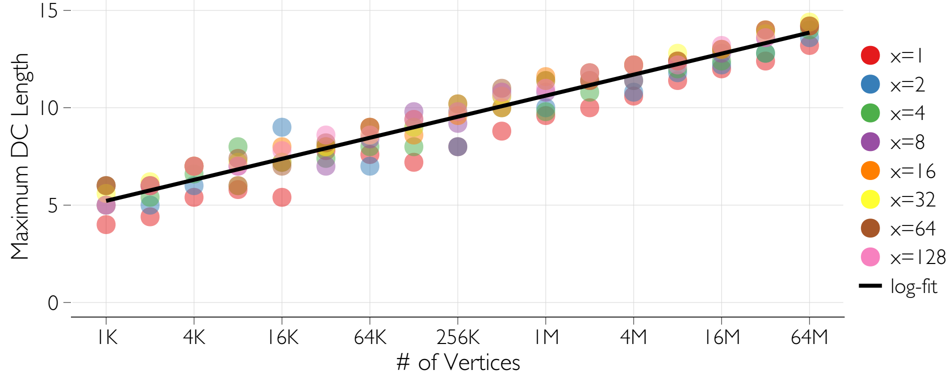


Figure 3.3: *Experimental result shows that the maximum length of a dependency chain is $O(\log n)$. The horizontal axis (in* log *scale) represents the number of vertices and the vertical axis represents the length of a dependency chain. Filled circles show the maximum length of a dependency chain for each pair of n and x. The solid line represents a logarithmic fit of the function $y = a \log n + c$.*

3.3.5 Waiting Queue

In our parallel algorithm, after receiving a request message for an edge $F_l(k)$ (Line 14 of Algorithm 2), a processor sends a corresponding response message immediately if the edge $F_l(k)$ is already known. Otherwise, the request message is stored in a queue called the *waiting queue* (Line 18 of Algorithm 2). If a processor receives a large number of such request messages whose responses could not be sent immediately, the size of the waiting queues becomes large, leading to a large memory requirement and the parallel algorithm has poor performance. Fortunately, the number of such request messages is not large. In this section, we provide a rigorous analysis showing that the maximum number of items for the waiting queue of a vertex is $O\left(x \log n\right)$ with high probability as shown in Theorem 5.

Theorem 5. *The maximum number of items stored in the waiting queue of a vertex is $O\left(x \log n\right)$ with high probability.*

Proof. Assume that the l-th outgoing edge of a vertex t executes the copy model and creates an edge with the endpoint of the ℓ-th edge of a vertex k, i.e., $F_l(t) = F_\ell(k)$ (Copy Edge). A request message $\langle F_l(t), F_\ell(k)\rangle$ is sent to processor $\mathcal{P}_j$ where $k \in V_j$. If $F_\ell(k)$ is not known at the time of receiving the message, the request will be put on a queue Q_k for vertex k in processor $\mathcal{P}_j$. The queue Q_k is called the *waiting queue* for vertex k. Once $F_\ell(k)$ is known, all the messages in Q_k for that edge will be processed and a corresponding response message will be sent (Line 23 of Algorithm 2).

Therefore, while creating a copy edge $(t, F_\ell(k))$, the event that the request message will be

put in the waiting queue Q_k consists of three events: 1) t selects $F_\ell(k)$, 2) t chooses to make the copy edge with probability $1-p$, and 3) $F_\ell(k)$ is not known. According to the step 1 of the copy model, t picks $F_\ell(k)$ with probability $\frac{1}{t-1}\frac{1}{x}$. Furthermore, $F_\ell(k)$ is already known with probability at least p (Direct Edge). Therefore, $F_\ell(k)$ is not known with probability at most $1-p$. Let $P_{k_\ell}(t)$ denotes the probability that any outgoing edge from vertex t makes a copy edge $(t, F_\ell(t))$ and the corresponding request message is put in the waiting queue Q_k. Therefore, we have:

$$\begin{aligned}
P_{k_\ell}(t) &= \Pr\left[F_\ell(k) \text{ is selected}\right] \times \Pr\left[\text{copy edge is created}\right] \times \Pr\left[F_\ell(k) \text{ is not known}\right] \\
&\le \frac{1}{t-1}\frac{1}{x}(1-p)(1-p) \\
&\le (1-p)^2 \frac{1}{x(t-1)}. \qquad (3.9)
\end{aligned}$$

Let $X_{k_\ell}(t)$ be a random variable that denotes the number of request messages stored in Q_k for the edge $F_\ell(k)$. Vertex t creates x edges independently and each of these edges stores a request messages in Q_k with probability $P_{k_\ell}(t)$. Therefore, we have:

$$E\left[X_{k_\ell}(t)\right] = xP_{k_\ell}(t) \le (1-p)^2 \frac{1}{(t-1)}.$$

Let $Y_k(t)$ be another random variable that denotes the total number of messages stored in Q_k from vertex t. Therefore, we have:

$$Y_k(t) = \sum_{\ell=1}^{x} X_{k_\ell}(t).$$

According to the parallel algorithm, Q_k can store messages from vertex $k+1$ to $n-1$. Thus, the total number of messages stored in Q_k is given by:

$$|Q_k| = \sum_{t=k+1}^{n-1} Y_k(t).$$

Therefore, the expected number of request messages stored in the queue Q_k is given by:

$$\begin{aligned}
E\left[|Q_k|\right] = \sum_{t=k+1}^{n-1} E[Y_k(t)] &= \sum_{t=k+1}^{n-1} \sum_{\ell=1}^{x} E\left[X_{k_\ell}(t)\right] \\
&\le \sum_{t=k+1}^{n-1} \sum_{\ell=1}^{x} (1-p)^2 \frac{1}{(t-1)}
\end{aligned}$$

$$
\begin{aligned}
&\leq (1-p)^2 x \sum_{t=k+1}^{n-1} \frac{1}{t-1} \\
&\leq (1-p)^2 x \sum_{i=k}^{n-2} \frac{1}{i} \\
&\leq (1-p)^2 x \left(H_{n-2} - H_i\right) \\
&\leq (1-p)^2 x H_n \\
&\leq (1-p)^2 x \log n. \qquad (3.10)
\end{aligned}
$$

Note that the random variables $Y_k(t)$ are mutually independent of each other. Applying the Chernoff bound on independent random variables, we have:

$$
\Pr\left[\sum_t Y_k(t) \geq (1+\delta)\mu\right] \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu
$$

In the Chernoff bound, we set $\delta = \frac{5x \log n}{\mu} - 1$. Since $\mu \leq (1-p)^2 x \log n$, where $0 \leq p \leq 1$. Note that when $p = 1$ no copy edge will be created, therefore, no item will be place in the waiting queue and the maximum number of items in Q_k is 0. For $p < 1$, we have $\delta > 0$. Then,

$$
\begin{aligned}
\Pr\left[|Q_k| \geq 5x \log n\right] &= \Pr\left[\sum_t Y_k(t) \geq (1+\delta)\mu\right] \\
&\leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu \\
&\leq \left(\frac{e}{1+\delta}\right)^{\mu(1+\delta)} \\
&\leq \left(\frac{e\mu}{5x \log n}\right)^{5x \log n} \\
&\leq \left(\frac{e(1-p)^2 x \log n}{5x \log n}\right)^{\log n^{5x}} \\
&\leq \frac{1}{\left(\frac{5x}{e}\right)^{\log n^{5x}}} && (1-p) < 1 \\
&\leq \frac{1}{n^{5x \log \frac{5x}{e}}} && [a^{\log b} = b^{\log a}] \\
&\leq \frac{1}{n^3} && [x \geq 1]
\end{aligned}
$$

Thus, with probability at least $1 - \frac{1}{n^3}$, the number of items in the waiting queue is $O(x \log n)$. Using the union bound, it holds simultaneously for all the n waiting queues with probability at least $1 - \frac{1}{n^2}$. Hence, we can say, the maximum number of items in the waiting queue of any vertex is $O(x \log n)$ w.h.p. □

Experimental Validation. In this section, we experimentally evaluate how the maximum size of a waiting queue varies with n, p, and x as shown in Theorem 5.

In Figure 3.4, we plot the maximum size of a waiting queue by varying n for a set of different x. We set $p = \frac{1}{2}$ in these experiments. In the figure, the circles represent the maximum size of a waiting queue collected experimentally, and the solid lines present a fit function $y = a \log n + c$ for different values of x. The horizontal axis is plotted in log scale. The figure demonstrates that the maximum size of a waiting queue is proportional to $\log n$.

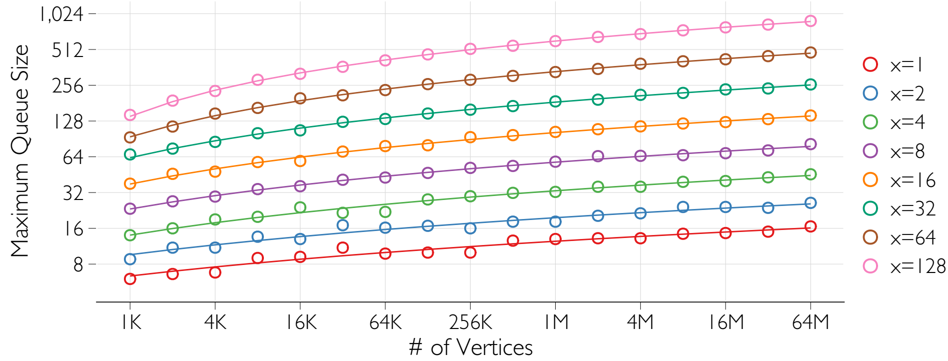


Figure 3.4: *The maximum size of a waiting queue varies logarithmically with n*

In Figure 3.5, we plot the maximum size of a waiting queue by varying x for a set of different n. We also set $p = \frac{1}{2}$ in these experiments. In the figure, the circles represent the maximum waiting queue size collected experimentally, and the solid lines present a linear fit function $y = ax + c$ for different values of n. The figure demonstrates that the maximum size of a waiting queue is proportional to x.

In Figure 3.6, we plot the maximum size of the waiting queues by varying p for a set of different n and x. In the figure, the circles represent the maximum waiting queue size collected experimentally, and the solid lines present a quadratic fit function $y = a(1-p)^2 + c$ for different values of n and x. The figure demonstrates that the maximum size of a waiting queue is proportional to $(1 - p)^2$.

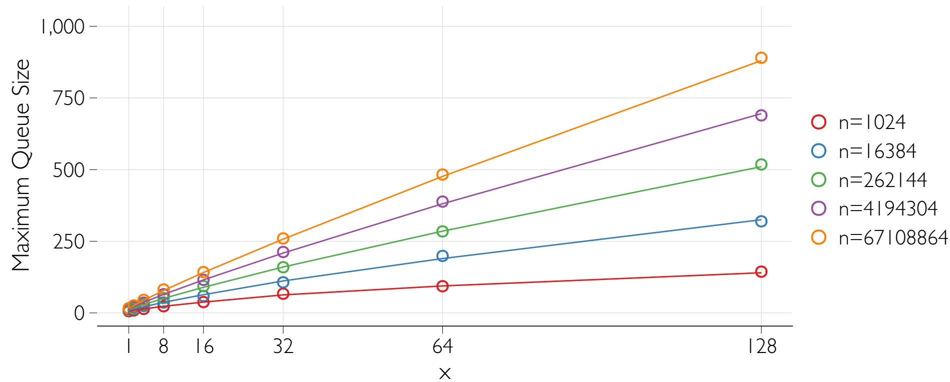


Figure 3.5: *The maximum size of a waiting queue varies linearly with x*

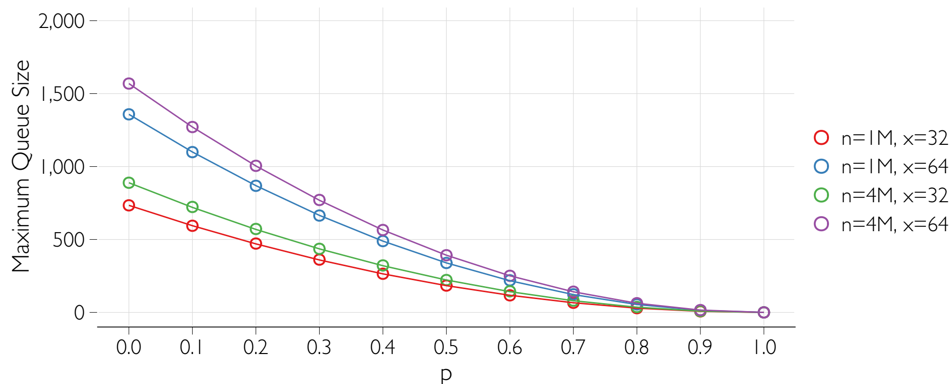


Figure 3.6: *The maximum size of the waiting queues changes linearly with p*

3.4 Partitioning and Load Balancing

Recall the formal definition of partitioning of the set of vertices $V = \{0, 1, \ldots, n-1\}$ into P subsets $V_0, V_1, \ldots, V_{P-1}$ as described at the beginning of Section 3.3.2. A good load balancing is achieved by properly partitioning the set of vertices V and assigning each subset to one processor. Vertex partitioning has significant effects on the performance of the algorithm. In this section, we study several partitioning schemes and their effects on load balancing and the performance of the algorithm. In our algorithm, we measure the computational load in terms of the number of vertices per processor, the number of outgoing messages (request messages) from a processor, and the number of incoming messages (response messages) to a processor.

There are several efficiency issues related to the partitioning of the vertices as described below. It is desirable that a partitioning of the vertices satisfies the following criteria.

A. For any given $k \in V$, finding the processor $\mathcal{P}_j$, where $k \in V_j$ (Line 8, Algorithm 1), can be done efficiently, preferably in constant time without communicating with the other processors.

B. The partitioning should lead to a good load balancing. The degrees of the vertices vary significantly, and a vertex with a larger degree causes more messages to work with. As a result, naïve partitioning may lead to poor load balancing.

C. As we discuss later, combining multiple messages (to the same destination) and using an `MPI_send` operation for them can increase the efficiency of the algorithm. However, combining multiple messages may not be possible with an arbitrary partitioning as it may cause deadlocks.

With the objective of satisfying the above criteria, we study the following partition schemes.

3.4.1 Consecutive Partitioning

In this partitioning scheme, the vertices are assigned to the processors sequentially. Partition V_i starts at vertex n_i and ends at $n_{i+1} - 1$, where $n_0 = 0$ and $n_{\mathbb{P}} = n$. That is, $V_i = \{n_i, n_i + 1, \ldots, n_{i+1} - 1\}$ for all i. With consecutive vertex partitioning, the only decision to be made is the number of vertices to be assigned to each set V_i. The simplest way to do so is to assign an equal number of vertices to each set, i.e., $|V_i| = \lceil \frac{n}{\mathbb{P}} \rceil$ for all i. We call this partitioning scheme *Simple Consecutive Partitioning* (SCP).

Simple Consecutive Partitioning

As discussed earlier, the sizes of the partitions are almost equal. Let $B = \lceil \frac{n}{\mathbb{P}} \rceil$. Then, the size of a partition is either B or $B - 1$. Partition V_i includes the vertices from iB to $(i + 1)B - 1$. Finding the rank of the processor from a vertex u is pretty straightforward in the SCP scheme. For a vertex $u \in V_i$, the rank of the processor $\mathcal{P}_i$ is given by $i = \lfloor \frac{u}{B} \rfloor$.

Optimal Consecutive Partitioning

The simple consecutive partitioning scheme satisfies Criterion A and C above; however, it is clear that such partitioning can lead to poor load balancing. The computation in each processor $\mathcal{P}_i$ involves the following three types of load:

A. generating random numbers and some other processing for each vertex $t \in V_i$,

B. sending request messages for the vertices in V_i and receiving their replies, and

C. receiving request messages from other processors and sending their replies.

The computational load for load type A and B above is directly proportional to the number of vertices in partition V_i. Computational load for load type C depends not only on the number of vertices in a processors but also on i, the rank of the processor. With simple consecutve vertex partitioning (SCP), a lower ranked processor receives more request messages than a higher ranked processor, because with $j < k$, $E[M_j] > E[M_k]$, where M_k is the number of request messages received for Vertex k (see Lemma 6).

Lemma 6. *Let M_k be the number of request messages received for vertex k. Then $E[M_k] = (1-p)(H_{n-1} - H_k)$, where H_k is the kth harmonic number.*

Proof. Vertex k receives a request message from vertex $t > k$ if and only if t randomly picks k and decided to assign F_k to F_t. The probability of such an event is $(1-p)\frac{1}{t}$. Then the expected number of messages received for Vertex k is given by

$$\sum_{t=k+1}^{n-1} (1-p)\frac{1}{t} = (1-p)(H_{n-1} - H_k)$$

□

Next we calculate the computational load for each processor with an arbitrary number of vertices assigned to the processors. To do so, we make the following simplifying assumptions: i) Sending a message takes the same computation time as receiving a message, and ii) $p = \frac{1}{2}$ (the same analysis will follow for arbitrary p by simply multiplying each term with $2(1-p)$). The number of vertices in Processor $\mathcal{P}_i$ is $n_{i+1} - n_i$. Then the computation cost for a load of type A and B is $c(n_{i+1} - n_i)$ for some constant c. Following Lemma 6, the expected load for type C in Processor $\mathcal{P}_i$ is

$$\begin{aligned}\sum_{k=n_i}^{n_{i+1}-1} (H_{n-1} - H_k) &= (n_{i+1} - n_i)H_{n-1} - \sum_{k=n_i}^{n_{i+1}-1} (H_k) \\ &= (n_{i+1} - n_i)H_{n-1} - (n_{i+1}H_{_{i+1}} - n_iH_{n_i}) + (n_{i+1} - n_i) \\ &= (n_{i+1} - n_i)(H_{n-1} + 1) - (n_{i+1}H_{n_{i+1}} - n_iH_{n_i}) \end{aligned} \tag{3.11}$$

The second to the last line follows from Equation 2.36 in page 41 of [64]. Thus, using another constant $b = 1 + c$, the total computational load at Processor $\mathcal{P}_i$ is

$$c(P_i) = (n_{i+1} - n_i)(H_{n-1} + b) - (n_{i+1}H_{n_{i+1}} - n_iH_{n_i})$$

The combined load for all processors is $c'n$ for some constant c' and desired load in each processor is $\frac{c'n}{P}$. Thus n_i, for all i, can be determined by solving the following system of equations, which is unfortunately nonlinear.

$$n_0 = 0$$
$$n_P = n - 1$$
$$c(P_i) = (n_{i+1} - n_i)(H_{n-1} + b) - (n_{i+1}H_{n_{i+1}} - n_iH_{n_i}) = \frac{c'n}{P} \quad (3.12)$$

Linear Consecutive Partitioning

A good load balancing can be achieved by solving the above system of equations. However two major difficulties arise:

- It seems the only way the above equations can be solved is by numerical methods that can take a prohibitively long time to compute.
- Criterion A for load balancing may not be satisfied, leading to poor performance.

To overcome these difficulties, guided by experimental results, we approximate the solution of the above system of equations with a linear function and call the resultant partitioning scheme *linear consecutive partitioning* (LCP). Figure 3.7 shows the distribution of the vertices among processors for actual solutions of Equation 3.12 and for the linear approximation. As we will see later in Section 3.5, our approximation scheme LCP provides very good load balancing and performance for the algorithm.

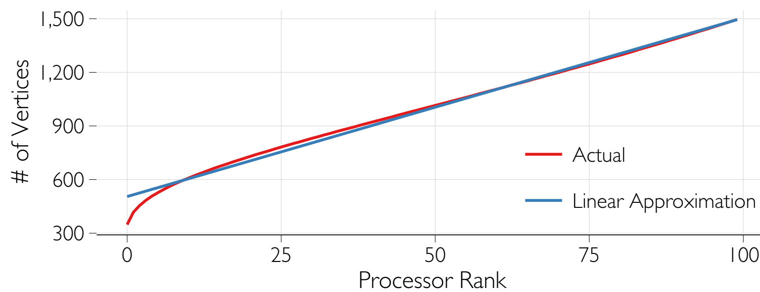


Figure 3.7: *Distribution of the vertices among processors for actual solutions of Equation 3.12 and its linear approximation.*

As in the LCP scheme, the number of vertices is increasing linearly with i (the ranks of the processors), the number of vertices in Processor $\mathcal{P}_i$ follows the arithmetic progression a, $a + d$, $a + 2d$, $\ldots$, $a + (P - 1)d$, that is, the number of vertices in Processor $\mathcal{P}_i$ is $B_i = a + id$,

where d is the slope of the line for linear approximation as shown in Figure 3.7. Slope d can be approximated easily by sampling two points on the actual line. Partition V_i has the vertices from $\sum_{j=0}^{i-1}(a + jd) = i\frac{(2a+(i-1)d)}{2}$ to $\sum_{j=0}^{i}(a + jd) - 1 = (i + 1)\frac{(2a+id)}{2} - 1$. Finding the rank of the processor for vetex u is more complicated in this scheme. Given a vertex u, we need to find the processor $\mathcal{P}_i$ such that $u \in V_i$. Vertex u satisfies the following inequality:

$$\sum_{j=0}^{i-1}(a + jd) \leq u < \sum_{j=0}^{i}(a + jd)$$
$$\frac{i\,(2a + (i-1)d)}{2} \leq u < \frac{(i+1)\,(2a + id)}{2} \tag{3.13}$$

Solving the inequality 3.13, we have

$$i = \left\lfloor \frac{-(2a-d) + \sqrt{(2a-d)^2 + 8du}}{2d} \right\rfloor \tag{3.14}$$

Determining partition parameters a and d. The parameters a and d are determined using the number of vertices n and the number of processors P. Parameter d is the slope of the straight line $y = a + dx$, where y represent the number of vertices in the processor with rank $x = i$. We calculate d by finding two points on this straight line. Putting $i = 0$ and $i = P - 1$ in Equation 3.12, we can compute n_1 and n_{P-1}. Then, the number of vertices in the first processor is $n_1 - n_0 = n_1$ and the number of vertices in the last processor is $n_P - n_{P-1} = n - 1 - n_{P-1}$. Hence, we have

$$d = \frac{n - 1 - n_{P-1} - n_1}{P}.$$

Now, we have

$$\sum_{j=0}^{P-1}(a + jd) = n$$
$$\frac{P\,(2a + (P-1)d)}{2} = n$$
$$a = \frac{n}{P} - \frac{(P-1)d}{2} \tag{3.15}$$

Message Buffering. The processors exchange two types of messages: request messages and resolve messages. For each vertex t, a processor may need to send one request message and receive one resolve message. If Processor $\mathcal{P}_i$ has multiple messages destined to the

same processor, say Processor $\mathcal{P}_j$, Processor $\mathcal{P}_i$ can combine them into a single message by buffering them instead of sending them individually. Each processor can do so by maintaining $P-1$ buffers, one for each of the other processors. If the messages are not combined, for large n, there can be a large number of outstanding messages in the system, and the system may not be able to deal with such a large number of messages at a time, limiting our ability to generate a large network. Further message buffering reduces the overhead of packet header and thus improves efficiency.

3.4.2 Round-Robin Partitioning

In Round-Robin Partitioning, vertices are distributed in a round robin fashion among all processors. Partition V_i contains the vertices $\langle i, i+p, i+2p, \ldots, i+kp\rangle$ such that $i+kp \le n < i+(k+1)p$; that is, $V_i = \{j | j \mod P = i\}$. In other words, vertex i is assigned to set $V_{i \mod p}$. Similar to SCP, in this RRP scheme also, the number of vertices in the sets is almost equal. The number of vertices in a set is either $\left\lceil \frac{n}{p} \right\rceil$ or $\left\lfloor \frac{n}{p} \right\rfloor$. The difference between the number of vertices in two sets is at most 1.

From Lemma 6, it is clear that the expected number of received messages decreases monotonically with increasing vertex labels. Round robin partitioning on such a monotonic distribution typically performs better. For the round robin vertex partitioning scheme, the computational load among processors are well-balanced as shown in Lemma 7.

Lemma 7. *The difference between the computational load for any two processors is at most $O(\log n)$, while the total computational load is $\Omega(n)$.*

Proof. The expected number of request messages received for vertex k is $(H_{n-1} - H_k)$ (see Lemma 6). Other loads for any vertex is constant. Then the total load for vertex k is $CL(k) = (H_{n-1} - H_k) + b$, for some constant b. Thus, the total load for Processor $\mathcal{P}_i$ with partition $V_i = \{j | j \mod P = i\}$ is $PL(i) = \sum_{k \in V_i} (H_{n-1} - H_k + b)$.

Notice that for any $k_1 < k_2$, $CL(k_1) > CL(k_2)$. As a result, we have $PL(i_1) > PL(i_2)$ for any $i_1 < i_2$. Thus the largest difference between the loads of two processors is

$$PL(0) - PL(P-1) = \sum_{k \in V_0} (H_{n-1} - H_k + b) - \sum_{k \in V_{P-1}} (H_{n-1} - H_k + b) \tag{3.16}$$

$$\le (H_{n-1} + b)(|V_0| - |V_{P-1}|) - \sum_{k \in V_0} H_k + \sum_{k \in V_{P-1}} H_k \tag{3.17}$$

If n is a multiple of P, we have

$$|V_0| - |V_{P-1}| = 0, \tag{3.18}$$

$$\sum_{k \in V_{P-1}} H_k < \sum_{k \in V_0} H_k + H_n, \tag{3.19}$$

$$\text{and thus, } PL(0) - PL(P-1) < H_n = \mathcal{O}(\log n). \tag{3.20}$$

Otherwise,

$$|V_0| - |V_{P-1}| = 1, \tag{3.21}$$

$$\sum_{k \in V_{P-1}} H_k \leq \sum_{k \in V_0} H_k, \tag{3.22}$$

$$\text{and thus, } PL(0) - PL(P-1) \leq H_{n-1} + b = \mathcal{O}(\log n). \tag{3.23}$$

□

The RRP Scheme also satisfies Criterion A: given a vertex, finding the processor where the vertex belongs to can be computed in constant time. Finding the rank of processor $\mathcal{P}_i$ for a given vertex $u \in V_i$ is determined by $i = u \mod P$.

Message buffering. For consecutive vertex partitioning (both naïve and LCP), message buffering (combining messages) does not require any special care to avoid deadlock. In SCP and LCP, since Processor $\mathcal{P}_i$ may wait only for Processor $\mathcal{P}_k$ such that $k < i$, there cannot be a circular waiting among the processors, and therefore deadlock cannot arise.

However, in the RRP scheme, deadlock can occur if the messages are not buffered carefully. The request messages can be buffered as it is done in SCP or LCP. The resolved message can also be buffered, but it needs to be done in a special way to avoid deadlock. To avoid deadlock, resolved messages must be sent out from the buffer (even if the buffer is not full yet) after processing every group of received messages (when buffering is used, messages are sent and received in groups). Sending the resolved messages cannot wait any longer. Otherwise, it can cause circular waiting among the processors leading to a deadlock situation.

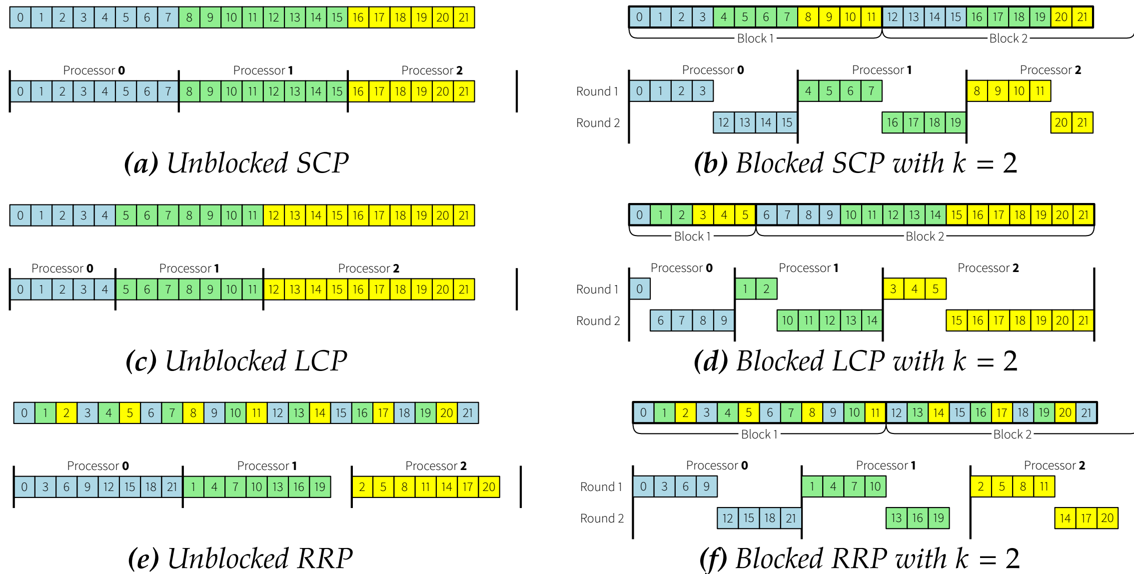
3.4.3 Block Partitioning

So far we have studied partitioning schemes where the entire set of vertices is partitioned into P subsets and each processor works on one subset. In this section, we present another fine-grained partitioning technique called *Block Partitioning*.

In the block partitioning technique, first the entire set of vertices are partitioned into k consecutive subsets called *blocks* (similar to the consecutive partitioning). The parallel algorithm for the copy model is executed in rounds. In each round, every block is further partitioned into P subsets (using the previous schemes) and processed using the P processors. After a round is completed, every edge originating from the vertices in the block is completely determined. We used blocked partitioning technique for SCP, LCP, and RRP schemes. The technique is illustrated in Figure 3.8.

As we will see in the experimental section, the block partitioning has several benefits. First, the technique offers fine grained tuning of load balancing. It also reduces the size of the waiting queue dramatically, as before going into the next round, all the edges are already processed. Therefore, the average size of the waiting queue reduces to $O((1-p)^2 x \log \frac{n}{k})$, where k is the number of blocks and the total number of items in the waiting queue is reduced with increasing block size. Additionally, the memory consumption is also reduced.

However, as block size increases beyond some point, we start losing the advantages because of synchronization issues needed to perform in each round. Therefore, there is an optimal value of k. We experimentally varied k to determine the optimal value for each partitioning scheme.



***(a)** Unblocked SCP*

***(b)** Blocked SCP with $k = 2$*

***(c)** Unblocked LCP*

***(d)** Blocked LCP with $k = 2$*

***(e)** Unblocked RRP*

***(f)** Blocked RRP with $k = 2$*

***Figure 3.8:** Blocked partitioning with $P = 3$ processors*

3.5 Experimental Results

In this section, we evaluate the performance of our algorithms experimentally. The accuracy of our parallel algorithm is demonstrated by showing that the algorithm produces a network with a power law degree distribution. Then we present the strong and weak scaling of the algorithms. These algorithms scale very well with the number of processors. We also present experimental results showing the impact of the partitioning schemes on load balancing and performance of the algorithms.

Experimental Setup. We used a high-performance computing cluster of 64 Intel Sandy Bridge nodes. Each node consists of two dual-socket Intel Sandy Bridge E5-2670 2.60GHz 8-core processors (16 cores per node) and 64GB of 1600MHz DDR3 RAM. The nodes are interconnected by QLogic QDR InfiniBand interconnects. For the MPI-based implementation of our algorithms, we used MPICH2 (version 1.7), which is optimized for QLogic InfiniBand cards.

In the experiments, we used up to 1024 processors. Each of the algorithms we considered generates the network in main memory, and the run time does not include the time required to write the graph to disk.

3.5.1 Degree Distribution

The degree distribution of the graph generated by our parallel algorithm is shown in Figure 3.9 in a $\log - \log$ scale. We used $n = 1$B vertices and $x = 4$ that generates a network with 4B edges. As shown in the figure, the copy model produces power-law degree distributions for various values of p. When $p = 0.5$, the degree distribution is the same as the BA model. As the figure shows, the distribution is heavy tailed, which is a distinct feature of the real-world power-law networks. The exponent γ of this power-law degree distribution is measured to be 2.7, which supports the fact that for a finite average degree of a scale-free network, the exponent γ satisfies $2 < \gamma < \infty$ [48]. When p is very close to 0, the network is mainly built on copy edges, therefore, there is a higher level of bias towards the higher degree vertices as evident from the longer tail. However, when p is close to 1, the network mainly consists of direct edges, and we do not see long tails, a salient property of many real world networks. The above results show that the copy model is more general than the Barabási-Albert model and capable of generating many interesting degree distributions. Further, it also shows that our algorithms produce scale-free networks very accurately.

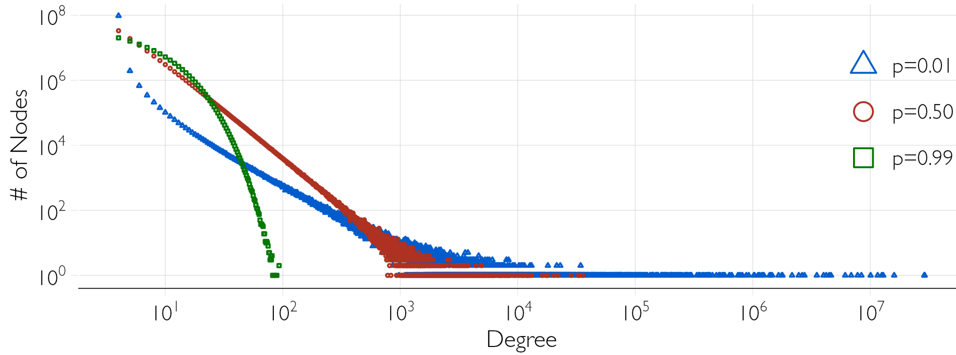


Figure 3.9: *The degree distribution (in* log − log *scale) of the network generated by our parallel algorithms. The network is generated with* $n = 10^9$ *and* $x = 4$*.*

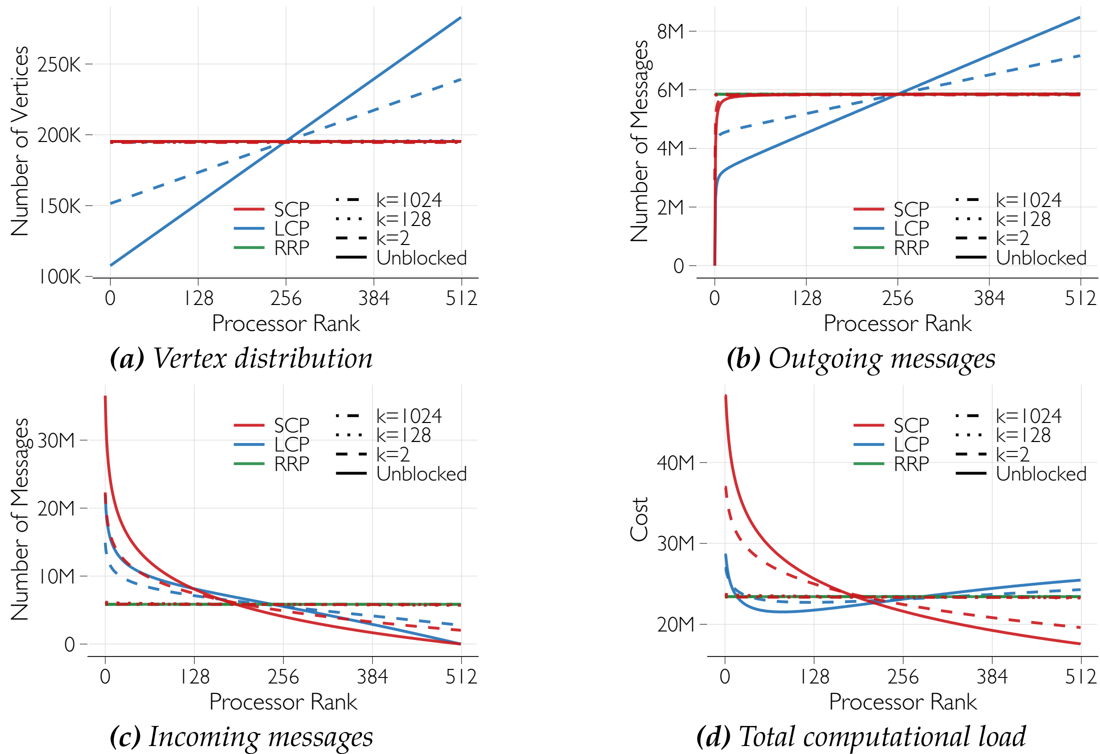
3.5.2 Partitioning and Load Balancing

Vertex partitioning has significant effects on load balancing and performance of the algorithm. In Section 3.4, we have discussed three partitioning schemes UCP, LCP, and RRP, and theoretically analyzed them. In this section, we experimentally study these schemes and their effect on the performance of the algorithm. In these experiments, we use $n = 100$M vertices, $x = 60$ edges per vertex, and 512 processors. Note that 512 processors are sufficient to demonstrate the behavior and differences of the partitioning schemes. For each of the three schemes, we measure the computational load in the processors by the number of vertices per processor, the number of outgoing messages from the processors, and the number of incoming messages to the processors. The results are shown in Figure 3.10.

Vertex Distribution. The vertex distribution is shown in Figure 3.10(a). For SCP and RRP, vertices are distributed uniformly among the processors, and each processor has about 195K vertices. For LCP, the number of vertices in the processors are increasing linearly with the rank of the processors.

Message Distribution. In a consecutive partitioning (SCP and LCP), processor $\mathcal{P}_i$ sends outgoing request messages to processors $\mathcal{P}_0$ to $\mathcal{P}_{i-1}$ and receives incoming messages from processors $\mathcal{P}_{i+1}$ to $\mathcal{P}_{P-1}$. For each vertex, a processor sends a request message with probability at most $1-p$ (see Equation 3.2). Thus, the expected number of request messages sent by a processor is proportional to the number of vertices in the processor, as shown in Figure 3.10(b). Note that in the SCP and LCP schemes, processor $\mathcal{P}_0$ does not need to send any request messages at all.

Figure 3.10(c) shows the number of incoming request messages for each processor. It is clear that a lower ranked processor receives more messages than a higher ranked processor



(a) *Vertex distribution* **(b)** *Outgoing messages*

(c) *Incoming messages* **(d)** *Total computational load*

Figure 3.10: *Vertex and message distribution for the partitioning schemes*

in consecutive partitioning (SCP and LCP) as suggested by Lemma 6. In the RRP scheme, both incoming and outgoing messages are evenly distributed among the processors.

Total Load Distribution. Besides sending and receiving messages, for each vertex, a processor can incur a constant amount of additional computational cost. Thus, for analysis purposes, we measure the total computational load of a processor as the sum of the number of vertices in the processor and the number of incoming and outgoing messages. Figure 3.10(d) shows the total load for the three partitioning schemes. The RRP scheme distributes the load almost perfectly among the processors. Load balancing in the LCP scheme is also quite good. On the other hand, the SCP scheme distributes the load very poorly. These experimental results verify our theoretical analysis given in Section 3.4.

Size of the Waiting Queue. With the blocked partitioning scheme, the total size of the waiting queues is reduced with increasing block size as shown in Figure 3.11. Therefore, blocked partitions yield better performance in our algorithm.

Effect of Block Size. Although an increasing block size reduces the size of a waiting queue, it also reduces concurrency. Therefore, if the block size is increased beyond some limit, the performance would start to decrease. This is demonstrated in Figure 3.12.

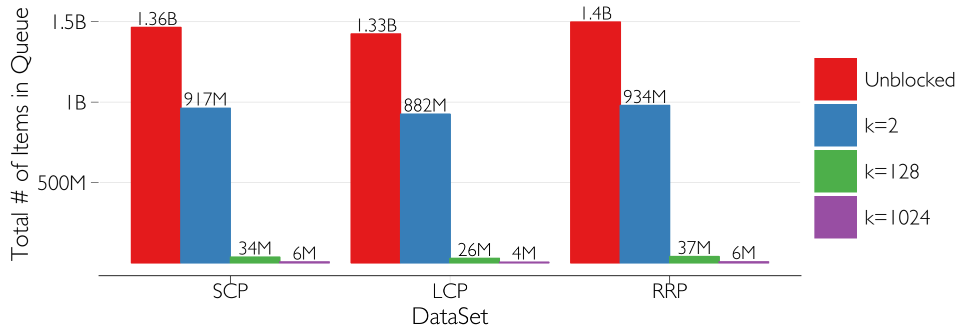


Figure 3.11: *Change of run time based on block size*

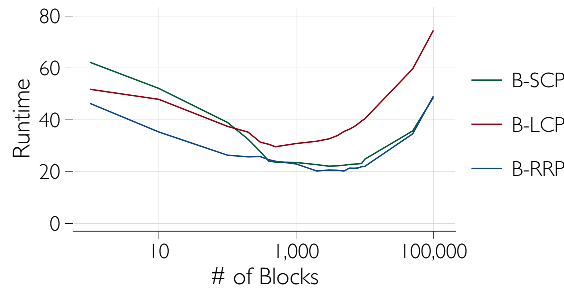
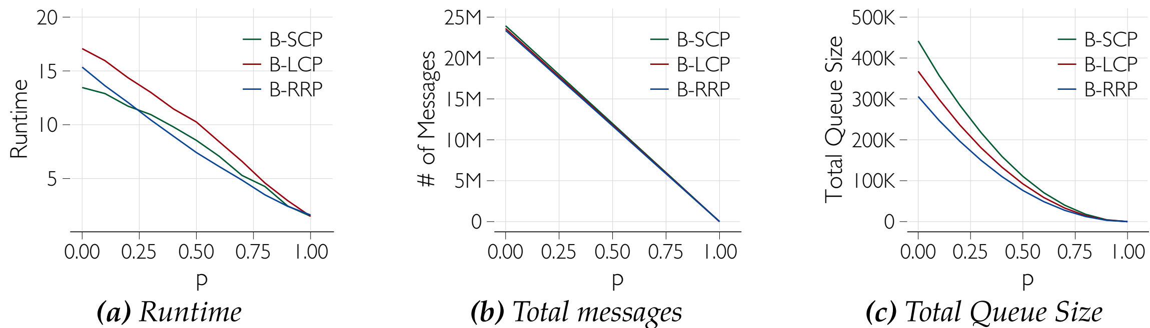


Figure 3.12: *Change of run time based on block size*

Effect of p on Performance. If p is reduced, most of the edges produced consist of copy edges, therefore requiring more message exchanges. As p is increased towards 1, most edges consist of direct edges. Therefore communication is reduced. This is shown in Figure 3.13.



(a) *Runtime* ***(b)*** *Total messages* ***(c)*** *Total Queue Size*

Figure 3.13: *Effect of blocked partitioning on block size*

3.5.3 Scalability

Strong Scaling. Strong scaling of a parallel algorithm shows its performance with an increasing number of processors keeping the problem size fixed. Figure 3.14 shows speedup factors of our algorithms with blocked and unblocked techniques using simple consecutive (SCP), linear consecutive (LCP), and round-robin partitioning (RRP) partitioning schemes, as the number of processors increases with problem size $n = 100\mathbf{M}$ and $x = 60$. Speedup factors are measured as T_s/T_p, where T_s and T_p are the running time of a sequential algorithm and the parallel algorithm, respectively. We have implemented the sequential version of our algorithm in C++. This sequential implementation outperforms the best available implementation of the BA model given in the NetworkX graph algorithm library [68]. As the sequential algorithm cannot generate more than 6**B** edges due to memory limitations, we choose $n = 100\mathbf{M}$ and $x = 60$. We varied the number of processors from 1 to 1024 for this experiment.

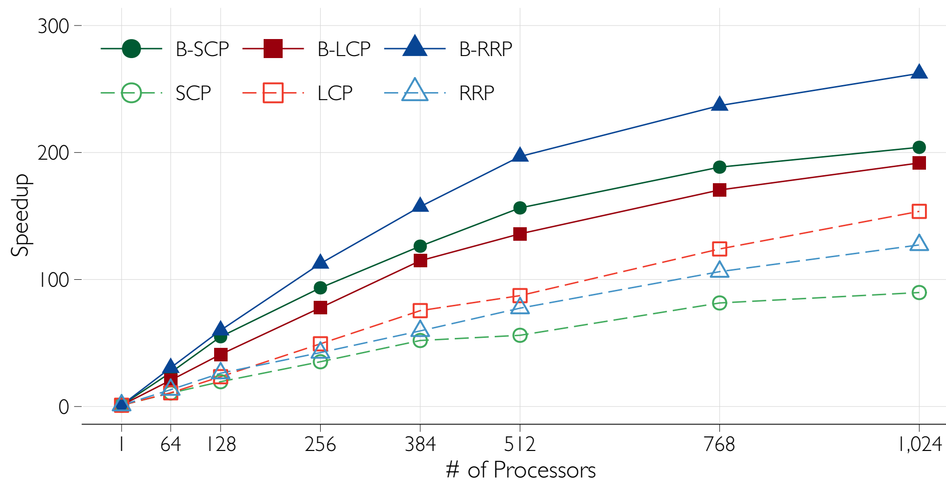


Figure 3.14: *The strong scaling of our parallel algorithms for the problem size $n = 100M$ and $x = 60$.*

Parallelization of network algorithms is notoriously hard. Furthermore, we have observed that the problem of generating a scale-free random network is quite sequential in nature due to the dependencies among the edges. As Figure 3.14 shows, the speedups of our algorithms are increasing almost linearly with the number of processors. Given the sequential nature of the problem, our algorithms show very good speedup. Further, the speedup of blocked versions performs better than the original versions. Note that both B-SCP and B-RRP are performing the best, due to better load balancing and reduced queue

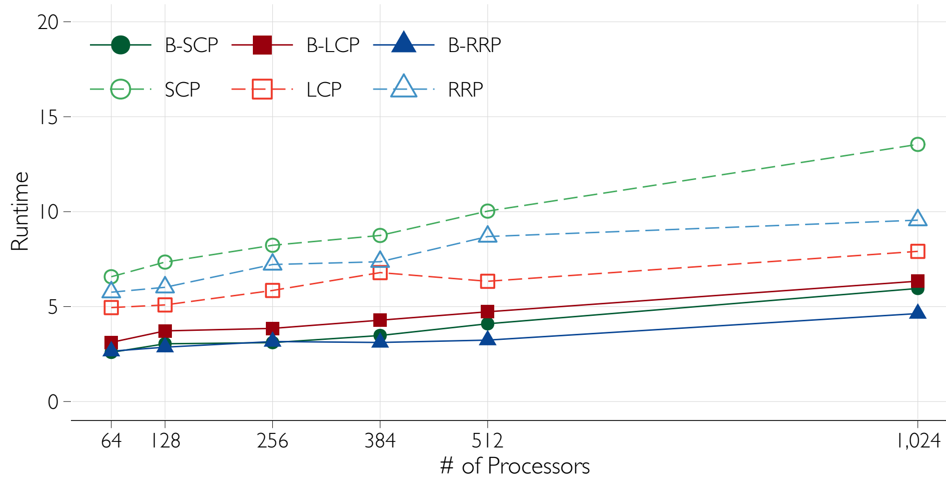


Figure 3.15: *Weak scaling of our parallel PA algorithm.*

size.

Weak Scaling. Weak scaling measures the performance of a parallel algorithm when the input size per processor remains constant. For this experiment, we varied the number of processors from 16 to 1024. With the number of processors, the input size is also increased proportionally: for P processors, a network with $10^7 P$ edges is generated. Figure 3.15 shows the weak scaling of our algorithms with the increasing number of processors.

In a perfect weak scaling case, the run time is expected to remain constant as the number of processors (P) increases. However, in practice, communication among processors increases with P, leading to an increase in run time. Our algorithm with the LCP and RRP schemes shows very good weak scaling, almost constant run time. Again, due to poor load balancing in the SCP scheme, we have worse weak scaling.

Generating Large Networks. Our main goal for designing this algorithm is to generate very large random networks. Using our algorithm with the RRP scheme, we are able to generate a network with 400 billion edges, with $n = 10\,\mathbf{B}$ and $x = 40$. Using 1024 processors, the generation of this network takes only five minutes.

3.6 Conclusion

We developed a parallel algorithm to generate massive scale-free networks using the preferential attachment model. We analyzed the dependency nature of the problem in detail, which led to the development of an efficient parallel algorithm for the problem.

Various node partitioning schemes and their effect on the algorithm were discussed as well. Our algorithm produces networks that strictly follow a power-law distribution. The linear scalability of our algorithm enables us to produce a network of 400 billion edges in just five minutes. It will be interesting to develop scalable parallel algorithms for other classes of random networks in the future.

Acknowledgment

We thank our external collaborators, members of the Network Dynamics and Simulation Science Laboratory (NDSSL), and the anonymous reviewers for their suggestions and comments. This work has been partially supported by DTRA CNIMS Contract HDTRA1-11-D-0016-0001, DTRA Grant HDTRA1-11-1-0016, DTRA NSF NetSE Grant CNS-1011769 and NSF SDCI Grant OCI-1032677.

Chapter 4

Parallel Algorithm for Generating Massive Random Networks with a Given Sequence of Expected Degrees

4.1 Introduction

Random networks are widely used for modeling and analyzing complex processes. Many mathematical models have been proposed to capture diverse real-world networks. One of the most important aspects of these models is degree distribution. Chung-Lu (CL) model [39] is a random network model, which can produce networks with any given arbitrary degree distribution. In this chapter, we present time- and space-efficient MPI-based distributed memory parallel algorithms for generating random networks from a given sequence of expected degrees using the CL model. Please note that this algorithm can easily be adapted for shared-memory parallel systems. To the best of our knowledge, it is the first parallel algorithm for the CL model. The most challenging part of this algorithm is load balancing. Partitioning the vertices with a balanced computational load is a non-trivial problem. In a sequential setting, many algorithms for the load-balancing problem were studied [102, 110, 116]. Some of them are exact and some are approximate. These algorithms use many different techniques such as heuristic, iterative refinement, dynamic programming, and parametric search. All of these algorithms require at least $\Omega(n + P \log n)$ time, where n, P are the number of vertices and processors respectively. To the best of our knowledge, there is no parallel algorithm for this problem. In this chapter, we present a novel and efficient parallel algorithm for computing the balanced partitions in $\mathcal{O}\left(\frac{n}{P} + P\right)$ time. The parallel algorithm for load balancing can be of independent interest

and probably could be used in many other problems. Using this load balancing algorithm, the parallel algorithm for the CL model takes an overall run time of $O\left(\frac{n+m}{P} + P\right)$ with high probability (w.h.p.). The algorithm requires $O(n)$ space per processor. We also present a space-efficient algorithm for the CL model that requires only $O\left(\frac{n}{P}\right)$ space per processor. Our algorithm scales very well to a large number of processors and can generate a power-law network with one billion vertices and 250 billion edges in memory in less than a minute using 1024 processors.

The rest of the chapter is organized as follows. In Section 4.2 we describe the problem and the efficient sequential algorithm. In Section 4.3, we present the time-efficient parallel algorithm along with an analysis of partitioning and load balancing. In Section 4.4, we present the space-efficient algorithm. Experimental results showing the performance of our parallel algorithms are presented in Section 4.5. We conclude in Section 4.6.

4.2 The Chung-Lu Model and Efficient Sequential Algorithm

The Chung-Lu (CL) model [39] generates random networks from a given sequence of expected degrees. We are given n vertices and a set of non-negative weights $w = (w_0, \ldots w_{n-1})$ assuming $\max_i w_i^2 < S$, where $S = \sum_k w_k$ [39]. For every pair of vertices i and j, edge (i, j) is added to the graph with probability $p_{i,j} = \frac{w_i w_j}{S}$. If no self loops are allowed, i.e., $i \neq j$, the expected degree of vertex i is given by $\sum_j \frac{w_i w_j}{S} = w_i - \frac{w_i^2}{S}$. For massive graphs, where n is very large, the average degree converges to w_i, thus w_i approximates the expected degree of vertex i [105].

The naïve algorithm for the CL model for an undirected graph with n vertices takes each of the $\frac{n(n-1)}{2}$ possible vertex pairs $\{i, j\}$ and creates the edge with probability $p_{i,j}$, therefore requiring $O\left(n^2\right)$ time. An $O(n + m)$ algorithm was proposed in [105] to generate networks, assuming w is sorted in non-increasing order, where m is the number of edges. It is easy to see that $O(n + m)$ is the best possible run time to generate m edges. The algorithm is based on the edge skipping technique introduced in [19] for the Erdős-Rényi model. Adaptation of that technique leads to the efficient sequential algorithm in [105]. The pseudocode of the algorithm is given in Algorithm 3, consisting of two procedures Serial-CL() and Create-Edges(). Note that we restructured Algorithm 3 by defining procedure Create-Edges() to use it without any changes later in our parallel algorithm. Below we provide an overview and a brief description of the algorithm (for complete explanation and correctness see [105]).

Algorithm 3: Sequential Chung-Lu Algorithm

```
1  Procedure SERIAL-CL(w)
2      S ← ∑_k w_k
3      E ← CREATE-EDGES(w, S, V)

4  Procedure CREATE-EDGES(w, S, V)
5      E ← ∅
6      forall i ∈ V do
7          j ← i + 1
8          p ← min(w_i w_j / S, 1)
9          while j < n and p > 0 do
10             if p ≠ 1 then
11                 choose a random r ∈ (0, 1)
12                 δ ← ⌊log(r) / log(1−p)⌋
13             else
14                 δ ← 0
15             v ← j + δ                                  // skip δ edges
16             if v < n then
17                 q ← min(w_i w_v / S, 1)
18                 choose a random r ∈ (0, 1)
19                 if r < q/p then
20                     E ← E ∪ {i, v}
21                 p ← q
22                 j ← v + 1
23     return E
```

The algorithm starts at SERIAL-CL(), which computes the sum S and calls procedure CREATE-EDGES(w, S, V), where V is the entire set of vertices. For each vertex $i \in V$, the algorithm selects some random vertices v from $[i+1, n-1]$, and creates the edges (i, v). A naïve way to select the vertices v from $[i+1, n-1]$ is: for each $j \in [i+1, n-1]$, select j independently with probability $p_{i,j} = \frac{w_i w_j}{S}$, leading to an algorithm with run time $O(n^2)$. Instead, the algorithm skips the vertices that are not selected by a random skip length δ as follows. For each $i \in V$ (Line 6), the algorithm starts with $j = i + 1$ and computes a random skip length $\delta \leftarrow \left\lfloor \frac{\log(r)}{\log(1-p)} \right\rfloor$, where r is a real number in $(0, 1)$ chosen uniformly at random and $p = p_{i,j} = \frac{w_i w_j}{S}$. Then vertex v is selected by skipping the next δ vertices

(Line 15), and edge (i, v) is selected with probability $\frac{q}{p}$, where $q = p_{i,v} = \frac{w_i w_v}{S}$ (Line 17-20). Then from the next vertex $j + v$, this cycle of skipping and selecting edges is repeated (while loop in Line 9-22). As we always have $i < j$ and no edge (i, j) can be selected more than once, this algorithm does not create any self-loops or parallel edges. As the set of weights w is sorted in non-increasing order, for any vertex i, the probability $p_{i,j} = \frac{w_i w_j}{S}$ decreases monotonically with the increase of j. It is shown in [105] that for any i, j, edge (i, j) is included in E with probability exactly $\frac{w_i w_j}{S}$, as desired, and that the algorithm runs in $O(n + m)$ time.

4.3 A Time Efficient Parallel Algorithm for the CL Model

Next we present our time efficient distributed memory parallel algorithm for the CL model. Although our algorithm generates undirected edges, for the ease of discussion we consider u as the *source vertex* and v as the *destination vertex* for any edge (u, v) generated by the procedure Create-Edges(). Let T_u be the task of generating the edges from source vertex u (Lines 6-22 in Algorithm 3). It is easy to see that, for any pair of vertices (u, v), generating edges in task T_u does not depend on generating edges in task T_v, i.e., tasks T_u and T_v can be executed independently by two different processors. Now execution of procedure Create-Edges(w, S, V) is equivalent to executing the set of tasks $\{T_u : u \in V\}$. Efficient parallelization of Algorithm 3 requires:

- Computing the sum $S = \sum_{k=0}^{n-1} w_k$ in parallel
- Dividing the task of executing Create-Edges() into independent subtasks
- Accurately estimating the computational cost for each task
- Balancing computational load among the processors

To compute the sum S efficiently, a parallel sum operation is performed on w using P processors, which takes $O\left(\frac{n}{P} + \log P\right)$ time. To divide the task of executing procedure Create-Edges() into independent subtasks, the set of vertices V is divided into P disjoint subsets $V_1, V_2, \ldots, V_P$, that is, $V_i \subset V$, such that for any $i \neq j$, $V_i \cap V_j = \emptyset$ and $\bigcup_i V_i = V$. Then V_i is assigned to processor $\mathcal{P}_i$, and $\mathcal{P}_i$ executes the tasks $\{T_u : u \in V_i\}$, that is, $\mathcal{P}_i$ executes Create-Edges(w, S, V_i).

Estimating and balancing computational loads accurately are the most challenging tasks. To achieve a good speedup of the parallel algorithm, both tasks must also be done in

parallel, which are non-trivial problems. A good load balancing is achieved by properly partitioning the set of vertices V such that the computational loads are equally distributed among the processors. We use two classes of partitioning schemes named consecutive partitioning (CP) and round-robin partitioning (RRP). In the CP scheme, consecutive vertices are assigned to each subset, whereas, in the RRP scheme, vertices are assigned to the partitions in a round-robin fashion. The use of various partitioning schemes is not only interesting for understanding the performance of the algorithm but also useful in analyzing the generated networks. It is sometimes desirable to generate networks on the fly and analyze them without performing disk I/O. Different partitioning schemes can be useful for different network analysis algorithms. Many network analysis algorithms require partitioning the network into an equal number of vertices (or edges) per processor. Some algorithms also require the consecutive vertices to be stored in the same processor. Before discussing the partitioning schemes in detail, we describe some formulations that are applicable to all of these schemes.

Let e_u be the expected number of edges produced and c_u be the computational cost in task T_u for a *source vertex* u. For the sake of simplicity, we assign one unit of time to process a vertex or an edge. With $S = \sum_{v=0}^{n-1} w_v$, we have:

$$e_u = \sum_{v=u+1}^{n-1} p_{u,v} = \sum_{v=u+1}^{n-1} \frac{w_u w_v}{S} = \frac{w_u}{S} \sum_{v=u+1}^{n-1} w_v \tag{4.1}$$

$$c_u = e_u + 1. \tag{4.2}$$

For two vertices $u, v \in V$ such that $u < v$, we have $c_u \geq c_v$ as shown in Lemma 8.

Lemma 8. *For any two vertices $u, v \in V$ such that $u < v$, $c_u \geq c_v$.*

Proof. The lemma follows immediately from Equation 4.2 and the fact that, the weights are sorted in non-increasing order. □

The expected number of edges generated by the tasks $\{T_u : u \in V_i\}$ is given by $m_i = \sum_{u \in V_i} e_u$. Note that the expected number of edges in the generated network, i.e., the expected total number of edges generated by all processors is $m = |E| = \sum_{i=0}^{P-1} m_i = \sum_{u=0}^{n-1} e_u$. The expected computational cost for processor $\mathcal{P}_i$ is given by

$$c(V_i) = \sum_{u \in V_i} c_u = \sum_{u \in V_i} (e_u + 1) = m_i + |V_i|. \tag{4.3}$$

Therefore, the total cost for all processors is given by

$$\sum_{i=0}^{P-1} c(V_i) = \sum_{i=0}^{P-1} (m_i + |V_i|) = m + n. \tag{4.4}$$

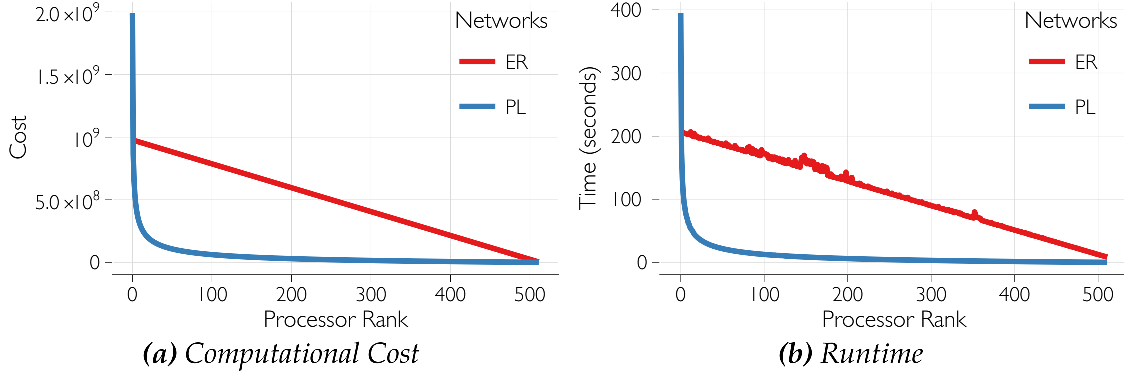
4.3.1 Consecutive Partitioning (CP)

Let subset V_i start at vertex n_i and end at vertex $n_{i+1} - 1$, where $n_0 = 0$ and $n_P = n$, i.e., $V_i = \{n_i, n_i + 1, \ldots, n_{i+1} - 1\}$, for all i. We say n_i is the *lower boundary* of subset V_i. A naïve way for partitioning V makes each V_i consist of an equal number of vertices, i.e., $|V_i| = \lceil \frac{n}{P} \rceil$ for all i. To keep the discussion neat, we simply use $\frac{n}{P}$. Although the number of vertices in each subset is the same, the computational cost among the processors is unbalanced. Lemma 9 shows that for two consecutive subsets V_i and V_{i+1}, $c(V_i) > c(V_{i+1})$ for all i, and the difference is at least $\frac{n^2}{SP^2}\overline{W}_i\overline{W}_{i+1}$, where $\overline{W}_i = \frac{1}{|V_i|}\sum_{u \in V_i} w_u$, the average weight (degree) of the vertices in V_i.

Lemma 9. *Let $c(V_i)$ be the computational cost for subset V_i. In the naïve partitioning scheme, we have $c(V_i) - c(V_{i+1}) \geq \frac{n^2}{SP^2}\overline{W}_i\overline{W}_{i+1}$, where $\overline{W}_i = \frac{1}{|V_i|}\sum_{u \in V_i} w_u$, the average weight of the vertices in V_i.*

Proof. In the naïve partitioning scheme, each of the subsets has $x = \frac{n}{P}$ vertices, except the last subset, which may have fewer than x vertices. For the ease of discussion, assume that for $u \geq n$, $w_u = 0$ and consequently $e_u = 0$. Now, $V_i = \{ix, ix + 1, \ldots, (i + 1)x - 1\}$. Using Equation 4.3, we have

$$\begin{aligned}
c(V_i) - c(V_{i+1}) &= \sum_{u \in V_i} (e_u + 1) - \sum_{u \in V_{i+1}} (e_u + 1) \\
&\geq \sum_{u=ix}^{(i+1)x-1} (e_u + 1) - \sum_{u=(i+1)x}^{(i+2)x-1} (e_u + 1) \\
&= \sum_{u=ix}^{(i+1)x-1} (e_u - e_{u+x}) \\
&= \sum_{u=ix}^{(i+1)x-1} \left(\frac{w_u}{S} \sum_{v=u+1}^{n-1} w_v - \frac{w_{u+x}}{S} \sum_{v=u+x+1}^{n-1} w_v \right) \\
&\geq \sum_{u=ix}^{(i+1)x-1} \frac{w_u}{S} \sum_{v=u+1}^{u+x} w_v \\
&\geq \sum_{u=ix}^{(i+1)x-1} \frac{w_u}{S} x\overline{W}_{i+1} \\
&= \frac{x\overline{W}_{i+1}}{S} \cdot x\overline{W}_i
\end{aligned}$$



***(a)** Computational Cost* ***(b)** Runtime*

***Figure 4.1:** Computational cost and run time in naïve CP scheme*

$$= \frac{n^2}{SP^2}\overline{W}_i\overline{W}_{i+1}.$$

□

Thus $c(V_i)$ gradually decreases with i by a large amount leading to a very unbalanced distribution of the computational cost.

To demonstrate that the naïve CP scheme leads to an unbalanced distribution of computational cost, we generated two networks, both with one billion vertices: i) an Erdős-Rényi network with an average degree of 500, and ii) a power-law network with an average degree of 49.72. We used 512 processors, which is good enough for this experiment.

Figure 4.1 shows the computational cost and run time per processor. In both cases, the cost is not well-balanced. For the power-law network, the imbalance of computational cost is more prominent. Observe that the run time is almost directly proportional to the cost, which justifies our choice of cost function, that is, balancing the cost will also balance the run time.

We need to find the sets V_i such that each set has equal cost, i.e., $c(V_i) \approx \overline{Z}$, where $\overline{Z} = \frac{(m+n)}{P}$ is the average cost per processor. We refer to such a partitioning scheme as uniform cost partitioning (UCP). Although determining the partition boundaries in the naïve scheme is very easy, finding the boundaries in the UCP scheme is a non-trivial problem and requires: (i) computing the cost c_u for each vertex $u \in V$ and (ii) finding the boundaries of the partitions such that every subset has a cost of $\overline{Z}$. Naïvely computing costs for all vertices takes $O\left(n^2\right)$ time as each vertex independently requires $O\left(n\right)$ time using Equation 4.1 and 4.2. A trivial parallelization achieves $O\left(\frac{n^2}{P}\right)$ time. Our algorithm performs this computation in parallel in $O\left(\frac{n}{P} + \log P\right)$ time.

Finding the partition boundaries such that the maximum cost of a subset is minimized is a

well-known problem named the *chains-on-chains partitioning* (CCP) problem [116]. In CCP, a sequence of $P-1$ separators is determined to divide a chain of n tasks with associated non-negative weights (c_u) into P sets so that the maximum cost in the sets is minimized. Sequential algorithms for CCP are studied quite extensively [102, 110, 116]. Since these algorithms take at least $\Omega(n + P\log n)$ time, using any of these sequential algorithms to find the partition boundaries, along with the parallel algorithm for the CL model, does not scale well. To the best of our knowledge, there is no known parallel algorithm for the CCP problem. We present a novel parallel algorithm for determining the partition boundaries that takes $O\left(\frac{n}{P} + P\right)$ time in the worst case.

To determine the partition boundaries, instead of using c_u directly, we use the cumulative cost $C_u = \sum_{v=0}^{u} c_v$. We call a set V_i a *balanced subset* if the computational cost of V_i is $c(V_i) = \sum_{u=n_i}^{n_{i+1}-1} c_u = C_{n_{i+1}-1} - C_{n_i-1} \approx \overline{Z}$. Also note that for lower boundary n_i of set V_i we have, $C_{n_i-1} < i\overline{Z} \leq C_{n_i}$ for $0 < i \leq P-1$. Thus, we have

$$n_i = \arg\min_u \left(C_u \geq i\overline{Z}\right). \tag{4.5}$$

In other words, a vertex u with cumulative cost C_u belongs to subset V_i such that $i = \left\lfloor \frac{C_u}{\overline{Z}} \right\rfloor$. The partitioning scheme is shown visually in Figure 4.2.

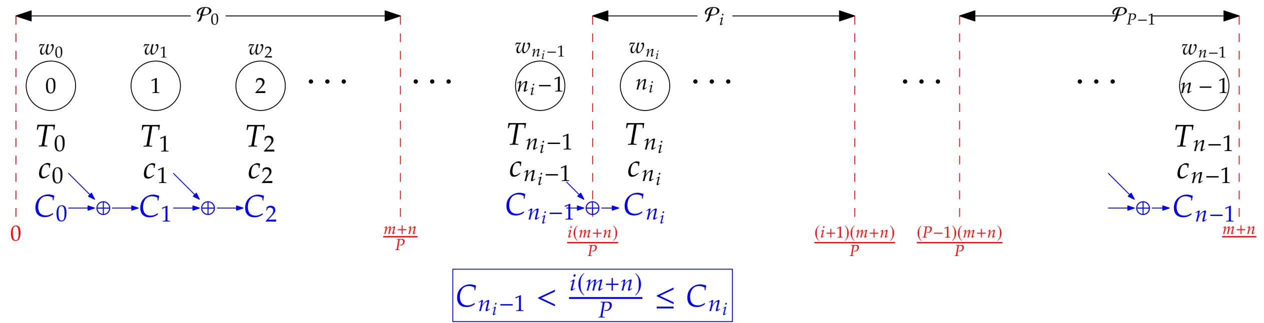


Figure 4.2: *Uniform cost partitioning (UCP) scheme*

Computing C_u in Parallel. Computing C_u has two difficulties: i) for a vertex u, computing c_u by using Equation 4.1 and 4.2 directly is inefficient and ii) C_u is dependent on C_{u-1}. To overcome the first difficulty, we use the following form of e_u to calculate c_u. From Equation 4.1 we have

$$\begin{aligned} e_u &= \frac{w_u}{S} \sum_{v=u+1}^{n-1} w_v \\ &= \frac{w_u}{S} \left(\sum_{v=0}^{n-1} w_v - \sum_{v=0}^{u} w_v \right) \end{aligned}$$

$$= \frac{w_u}{S}\left(\sum_{v=0}^{n-1} w_v - \sum_{v=0}^{u-1} w_v - w_u\right)$$

$$c_u = e_u + 1 = \frac{w_u}{S}(S - \sigma_u - w_u) + 1 \qquad \text{where } \sigma_u = \sum_{v=0}^{u-1} w_v. \tag{4.6}$$

Therefore, c_u can be computed by successively updating $\sigma_u = \sigma_{u-1} + w_{u-1}$.

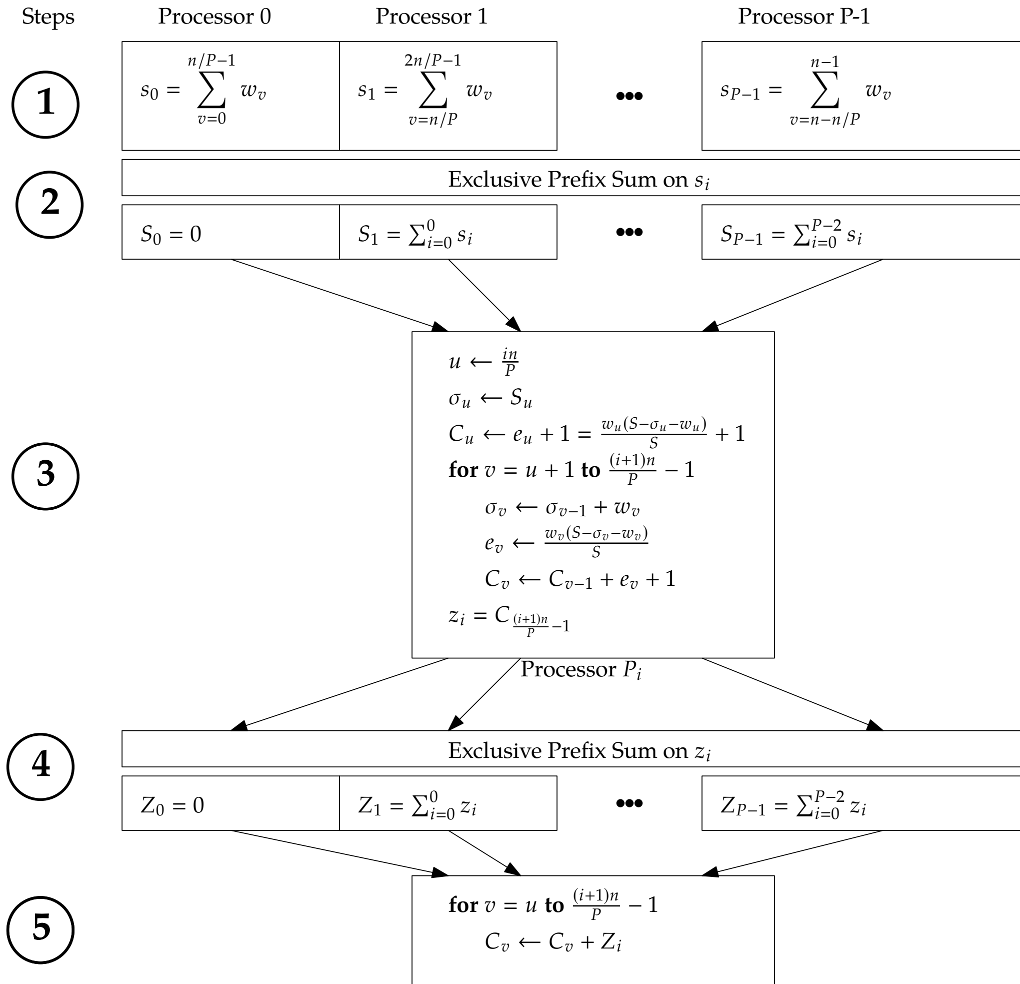


Figure 4.3: *Steps for determining cumulative cost in UCP*

To deal with the second difficulty, we compute C_u in several steps using procedure CALC-COST() as shown in Algorithm 4 (see Figure 4.3 for a visual representation of the algorithm). In each processor, the partitioning algorithm starts with procedure UCP() that calculates the cumulative costs using procedure CALC-COST(). Then procedure MAKE-PARTITION() is used to compute the partitioning boundaries. At the beginning of the CALC-COST() procedure, the task of computing costs for the n vertices are distributed

Algorithm 4: Uniform Consecutive Partitioning

```
// global:  i ← processor id

1  Procedure UCP(V, w, S)
2      Calc-Cost(w, V, S)
3      Make-Partition(w, V, S)

4  Procedure Calc-Cost(w, V, S)
5      s_i ← Σ_{u=i n/P}^{(i+1) n/P − 1} w_u
6      In Parallel: S_i ← Σ_{j=0}^{i−1} s_j
7      u ← in/P
8      σ_u ← S_i
9      C_u ← e_u + 1 = (w_u/S)(S − σ_u − w_u) + 1
10     for u = in/P + 1 to (i+1)n/P − 1 do
11         σ_u ← σ_u + w_u
12         e_u ← (w_u/S)(S − σ_u − w_u)
13         C_u ← C_{u−1} + e_u + 1
14     z_i ← C_{(i+1)n/P − 1}
15     In Parallel: Z_i ← Σ_{j=0}^{i−1} z_j
16     for u = in/P to (i+1)n/P − 1 do
17         C_u = C_u + Z_i

18 Procedure Make-Partition(w, V, S)
19     In Parallel: Z ← Σ_{i=0}^{P−1} z_i
20     Z̄ ← Z/P
21     Find-Boundaries(in/P, (i+1)n/P − 1, C, Z̄)
22     forall n_k ∈ B_i do
23         Send n_k to 𝒫_k and 𝒫_{k+1}
24     Receive boundaries n_i and n_{i+1}
25     return V_i = [n_i, n_{i+1} − 1]

26 Procedure Find-Boundaries(s, e, C, Z̄)
27     if ⌊C_s/Z̄⌋ = ⌊C_e/Z̄⌋ then
28         return
29     m ← (e+s)/2
30     if ⌊C_m/Z̄⌋ ≠ ⌊C_{m+1}/Z̄⌋ then
31         n_{⌊C_{m+1}/Z̄⌋} ← m + 1
32     Find-Boundaries(s, m, C, Z̄)
33     Find-Boundaries(m + 1, e, C, Z̄)
```

among the P processors equally, i.e., processor $\mathcal{P}_i$ is responsible for computing costs for the vertices from $i\frac{n}{P}$ to $(i+1)\frac{n}{P}-1$. Note that these are the vertices that processor $\mathcal{P}_i$ works with while executing the partitioning algorithm to find the boundaries of the subsets.

In Step **1** (Line 5), $\mathcal{P}_i$ computes a partial sum $s_i = \sum_{u=\frac{in}{P}}^{\frac{(i+1)n}{P}-1} w_u$ independently of other processors. In Step **2** (Line 6), *exclusive prefix sum* $S_i = \sum_{j=0}^{i-1} s_j$ is calculated for all s_i, where $0 \le i \le P-1$ and $S_0 = 0$. This exclusive prefix sum can be computed in parallel in $O(\log P)$ time [128]. We have

$$S_i = \sum_{j=0}^{i-1} s_j = \sum_{j=0}^{i-1} \sum_{u=\frac{jn}{P}}^{\frac{(j+1)n}{P}-1} w_u = \sum_{u=0}^{\frac{in}{P}-1} w_u = \sigma_{\frac{in}{P}}.$$

In Step **3**, $\mathcal{P}_i$ partially computes C_u, where $\frac{in}{P} \le u < \frac{(i+1)n}{P}$. By assigning $\sigma_{\frac{in}{P}} = S_i$, $C_{\frac{in}{P}}$ is determined partially using Equation 4.6 in constant time (Line 9). For each u, values of σ_u,

e_u, and C_u are also determined in constant time (Line 10-13), where $\frac{in}{P} + 1 \leq u \leq \frac{(i+1)n}{P} - 1$. After Step **3**, we have $C_u = \sum_{v=\frac{in}{P}}^{u} c_v$. To obtain the final value of $C_u = \sum_{v=0}^{u} c_v$, the value $\sum_{v=0}^{v=\frac{in}{P}-1} c_v$ needs to be added. For a processor $\mathcal{P}_i$, let $z_i = C_{\frac{(i+1)n}{P}-1} = \sum_{v=\frac{in}{P}}^{\frac{(i+1)n}{P}-1} c_v$. In Step **4** (Line 15), another exclusive parallel prefix sum operation is performed on z_i so that

$$Z_i = \sum_{j=0}^{i-1} z_j = \sum_{j=0}^{i-1} \sum_{v=\frac{jn}{P}}^{\frac{(j+1)n}{P}-1} c_v = \sum_{v=0}^{\frac{in}{P}-1} c_v.$$

Note that Z_i is exactly the value required to get the final cumulative cost C_u. In Step **5** (Lines 16-17), Z_i is added to C_u for $\frac{in}{P} \leq u \leq \frac{(i+1)n}{P} - 1$.

Finding Partition Boundaries in Parallel. The partition boundaries are determined using Equation 4.5. The procedure `Make-Partition()` generates the partition boundaries. In Line 19, the parallel sum is performed on z_i to determine $Z = \sum_{0}^{P-1} z_i = \sum_{0}^{n-1} c_u = n + m$, the total cost and $\overline{Z} = \frac{Z}{P}$, the average cost per processor (Line 20). `Find-Boundaries()` is called to determine the boundaries (Line 21). From Equation 4.5, it is easy to show that a partition boundary is found between two consecutive vertices u and $u + 1$, such that $\left\lfloor \frac{C_u}{\overline{Z}} \right\rfloor \neq \left\lfloor \frac{C_{u+1}}{\overline{Z}} \right\rfloor$. Vertex $u + 1$ is the lower boundary of subset V_i, where $i = \left\lfloor \frac{C_{u+1}}{\overline{Z}} \right\rfloor$. $\mathcal{P}_i$ executes `Find-Boundaries()` from vertices $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. `Find-Boundaries()` is a divide & conquer based algorithm to find all the boundaries in that range efficiently using the cumulative costs. All the found boundaries are stored in a local list. In Line 28, it is determined whether the range contains any boundary. If the range does not have any boundary, i.e., if $\left\lfloor \frac{C_s}{\overline{Z}} \right\rfloor = \left\lfloor \frac{C_e}{\overline{Z}} \right\rfloor$, the algorithm returns immediately. Otherwise, it determines the middle of the range m in Line 29. In Line 30, the existence of a boundary between m and $m + 1$ is evaluated. If $m + 1$ is indeed a lower partition boundary, it is stored in local list in Line 31. In Line 32 and 33, `Find-Boundaries()` is called with the ranges $[s, m]$ and $[m + 1, e]$ respectively. Note that the range $\left[\frac{in}{P}, \frac{(i+1)n}{P} - 1\right]$ may contain none, one or more boundaries. Let B_i be the set of those boundaries. Once the set of boundaries B_i, for all i, are determined, the processors exchange these boundaries with each other as follows. Vertex n_k, in some B_i, is the boundary between the subsets V_k and V_{k+1}, i.e., $n_k - 1$ is the upper boundary of V_k, and n_k is the lower boundary of V_{k+1}. In Line 22, for each n_k in the range $\left[\frac{in}{P}, \frac{(i+1)n}{P} - 1\right]$, processor $\mathcal{P}_i$ sends a boundary message containing n_k to processors $\mathcal{P}_k$ and $\mathcal{P}_{k+1}$. Notice that each processor i receives exactly two boundary messages from other processors (Line 24), and these two messages determine the lower and upper boundary of the i-th subset V_i. That is, now each processor i has subset V_i and is ready to execute the parallel algorithm for the CL model with the UCP scheme.

The run time of parallel Algorithm 4 is $\mathcal{O}\left(\frac{n}{P}+P\right)$ as shown in Theorem 10.

Theorem 10. *The parallel algorithm for determining the partition boundaries of the UCP scheme runs in $\mathcal{O}\left(\frac{n}{P}+P\right)$ time, where n and P are the number of vertices and processors, respectively.*

Proof. The parallel algorithm for determining the partition boundaries is shown in Algorithm 4. For each processor, Line 5 takes $\mathcal{O}\left(\frac{n}{P}\right)$ time. The exclusive parallel prefix sum operation requires $\mathcal{O}(\log P)$ time in Line 6. Lines 7-9 take constant time. The for loop at Line 10 iterates $\frac{n}{P}-1$ times. Each execution of the for loop takes constant time for Lines 11-13. Hence, the for loop at Line 10 takes $\mathcal{O}\left(\frac{n}{P}\right)$ time. The prefix sum in Line 15 takes $\mathcal{O}(\log P)$ time. The for loop at Line 16 takes $\mathcal{O}\left(\frac{n}{P}\right)$ time.

The parallel sum operation in Line 19 takes $\mathcal{O}(\log P)$ time using `MPI_Reduce` function. For each processor $\mathcal{P}_i$, n_k's are determined in `FIND-BOUNDARIES()` on the range of $\left[\frac{in}{P}, \frac{(i+1)n}{P}-1\right]$. Finding a single partition boundary on these $\frac{n}{P}$ vertices require $\mathcal{O}\left(\log\frac{n}{P}\right)$ time. If the range contains x partition boundaries, then it takes $\mathcal{O}\left(\min\left\{\frac{n}{P}, x\log\frac{n}{P}\right\}\right)$ time. For each partition boundary n_k, processor i sends exactly two messages to the processors $\mathcal{P}_k$ and $\mathcal{P}_{k-1}$. Thus each processor receives exactly two messages. There are at most P boundaries in $\left[\frac{in}{P}, \frac{(i+1)n}{P}-1\right]$. Thus, in the worst case, a processor may need to send at most $2P$ messages, which takes $\mathcal{O}(P)$ time. Therefore, the total time in the worst case is $\mathcal{O}\left(\frac{n}{P}+\min\left\{\frac{n}{P}, P\log\frac{n}{P}\right\}+P\right)=\mathcal{O}\left(\frac{n}{P}+P\right)$. □

Theorem 10 shows the worst case run time of $\mathcal{O}\left(\frac{n}{P}+P\right)$. Notice that this bound on time is obtained considering the case that all P partition boundaries n_k can be in a single processor. However, in most real-world networks, it is an unlikely event, especially when the number of processors P is large. Thus it is safe to say that for most practical cases, this algorithm will scale to a larger number of processors than the run time analysis suggests. Now

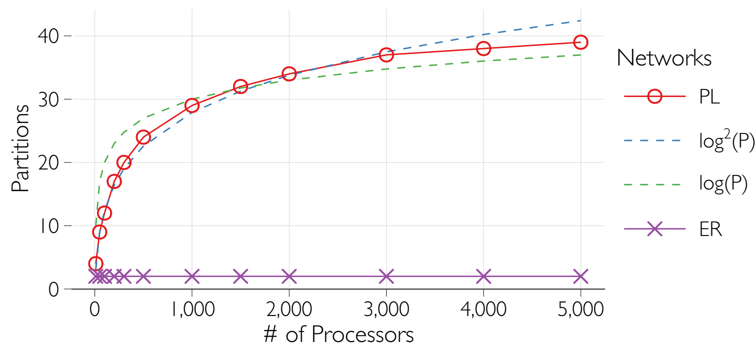


Figure 4.4: *The maximum number of partition boundaries in a single processor*

we experimentally show the number of partition boundaries found in the first subset for some popular networks. For the ER networks, the maximum number of boundaries in a processor is 2, regardless of the number of processors. Even for the power-law networks, which has very skewed degree distribution, the maximum number of boundaries in a single processor is very small. Figure 4.4 shows the maximum number of boundaries found in a single processor. Two fitted plots of $\log^2 P$ and $\log P$ is added in the figure for comparison. From the trend, it appears the maximum number of partition boundaries in a processor is somewhere between $O(\log P)$ and $O\left(\log^2 P\right)$. Since power-law has one of the most skewed degree distribution among real-world networks, we can expect the run time to find partition boundaries to be approximately $O\left(\frac{n}{P} + \log^2 P\right)$ time. The basic steps of our parallel algorithm for the Chung–Lu model using the UCP scheme is summarized in Algorithm 5.

Algorithm 5: Parallel Chung–Lu Algorithm

```
1 Procedure PARALLEL-CHUNG-LU(w)
      // i ← processor id
2     S_i ← Σ_j w_j                        // i(n/P) ≤ j < (i+1)(n/P)
3     In Parallel: S ← Σ_{j=0}^{P-1} S_j
4     In Parallel: V_i ← UCP(V, w, S)
5     E(i) ← CREATE-EDGES(w, S, V_i)
```

Using the UCP scheme, our parallel algorithm for generating random networks with the CL model runs in $O\left(\frac{m+n}{P} + P\right)$ time as shown in Theorem 12. To prove Theorem 12, we need a bound on computation cost which is shown in Theorem 11.

Theorem 11. *The computational cost in each processor is $O\left(\frac{m+n}{P}\right)$ w.h.p.*

Proof. For each $u \in V_i$ and $v > u$, (u, v) is a potential edge in processor $\mathcal{P}_i$, and $\mathcal{P}_i$ creates the edge with probability $p_{u,v} = \frac{w_u w_v}{S}$ where $S = \sum_{v \in V} w_v$. Let x be the number of potential edges in $\mathcal{P}_i$, and these potential edges are denoted by $f_1, f_2, \ldots, f_x$ (in any arbitrary order). Let X_k be an indicator random variable such that $X_k = 1$ if $\mathcal{P}_i$ creates f_k and $X_k = 0$ otherwise. Then the number of edges created by $\mathcal{P}_i$ is $X = \sum_{k=1}^{x} X_k$.

As discussed in Section 4.2, generating the edges efficiently by applying the edge skipping technique is stochastically equivalent to generating each edge (u, v) independently with probability $p_{u,v} = \frac{w_u w_v}{S}$. Let ξ_e be the event that edge e is generated. Regardless of the occurrence of any event ξ_e with $e \neq (u, v)$, we always have $\Pr[\xi_{(u,v)}] = p_{u,v} = \frac{w_u w_v}{S}$. Thus, the events ξ_e for all edges e are mutually independent. Following the definitions and

formalism given in Section 4.3.1, we have the expected number of edges created by $\mathcal{P}_i$, denoted by μ, as

$$\mu = E[X] = \sum_{u \in V_i} e_u = m_i.$$

Now we use the following standard Chernoff bound for independent indicator random variables and for any $0 < \delta < 1$,

$$\Pr\left[X \geq (1+\delta)\mu\right] \leq e^{-\delta^2 \frac{\mu}{3}}.$$

Using this Chernoff bound with $\delta = \frac{1}{2}$, we have

$$\Pr\left[X \geq \frac{3}{2} m_i\right] \leq e^{-\frac{m_i}{12}} \leq \frac{1}{m_i^3}$$

for any $m_i \geq 189$. We assume $m \gg P$ and consequently $m_i > P$ for all i.

Now using the union bound,

$$\Pr\left[X \geq \frac{3}{2} m_i\right] \leq m_i \frac{1}{m_i^3} = \frac{1}{m_i^2}$$

for all i simultaneously. Then with probability at least $1 - \frac{1}{m_i^2}$, the computation cost $X + |V_i|$ is bounded by $\frac{3}{2}m_i + |V_i| = O\left(m_i + |V_i|\right)$. By construction of the partitions by our algorithm, we have $O\left(m_i + |V_i|\right) = O\left(\frac{m+n}{P}\right)$. Thus the computation cost in all processors is $O\left(\frac{m+n}{P}\right)$ w.h.p. □

Theorem 12. *Our parallel algorithm with the UCP scheme for generating random networks with the CL model runs in* $O\left(\frac{m+n}{P} + P\right)$ *time w.h.p.*

Proof. Computing the sum S in parallel takes $O\left(\frac{n}{P} + \log P\right)$ time. Using the UCP scheme, vertex partitioning takes $O\left(\frac{n}{P} + P\right)$ time (Theorem 10). In the UCP scheme, each subset has $O\left(\frac{m+n}{P}\right)$ computation cost w.h.p. (Theorem 11). Thus creating edges using procedure Create-Edges() requires $O\left(\frac{m+n}{P}\right)$ time, and the total time is $O\left(\frac{n}{P} + P + \frac{m+n}{P}\right) = O\left(\frac{m+n}{P} + P\right)$ w.h.p. □

4.3.2 Round-Robin Partitioning (RRP)

In the RRP scheme, vertices are distributed in a round robin fashion. Subset V_i has the vertices $\langle i, i+P, i+2P, \ldots, i+kP \rangle$ such that $i + kP \leq n < i + (k+1)P$; i.e., $V_i = \{j | j$

mod $P = i\}$. In other words vertex i is assigned to $V_{i \bmod P}$. The number of vertices in each subset is almost equal, either $\lfloor \frac{n}{P} \rfloor$ or $\lceil \frac{n}{P} \rceil$.

To compare the computational cost, consider two subsets V_i and V_j with $i < j$. Now, for the x-th vertices in these two subsets, we have: $c_{i+(x-1)P} \geq c_{j+(x-1)P}$ as $i + (x-1)P < j + (x-1)P$ (see Lemma 8). Therefore, $c(V_i) = \sum_{u \in V_i} c_u \geq c(V_j) = \sum_{u \in V_j} c_u$, and, by the definition of the RRP scheme, $|V_i| \geq |V_j|$. The difference in cost between any two subsets is at most w_0, the maximum weight as shown in Lemma 13.

Lemma 13. *In the Round Robin Partitioning (RRP) scheme, for any $i < j$, we have $c(V_i) - c(V_j) \leq w_i$.*

Proof. The difference in cost between two subsets V_i and V_j is given by

$$
\begin{aligned}
c(V_i) - c(V_j) &= \sum_{u \in V_i} c_u - \sum_{u \in V_j} c_u = \sum_{x=0}^{k} \left(c_{i+xP} - c_{j+xP}\right) \\
&= c_i - \sum_{x=0}^{k-1} \left(c_{j+xP} - c_{i+(x+1)P}\right) - c_{j+kP} \\
&\leq c_i - c_{j+kP} \qquad \left[c_{j+xp} \geq c_{i+(x+1)P}\right] \\
&\leq e_i \\
&= \frac{w_i}{S} \sum_{v=i+1}^{n-1} w_v \\
&< \frac{w_i}{S} S = w_i.
\end{aligned}
$$

□

Thus, the RRP scheme provides quite good load balancing. However, it is not as good as the UCP scheme. It is easy to see that in the RRP scheme, for any two subsets V_i and V_j such that $i < j$, we have $c(V_i) > c(V_j)$. But, by design, the UCP scheme makes the subset such that costs are equally distributed among the processors. Furthermore, although the RRP scheme is simple to implement and provides quite good load balancing, it has another subtle problem. In this scheme, the vertices of a partition are not consecutive and are scattered in the entire range leading to some serious efficiency issues in accessing these vertices. One major issue is that locality of reference is not maintained, leading to a very high rate of cache misses during the execution of the algorithm. This contrast of performance between UCP and RRP is even more prominent when the goal is to generate massive networks as shown by experimental results in Section 4.5.

4.4 Space-Efficient Parallel Algorithm for the CL Model

The time efficient algorithm for the CL model requires the weights of all vertices to be stored in each processor, which requires $O(n)$ memory per processor. When generating a network with a huge number of vertices, it might not be possible to fit the weights of all vertices in each processor's memory. In this section, we present a space-efficient version of the parallel algorithm for the CL model that only requires $O(\frac{n}{P})$ memory.

In this algorithm, the set of vertices V is partitioned into P subsets $V_1, V_2, \ldots, V_{P-1}$. Subset V_i includes the vertices from $\frac{in}{P}$ to $\frac{(i+1)n}{P}$ and assigned to processor $\mathcal{P}_i$. Note that the vertex $\frac{(i+1)n}{P}$ is common in both subsets V_i and V_{i+1} where $0 \geq i < P-1$. Processor $\mathcal{P}_i$ stores the weights $\{w_u : u \in V_i\}$. Hence, each processor requires $O(\frac{n}{P})$ memory to store the weights.

Similar to the time efficient algorithm, processor $\mathcal{P}_i$ also computes the sum of weights S in parallel. However, unlike that algorithm, the tasks $\{T_u : u \in V_i\}$ for subset V_i are executed in a different manner. Executing a task T_u requires the weights from w_u to w_{n-1}, however a processor does not have the full weight array. A straightforward way to overcome this difficulty is to collect the required weights from other processors. Using this method requires a excessive message communication, which leads to poor performance. To avoid unnecessary data movements, task T_u is distributedly computed by many processors in our space-efficient algorithm. Note that the task $T_{\frac{(i+1)n}{P}}$ will not be started in processor $\mathcal{P}_i$, rather it will be initiated by processor $\mathcal{P}_{i+1}$. For any task T_u, the algorithm starts with vertex $v = u + 1$, adds a random skip length δ computed from the probability $\frac{w_u w_v}{S}$ and selects vertex $v + \delta$. If this vertex is in the same subset V_i then edge $(u, v + \delta)$ is created with probability $\frac{q}{p}$, where $q = \frac{w_u w_{v+\delta}}{S}$, and the process of skipping and selecting edges is repeated with $v = v + \delta + 1$.

If a selected vertex v is in some other subset V_j, then a token message $\langle u, v, w_u \rangle$ is sent to processor $\mathcal{P}_j$. At this point, execution of task T_u is stopped in processor $\mathcal{P}_i$ and is to be resumed by processor $\mathcal{P}_j$ upon receiving the message. Processor $\mathcal{P}_i$ continues to process the next task or any message it received.

Upon receiving a token message $\langle u', v', w' \rangle$ from processor $\mathcal{P}_i$, processor $\mathcal{P}_j$ resumes task $T_{u'}$. Note that only $w_{u'}$ is required to resume the task, which is given in the message from processor $\mathcal{P}_i$. Resuming the task means setting the starting vertex $v = v'$ and repeating the process of skipping and selecting edges within the partition boundary.

The pseudocode of the algorithm is given in Algorithm 6. Note that, in Lines 4-6, all available messages are processed first. This is done so that the sent messages do not fill the MPI message buffer while waiting to be received.

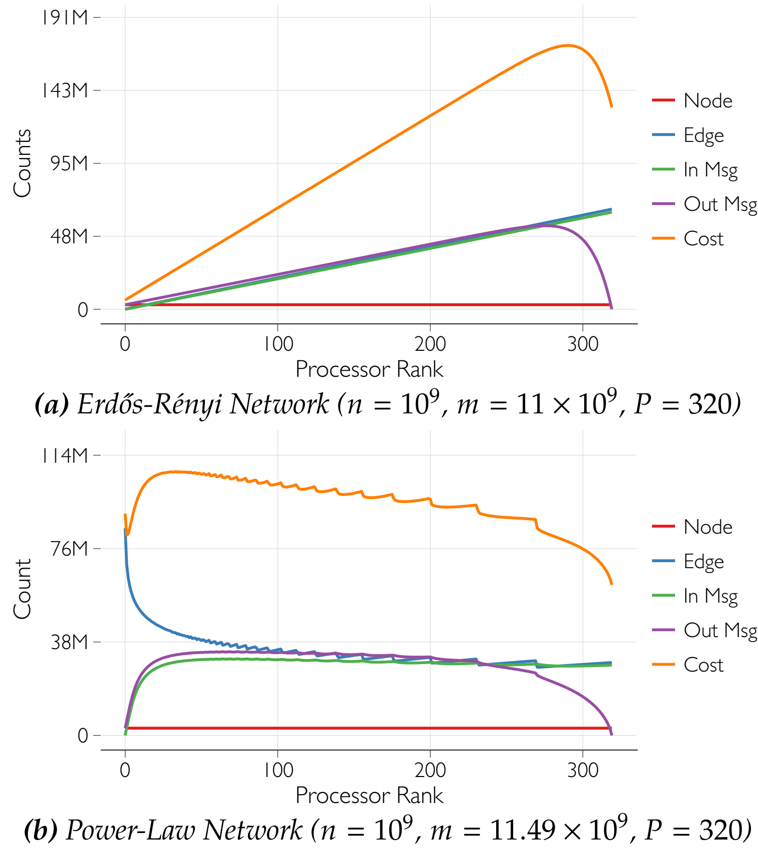
Algorithm 6: Space-efficient CL Algorithm

```
// global:  i ← processor id

1  Procedure CL-SPACE-EFFICIENT(w)
2      In Parallel: S ← ∑_k w_k
3      forall u ∈ V_i do
4          while message is available to receive do
5              ⟨u′, v′, w′⟩ ← received message
6              NODETASK(u′, v′, w′)
7          NODETASK(u, u + 1, w_u)
8      forall remaining messages do
9          ⟨u′, v′, w′⟩ ← received message
10         NODETASK(u′, v′, w′)

11 Procedure NODETASK(u, v, w)
12     p ← min(w · w_v / S, 1)
13     while p > 0 do
14         if v ∈ V_i then
15             compute skip length δ
16             v ← v + δ                                    // skip δ edges
17             if v ∈ V_i then
18                 q ← min(w · w_v / S, 1)
19                 create edge (u, v) with probability q/p
20                 p ← q, v ← v + 1
21         else
22             Send message ⟨u, v, w_u⟩ to P_j where v ∈ V_j
23             break
```

In the space-efficient version of the parallel CL algorithm, each processor has the same number of vertices but the number of edges generated by each processor varies significantly. Each processor $\mathcal{P}_i$ send tokens only to higher ranked processors $\mathcal{P}_j$, where $j > i$. It is easy to see that higher ranked processors can receive at most n tokens. A processor $\mathcal{P}_i$, while working on task T_u, may select the next vertex v that is in processor $\mathcal{P}_j$ such that $j > i + 1$ (Line 22 in Algorithm 6). Then, processor $\mathcal{P}_{i+1}$ to $\mathcal{P}_{j-1}$ will not receive any message for task T_u. There are at most n tasks and some processors may not receive tokens for some of the tasks.



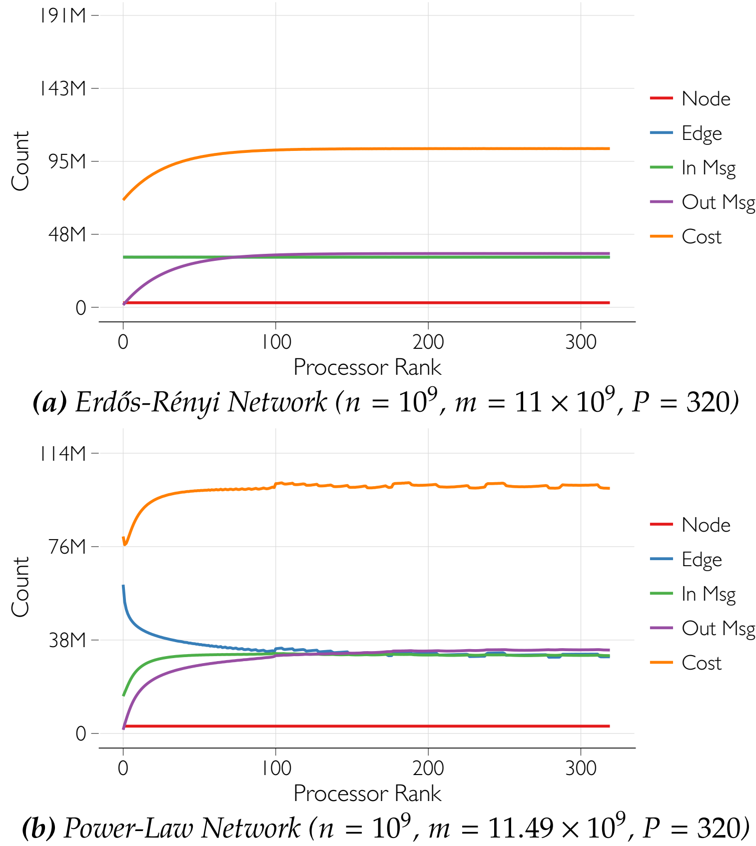
(a) *Erdős-Rényi Network* ($n = 10^9$, $m = 11 \times 10^9$, $P = 320$)

(b) *Power-Law Network* ($n = 10^9$, $m = 11.49 \times 10^9$, $P = 320$)

Figure 4.5: *Distribution of vertices, edges and messages for space-efficient parallel CL algorithm*

Figure 4.5 shows the workload for the Erdős-Rényi and Power-Law networks. In the figure, cost refers to the combined number of vertices, edges, and incoming and outgoing messages. This experimental result shows that the number of tokens received by a processor is significantly smaller than n.

In Figure 4.5(a), observe that the number of edges produced in Erdős-Rényi network increases with increasing processor rank. In contrast, for the power-law network, the number of edges produced decreases with increasing processor rank. With the objective of improving load balancing, we study another approach, where the set of vertices V is partitioned into $2P$ subsets with equal number of vertices where each subset contains consecutive set of vertices. Let the subsets be $V_0, V_1, \ldots, V_{2P-1}$. Each processor $\mathcal{P}_i$ is assigned two subsets: subset V_i and V_{2P-1-i}. The rest of the algorithm is basically similar to the previous space-efficient algorithm. The only difference is that processor $\mathcal{P}_i$ works on two subsets instead of one subset. Figure 4.6 shows the load distribution of the algorithm with $2P$ subsets. Observe that this algorithm provides better edge and message distribution than the previous space-efficient version.



(a) *Erdős-Rényi Network* ($n = 10^9$, $m = 11 \times 10^9$, $P = 320$)

(b) *Power-Law Network* ($n = 10^9$, $m = 11.49 \times 10^9$, $P = 320$)

Figure 4.6: *Space-efficient parallel CL algorithm with* $2P$ *subsets*

4.5 Experimental Results

In this section, we experimentally show the accuracy and performance of our algorithm. The accuracy of our parallel algorithms is demonstrated by showing that the generated degree distributions closely match the input degree distribution. The strong scaling of our algorithm shows that it scales very well to a large number of processors. We also present experimental results showing the impact of the partitioning schemes on load balancing and performance of the algorithm.

Experimental Setup. We used an 81-node HPC cluster for the experiments. Each node is powered by two octa-core SandyBridge E5-2670 2.60GHz (3.3GHz Turbo) processors with 64 GB memory. The algorithm is developed with MPICH2 (v1.7), optimized for QLogic InfiniBand cards.

Graph Datasets. In the experiments, degree distributions of real-world and artificial random networks were considered. The list of networks is shown in Table 4.1. The run

time does not include the I/O time to write the graph to disk.

Table 4.1: *Networks used in the experiments*

Network	Type	Nodes	Edges
PL	Power-Law Distribution	1**B**	249**B**
PL-Small	Power-Law Distribution	1**M**	11.5**B**
ER	Erdős-Rényi Network	1**M**	200**M**
Miami [16]	Contact Network	2.1**M**	51.4**B**
Twitter [162]	Real-World Social Network	41.65**M**	1.37**B**
Friendster [161]	Real-World Social Network	65.61**M**	1.81**B**

Degree Distribution of Generated Networks. Figure 4.7 shows the input and generated degree distributions for ER, PL, Miami, Twitter, and Friendster networks. As observed from the plots, the generated degree distributions closely follow the input degree distributions reassuring that our parallel algorithms generate random networks with given expected degree sequences accurately.

As a formal test, we use the Kullback-Leibler (KL) divergence [82] to compute the statistical difference between the input and output degree distributions. The KL divergence measures the difference between two probability distributions Q (input) and R (output) as information gain defined as:

$$D_{\mathrm{KL}}(Q \| R) = \sum_i Q(i) \log \frac{Q(i)}{R(i)}. \tag{4.7}$$

In other words, it measures the amount of information lost (in number of bits) when the output distribution R is used in place of the input distribution Q. Table 4.2 summarizes the KL-Divergence test results. The average minimum number of bits needed for each entry of input distribution Q are shown in column two of the table. The KL divergences between the input and output degree distributions of our parallel algorithm for are shown in column three. The difference of number of bits required in percentage is presented in column four. Note that the differences are negligible and expected due to the randomness of the network model.

Effect of Partitioning Schemes. As discussed in Section 4.3.1, partitioning significantly affects load balancing and performance of the algorithm. We demonstrate the effects of the

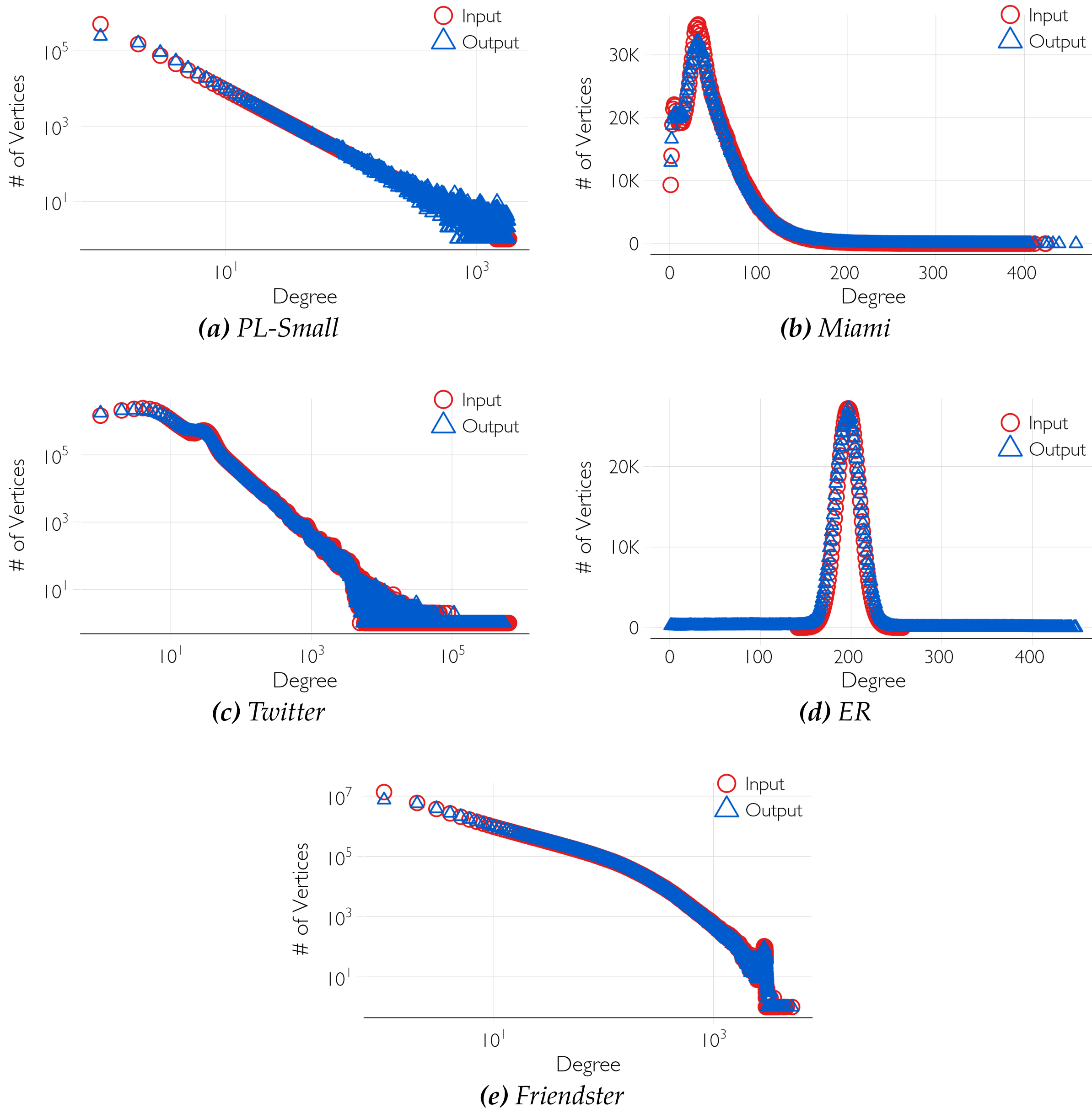
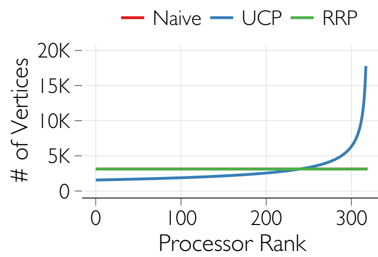


Figure 4.7: *Degree distributions of input and generated degree sequences*

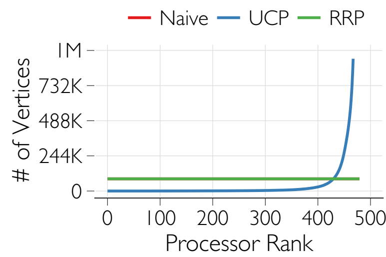
partitioning schemes in terms of computing time in each processor as shown in Figure 4.8 using ER, Twitter, and PL networks. Computational time for the naïve scheme is skewed. For all the networks, the computational times for the UCP and RRP scheme stay almost constant in all processors, indicating good load-balancing. The RRP scheme is a little slower than the UCP scheme because the locality of references is not maintained in RRP, leading to high cache miss as discussed in Section 4.3.2.

Table 4.2: *KL Divergence Test for the Input and Output Degree Distributions*

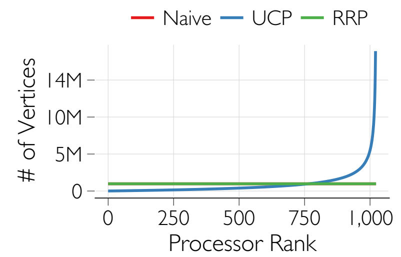
	Minimum required bits	$D_{KL}(Q\|R)$	% difference
PL-Small	3.117437	0.125099	4.01%
ER	5.699510	0.095964	1.68%
Miami	6.900232	0.002102	0.03%
Twitter	6.307555	0.007157	0.11%
Friendster	6.171355	0.038343	0.62%



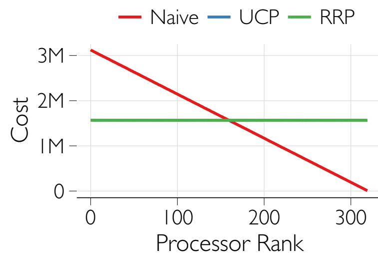
(a) *Number of Vertices (ER)*



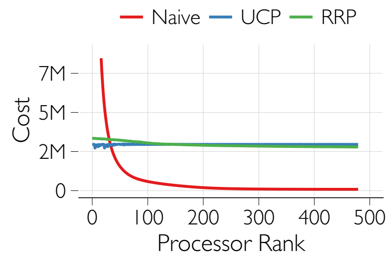
(b) *Number of Vertices (Twitter)*



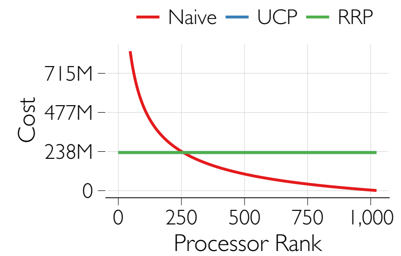
(c) *Number of Vertices (PL)*



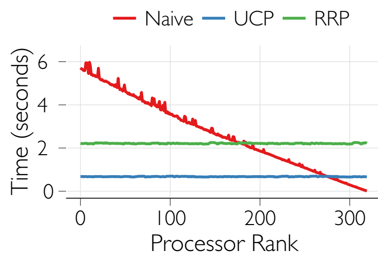
(d) *Computational Cost (ER)*



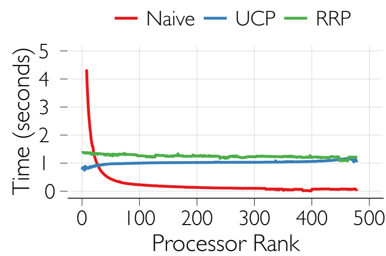
(e) *Computational Cost (Twitter)*



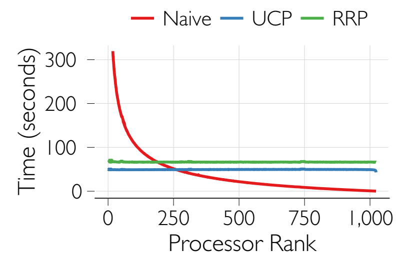
(f) *Computational Cost (PL)*



(g) *Computational Time (ER)*



(h) *Computational Time (Twitter)*



(i) *Computational Time (PL)*

Figure 4.8: *Comparison of partitioning schemes*

4.5.1 Memory Requirement

Table 4.3 shows the memory required to generate a power-law network with 1B vertices and 249B edges with 1024 processors and the Twitter network. As expected, our space-efficient algorithm requires P times less memory than the time-efficient algorithm to store the given degree sequence. Time-efficient algorithm also requires more memory for computing the partition boundaries.

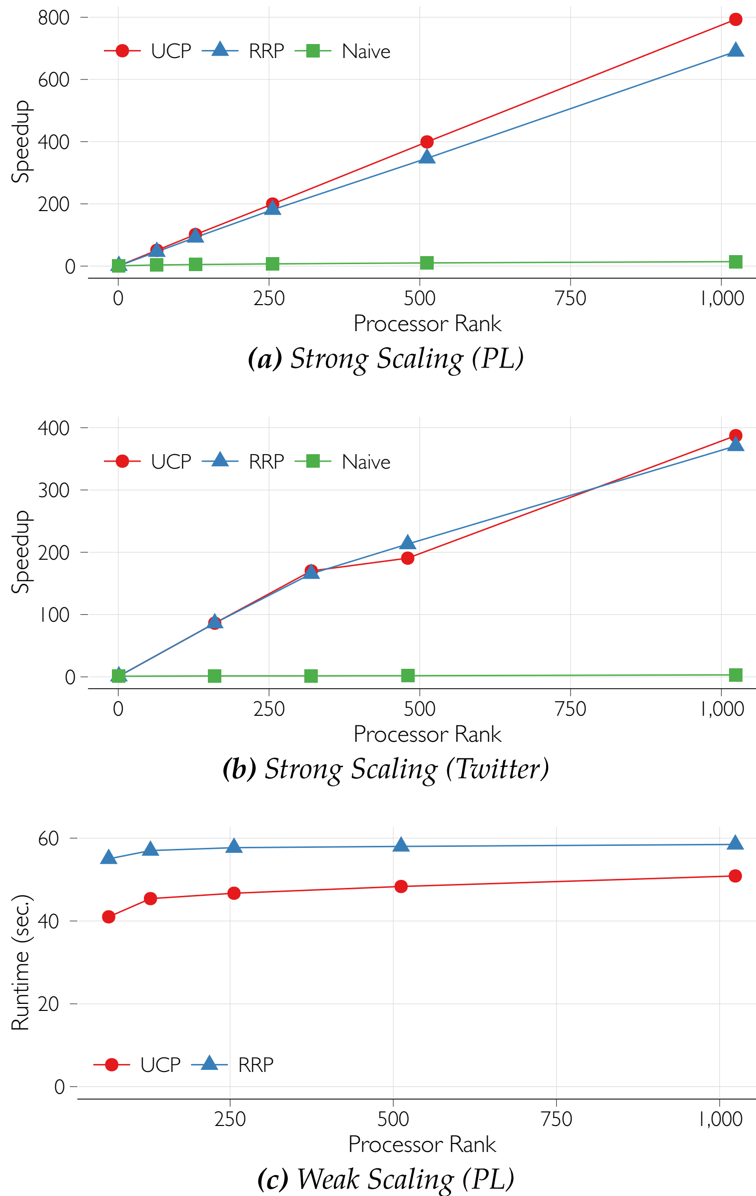
Table 4.3: *Required memory (in MB) for parallel algorithm*

Network	Algorithm	Weight	Computation
Power-Law	Space Efficient	1.98	0.93
	Time Efficient	1,907.36	7.48
Twitter	Space Efficient	0.077	0.9
	Time Efficient	79.44	0.31

Strong and Weak Scaling of Time Efficient Algorithm. Strong scaling of a parallel algorithm shows its performance with the increasing number of processors, while keeping the problem size fixed. Figure 4.9 shows the speedup of the naïve, UCP, and RRP partitioning schemes using the PL and Twitter networks. Speedups are measured as $\frac{T_s}{T_p}$, where T_s and T_p are the running time of the sequential and the parallel algorithm, respectively. The number of processors were varied from 1 to 1024. As Figure 4.9 shows, UCP and RRP achieve excellent linear speedups. The naïve scheme performs the worst as expected. The speedup for the PL is greater than that for the Twitter network. As Twitter is smaller than the PL network, the impact of the parallel communication overheads is higher, contributing to decreased speedup. Still, the algorithm to generate the Twitter network has a speedup of 400 using 1024 processors.

The weak scaling measures the performance of a parallel algorithm when the input size per processor remains constant. For this experiment, we varied the number of processors from 16 to 1024. For P processors, a PL network with 10^6P vertices and 10^8P edges is generated. Note that weak scaling can only be performed on artificial networks. Figure 4.9(c) shows the weak scaling for the UCP and RRP schemes using the PL networks. Both RRP and UCP show very good weak scaling with almost constant run time.

Strong Scaling of Space Efficient Algorithm. Figure 4.10 shows the speedup of space-efficient algorithm with P and $2P$ partitions. We used a power-law (n = 1B vertices



(a) *Strong Scaling (PL)*

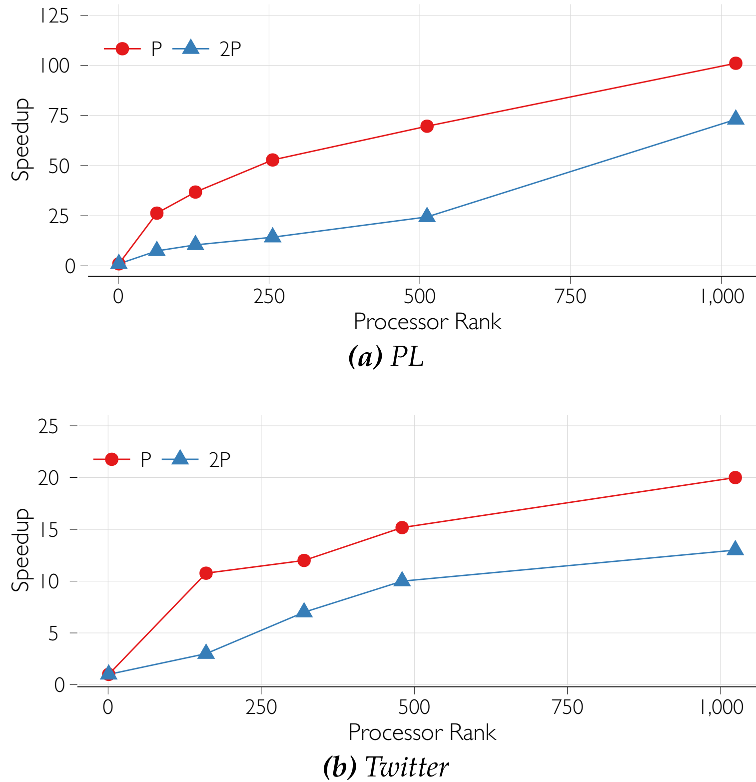
(b) *Strong Scaling (Twitter)*

(c) *Weak Scaling (PL)*

Figure 4.9: *Strong and weak scaling of the parallel algorithms*

and m = 249B edges) and the Twitter networks (n = 41.65M vertices and m = 1.37B edges) for this experiment. To make the algorithm space efficient, we had to sacrifice the time efficiency. It is easy to see that a processor can receive at most n token messages, which might require $\mathcal{O}(n)$ running time. However, in practice, a processor receives fewer messages. The space-efficient algorithm with P partitions and 1024 processors achieves a speedup of 100 for the Power-Law network and 20 for Twitter network.

Though the efficient memory algorithm with $2P$ partitions have better load distribution, in



***(a)** PL*

***(b)** Twitter*

***Figure 4.10:** Strong scaling of the space efficient parallel algorithm*

practice it does not perform well. This is due to the fact that each processor can receive messages from any other processor. Managing and synchronizing the two partitions of the space-efficient parallel algorithm is complex and if affects the run time of the algorithm.

Effect of Average Degree. There are three main parts of our time-efficient algorithm: calculating sum S, computing partition boundaries and creating edges. The first two parts depend on the number of vertices and processors. Creation of edges is dependent on the number of edges to be created. A breakdown of computational costs is given in Table 4.4. In this table, we used three networks using 1024 processors each of which has 1B vertices and different average degrees. As seen from the table, the summation and load balancing time is almost equal for a fixed n and P. With increasing average degree, the computation time to create edges becomes more prominent.

Generating Large Networks. The primary objective of the parallel algorithm is to generate massive random networks. Using the algorithm with the UCP scheme, we have generated power law networks with one billion vertices and 249 billion edges in one minute using 1024 processors with a speedup of about 800.

Table 4.4: *Effect of average degree on run time*

	Time μ-seconds		
Average Degree	Summation	Create Partition	Edge Creation
3.18	116,363 (17.74%)	13,657 (2.08%)	526,087 (80.18%)
38.96	119,665 (2.8%)	12,103 (0.28%)	4,145,957 (96.92%)
497.22	115,488 (0.23%)	12,790 (0.03%)	50,764,215 (99.75%)

4.6 Conclusion

We have developed two efficient parallel algorithms for generating massive networks with a given expected degree sequence using the Chung-Lu model. The main challenge in developing these algorithms is load balancing. To overcome this challenge, we have developed a novel parallel algorithm for balancing computational loads that results in a significant improvement in efficiency. We believe that the presented parallel algorithm for the Chung-Lu model will prove useful for modeling and analyzing emerging massive complex systems and uncovering patterns that emerges only in massive networks. As the algorithm can generate networks from any given degree sequence, its application will encompass a wide range of complex systems. While one of our algorithms is time-efficient and the other one is space-efficient, it remains an open problem to develop a parallel algorithm that is simultaneously time- and space-efficient with both time and space complexity in the ballpark of $O(\frac{m+n}{P} + f(P))$, where $f(P)$ is some polynomial of P with small degree.

Acknowledgment

This work has been partially supported by DTRA Grant HDTRA1-11-1-0016, DTRA CNIMS Contract HDTRA1-11-D-0016-0001, NSF NetSE Grant CNS-1011769, NSF SDCI Grant OCI-1032677, and NSF DIBBs Grant ACI-1443054.

Chapter 5

An Efficient and Scalable Algorithmic Method for Generating Large-Scale Random Graphs

Many real-world systems and networks are modeled and analyzed using various random graph models. These models must incorporate relevant properties such as degree distribution and clustering coefficient. Many models, such as the Chung-Lu (CL), stochastic Kronecker, stochastic block model (SBM), and block two-level Erdős-Rényi (BTER) models have been devised to capture those properties. However, the generative algorithms for these models are mostly sequential and take prohibitively long time to generate large-scale graphs. In this paper, we present a novel time and space efficient algorithmic method to generate random graphs using CL, BTER, and SBM models. First, we present an efficient sequential algorithm and an efficient distributed-memory parallel algorithm for the CL model. Our sequential algorithm takes $O(m)$ time and $O(\Lambda)$ space, where m and Λ are the number of edges and distinct degrees, and our parallel algorithm takes $O(\frac{m}{P} + \Lambda + P)$ time w.h.p. and $O(\Lambda)$ space using P processors. These algorithms are almost time optimal since any sequential and parallel algorithms need at least $\Omega(m)$ and $\Omega(\frac{m}{P})$ time, respectively. Our algorithms outperform the best known previous algorithms by a significant margin in terms of both time and space. Experimental results on various large-scale networks show that both of our sequential and parallel algorithms require 400-15000 times less memory than the existing sequential and parallel algorithms, respectively, making our algorithms suitable for generating very large-scale networks. Moreover, both of our algorithms are about 3-4 times faster than the existing sequential and parallel algorithms. Finally, we show how our algorithmic method also leads to efficient parallel and sequential algorithms for the SBM and BTER models.

5.1 Introduction

The advances of hardware technologies, software, and algorithms have enabled the detailed study of complex systems. These systems, such as the Internet [54, 138], biological networks [63], and social networks [86, 92, 161], are sometimes modeled by random graphs for the purpose of studying their behavior. Degree distribution is one of the most prominent features of these networks. Some well-understood graph models have been developed to capture the diversity of the degree distributions, such as the Erdős-Rényi [51], stochastic block models [73], small-world [153], Barabási-Albert [9, 15], exponential random graph [59, 123], recursive matrix [31], stochastic Kronecker graph [93, 94], and HOT [30] models.

Each of these models has been developed considering some aspect of the networks. The Erdős-Rényi model [51] was the first attempt to perform a systematic study of networks. The stochastic block model [73] has been studied for a long time to study the community structures found in many real-world networks. The small-world model [153] was proposed to capture the small-world property found in many real-world systems. The Barabási-Albert model [15] famously captured the preferential attachment and power-law degree distribution properties. However, these models generate graphs with specific types of degree distributions. The Chung-Lu model [38, 39] can generate a random graph with a given sequence of expected degrees and is capable of generating networks from almost any real-world degree distribution. Another model, called the block two-level Erdős-Rényi (BTER) [80, 134] has been developed recently to study community structure, which can capture both degree distribution and clustering coefficients. A generalization of BTER was proposed in [26]. A joint degree distribution model [140] was also presented to study the assortativity of real-world networks.

There have been some efforts to deal with massive networks. In one such effort, the Graph500 group [65] choose the SKG model in the supercomputer benchmark due to its simpler parallel implementation. The CL model is very similar to the SKG model [117], which could replace the SKG model due to similar properties and ability to generate a wider range of degree distributions. In this chapter, our main focus is on the CL model. The previous best known sequential algorithm for the CL model is given in [105] which takes $O(m + n)$ expected time and $O(n)$ space, where m and n are the number of edges and vertices in the graph. Based on this sequential algorithm, a distributed-memory parallel algorithm is presented in [7], which is the only known parallel algorithm for the CL model. This parallel algorithm takes $O\left(\frac{m+n}{P} + P\right)$ time with high probability (w.h.p.) and $O(n)$ space using P processors.

In this chapter, we present a novel method (called the *DG (Degree-based Grouping) method*), based on grouping the vertices by their degrees, that leads to space and time efficient

algorithms for several random graph models, including the CL model, with rigorous guarantees. Our main contributions are summarized below.

1. Space efficiency: Both our sequential and parallel algorithms for the CL model require only $O(\Lambda)$ space, where Λ is the number of distinct degrees, compared to $O(n)$ space required by the previous algorithms. In real-world networks, Λ is significantly smaller than n. Experimental results on a wide range of large-scale networks show that our algorithms require 400-15000 *times less memory than previous algorithms*. This space efficiency makes our algorithms suitable for generating very large-scale graphs.

2. Time efficiency: Our algorithms are more efficient in terms of run time also. We prove that our sequential and parallel algorithms have running time $O(m)$ and $O\left(\frac{m}{P} + \Lambda + P\right)$, respectively, with high probability (this is discussed formally later), where P denotes the number of processors. In contrast to earlier algorithms, the associated constants and overheads are significantly smaller for our algorithms. Experimental results show that our algorithms are about 3-4 times faster than the previous algorithms. Moreover, our parallel algorithm achieves an *almost optimal load balancing* using an efficient load balancing technique and scales very well to a large number of processors. *Our parallel algorithm can generate a network with* 250 *billion edges in just* 12 *seconds using* 1024 *processors.*

3. Extensions to other models: Finally, we show how our algorithmic method extends naturally to the BTER and SBM models and leads to significantly improved sequential and parallel algorithms. Experimental results show that after applying the DG method, the run time for the BTER model *improves by a factor of* 5-80 for various types and sizes of networks.

The rest of the chapter is organized as follows. In Section 5.2, we describe the problem and the DG method. In Section 5.3, we present the efficient parallel algorithm along with the optimal load balancing technique. In Section 5.4 and 5.5, we present the algorithms for BTER and SBM models, respectively, applying the DG method. Finally, we conclude in Section 5.6.

5.2 Generating random graphs with a desired degree distribution

Many models have been proposed to generate random graphs from a desired degree distribution or sequence. The configuration model [20, 106] is one of the first models that generates a graph with a given degree sequence. This model can generate every possible graph with the given degree sequence with equal probability [106]. However, it

can produce graphs with some undesirable properties such as parallel edges and self-loops that are unacceptable for many applications. The Chung-Lu (CL) model [38, 39] is another widely used model that generates random graphs from a given sequence of expected degrees by avoiding these undesirable properties.

5.2.1 The Chung-Lu Model

Assume that we are given n vertices labeled as $1, 2, \ldots, n$ and a sequence of expected degrees $W = \langle w_1, w_2, \ldots, w_n \rangle$ such that a vertex u has an expected degree of w_u. In the Chung-Lu (CL) model, any pair of vertices u and v are connected by an edge with the probability $p_{u,v} = \frac{w_u w_v}{S}$, where $S = \sum_u w_u$ (assuming $\max_u w_u^2 \leq S$, we have $p_{u,v} \leq 1$ for all u and v) [38, 39]. For simple graphs without self-loops, i.e., $u \neq v$, the expected degree of a vertex u is $\sum_{v \neq u} \frac{w_u w_v}{S} = w_u - \frac{w_u^2}{S}$, which converges to w_u for large graphs.

A naïve algorithm for the CL model is: individually consider each of the $\frac{n(n-1)}{2}$ possible pairs of distinct vertices $\{u, v\}$ and create the edge (u, v) with probability $p_{u,v}$. It requires $O\left(n^2\right)$ time and $O(n)$ space. Pinar et al. [117] presented a sampling-based algorithm (henceforth referred to as the PSK algorithm), where each of the m edges is created by selecting two end points randomly and independently using the degree distribution as the probability distribution. Each end point is sampled in constant time using $O(m)$ memory. This algorithm requires $O(m)$ time and $O(m)$ space. Although this algorithm is simple, it does not eliminate self-loops and parallel edges and requires large memory.

Miller and Hagberg presented an $O(m+n)$ time algorithm (referred to as the MH algorithm) that avoids self-loops or parallel edges and requires $O(n)$ memory [105]. They used an *accept-reject sampling* along with the *edge skipping technique* introduced in [19] for the Erdős-Rényi model. Although this is the fastest known sequential algorithm for the CL model, it has some limitations. It requires the sequence W be sorted in a non-increasing order leading to some computational overhead. Additionally, due to the rejection sampling, some potential edges are rejected. The number of such edges has been shown to be $O(m)$, which also incurs significant computational overhead, especially for skewed degree distributions found in most real-world graphs [105]. Moreover, the MH algorithm needs to generate two random numbers per edge (in contrast to one random number per edge in our algorithm).

We present an algorithm for the CL model using a novel method, called the DG algorithm, which takes $O(m)$ time and $O(\Lambda)$ space, where Λ is the number of distinct values in W. A comparison of the algorithms for the CL model is given in Table 5.1. Notice that $\Lambda < n$, and in most cases, Λ is very small compared to n. Thus, our algorithm requires significantly less memory comparing to the previous algorithms, making our algorithm

suitable for generating large-scale graphs. Furthermore, although the time complexity is similar, the lower overhead of our algorithm leads to smaller constants associated with the time complexity and makes our algorithm approximately three times faster than the MH algorithm.

Table 5.1: *A comparison of the algorithms for the CL model*

Algorithm	Runtime	Space
Naïve	$O(n^2)$	$O(n)$
PSK [117]	$O(m)$	$O(m)$
MH [105]	$O(m+n)$	$O(n)$
DG (this work)	$O(m)$	$O(\Lambda)$

5.2.2 DG: A New Time and Memory Efficient Algorithm

In this section, we present **DG**, a new time and memory efficient algorithm for the CL model. We are given a sequence of n expected degrees $W = \{w_1, w_2, \ldots w_n\}$. Let $\mathbb{D} = \{d_1, d_2, \ldots, d_\Lambda\}$ be the set of all Λ distinct expected degrees in W, and let n_i be the number of vertices with expected degree d_i. Then $\{n_i\}_{1\le i\le\Lambda}$ represents the degree distribution. The number of vertices $n = \sum_{i=1}^{\Lambda} n_i$, and the sum of the degrees $S = \sum_{i=1}^{\Lambda}(d_i n_i)$. DG takes either a sequence of degrees or a degree distribution as input. If a sequence is the input, it is converted into a degree distribution on the fly, without storing the sequence in the memory. Only the degree distribution is stored in the memory, and it takes $O(\Lambda)$ space.

The vertices are grouped by their expected degrees. Let $V_i = \{u \in V : w_u = d_i\}$ be the group of vertices with expected degree d_i, and $n_i = |V_i|$ be the number of vertices in V_i for $1 \le i \le \Lambda$. Now, there can be two types of edges: i) *Intra edges:* where both end-points of an edge (u, v) belong to the same group, i.e., $u, v \in V_i$ for some i, and ii) *Inter edges:* where the two end-points belong to two different groups, i.e., $u \in V_i$ and $v \in V_j$ with $i \ne j$.

Creating Intra Edges. For any $u, v \in V_i$, the edge (u, v) is created with probability $p_{u,v} = \frac{w_u w_v}{S} = \frac{d_i^2}{S}$, since $w_u = w_v = d_i$. Notice that for all pairs of $u, v \in V_i$, the probabilities $p_{u,v}$ are equal. Thus generating the intra edges on V_i is equivalent to generating an Erdős-Rényi (ER) random graph $G(n, p)$ with $n = n_i = |V_i|$ and $p = \frac{d_i^2}{S}$. The ER model $G(n, p)$ generates a random graph with n vertices where each of $\frac{n(n-1)}{2}$ possible potential edges is selected and added to the generated graph with probability p. We generate the intra edges

on V_i for all i by generating ER random graphs $G(n_i, \frac{d_i^2}{S})$. A Naïve algorithm to generate random graph $G(n, p)$ is: for each of the $\frac{n(n-1)}{2}$ potential edges, toss a biased coin and select the edge with probability p. This Naïve algorithm takes $O(n^2)$ time. An efficient algorithm for the ER model is given in [19], which runs in $O(m)$ time. This algorithm uses a technique called edge skipping technique. From a sequence of all potential edges, it selects a subset of the edges using the skipping technique. We also use this edge skipping technique to generate the inter edges. Below we briefly describe the edge skipping technique (see [19] for details).

Edge Skipping Technique. Consider a sequence of potential edges as shown in Figure 5.1. The goal is to select a subset of the edges such that each potential edge is selected with probability p. In Figure 5.1, each circle represents a potential edge, and a black circle represents a selected edge. Notice that before selecting an edge, a sequence of zero or more potential edges is discarded (white circles), e.g., in Figure 5.1, edge e_1 is discarded and e_2 is selected. Then e_3 and e_4 are discarded, and e_5 is selected, and so on. Let ℓ be a random variable, which denotes the number of consecutive edges discarded before selecting the next edge. Then the probability that ℓ edges are discarded is

$$\Pr\left[\ell \text{ edges are discarded}\right] = (1-p)^{\ell} p. \tag{5.1}$$

Notice that ℓ is a geometric random variable. A geometric random number can be generated in constant time from a uniform random number $r \in (0, 1]$ using the inverse transform method [19], which gives

$$\ell = \left\lfloor \frac{\log r}{\log(1-p)} \right\rfloor \tag{5.2}$$

Now to generate the intra edges, i.e., ER random graph $G(n_i, \frac{d_i^2}{S})$, we apply the edge skipping technique on a sequence of the potential edges. To save memory space, we do not create any explicit sequence of the edges. Instead, the edges are represented by a set of consecutive integers $1, 2, \ldots, M$, where $M = \binom{|V_i|}{2} = \binom{n_i}{2}$, following a lexicographic order of the edges as shown in Figure 5.2(a) and 5.2(b). Now we select a subset of the integers from $1, 2, \ldots, M$ by applying the skipping technique with the probability $p = \frac{d_i^2}{S}$ as follows. Let

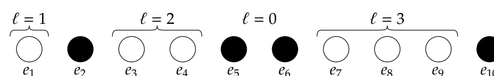
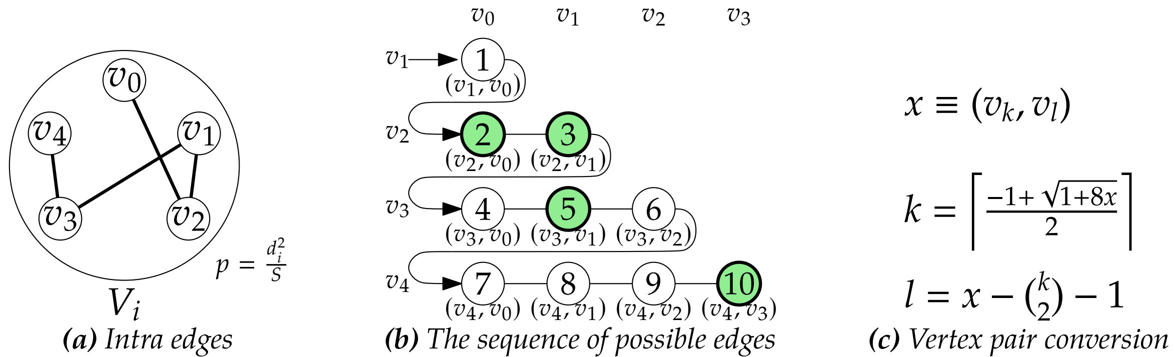


Figure 5.1: *Selecting edges from a sequence of potential edges. The black circles represents the selected edges.*

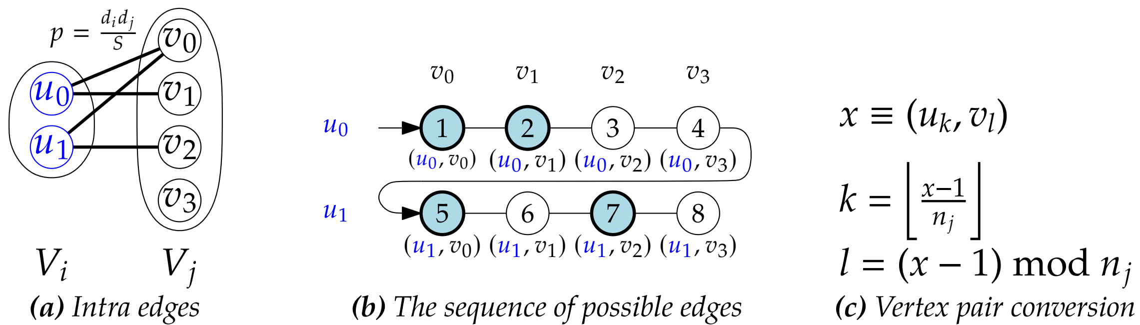
x be the last selected edge (initially $x = 0$). Then the skip length ℓ is computed using the Equation 5.2. The next selected edge is given by $x \leftarrow x + \ell + 1$. The selected edge number x is converted into an edge using the equations shown in Figure 5.2(c). This process is repeated until $x \geq M$.



(a) Intra edges *(b) The sequence of possible edges* *(c) Vertex pair conversion*

Figure 5.2: *Group V_i using $\mathcal{G}(n, p)$ model with $n_i = 5$ and $p = \frac{d_i^2}{S}$.*

Creating Inter Edges. For any $u \in V_i, v \in V_j$, the edge (u, v) is created with probability $p_{u,v} = \frac{d_i d_j}{S}$. Note that for all pairs of $u \in V_i, v \in V_j$, the probabilities $p_{u,v}$ are equal. Therefore, generating the inter edges between V_i and V_j is equivalent to generating a random bipartite graph [136] with n_i and n_j vertices and $p = \frac{d_i d_j}{S}$ edge probability (see Figure 5.3(a)).



(a) Intra edges *(b) The sequence of possible edges* *(c) Vertex pair conversion*

Figure 5.3: *Inter edges between V_i and V_j ($n_i = 4$ and $n_j = 2$).*

The edge skipping technique can also be applied to generate the inter edges using the random bipartite model (Figure 5.3(a)). In this case, the potential edges are represented by consecutive integers $1, 2, \ldots, M$, where $M = |V_i||V_j| = n_i n_j$ (Figure 5.3(b)). Next, the edge skipping technique is applied on this sequence with probability $p = \frac{d_i d_j}{S}$. The selected numbers x are converted to the edges using the equations shown in Figure 5.3(c).

Vertex Labels. Each vertex is identified by a unique integer label from 1 to n as follows.

Let λ_i be the label of the first vertex of a group V_i, where $\lambda_1 = 1$ and $\lambda_i = 1 + \sum_{j=1}^{i-1} n_j$ for $i > 1$. Then, the vertices in V_i are labeled by the integers from λ_i to $\lambda_{i+1} - 1$. Note that we only store the starting label for each group, which requires $O(\Lambda)$ memory.

Implementation. The pseudocode of our algorithm is presented in Algorithms 7 and 8. The procedure EDGE-SKIPPING() creates edges using the edge skipping technique. It takes two group indices i, j, probability p, and the $start$ and end of a sequence of potential edges as input. Using a random number $r \in (0, 1]$, the skip length ℓ is computed in line 6. The next selected edge x is computed in line 6 and converted to edge (u, v) in line 9 (intra edges) and line 11 (inter edges). The global labels of the endpoints u and v are denoted by $\lambda_i + u$ and $\lambda_j + v$, respectively.

Algorithm 7: Generating edges using edge skipping

```
1  Procedure EDGE-SKIPPING(i, j, p, start, end)
2      x ← start − 1
3      while x < end do
4          r ← a uniform random number in [0, 1)
5          ℓ ← ⌊ log r / log(1−p) ⌋
6          x ← x + ℓ + 1
7          if x ≤ end then
8              if i = j then
9                  u ← ⌈ (−1+√(1+8x)) / 2 ⌉;   v ← x − (u choose 2) − 1
10             else
11                 u ← ⌊ (x−1) / n_j ⌋;   v ← (x − 1) mod n_j
12             Output edge (λ_i + u, λ_j + v)
```

The procedure DG-CL() (Algorithm 8) generates edges for all pairs of groups using the procedure EDGE-SKIPPING(). Lines 2 to 4 compute the starting label of each group. The sum of expected degrees S is computed in line 5. Lines 6 and 7 iterate over all pairs of groups $\{\{V_i, V_j\} : 1 \leq i \leq j \leq \Lambda\}$. For any pair $\{V_i, V_j\}$, if $i = j$, intra edges for group V_i are created by calling the procedure EDGE-SKIPPING$(i, i, \frac{d_i^2}{S}, 1, \binom{n_i}{2})$ (line 9). Otherwise, inter edges are created between groups V_i and V_j by calling EDGE-SKIPPING$(i, j, \frac{d_i d_j}{S}, 1, n_i n_j)$ (line 11).

Space Complexity. For each group V_i, we only store d_i, n_i, and λ_i, leading to the space complexity of $O(\Lambda)$. In contrast, the MH algorithm requires storing the entire degree

Algorithm 8: The DG algorithm for the CL model

```
1  Procedure DG-CL(𝔻, {n_i}_{i∈𝔻})
2      λ_1 ← 1
3      for i = 2 to Λ do // Starting Labels
4          λ_i ← λ_{i−1} + n_i
5      S ← Σ_{i=1}^{Λ} (n_i × d_i)
6      for i = 1 to Λ do
7          for j = 1 to j do
8              if i = j then // Intra edges for V_i
9                  EDGE-SKIPPING(i, i, d_i^2/S, 1, C(n_i, 2))
10             else // Inter edges between V_i and V_j
11                 EDGE-SKIPPING(i, j, d_i d_j/S, 1, n_i n_j)
```

sequence W in the memory leading to the space complexity of $O(n)$.

Time Complexity. Computing the starting label of the groups takes $O(\Lambda)$ time. Computing the sum S also takes $O(\Lambda)$ time. The for loops in lines 6 and 7 iterate $O(\Lambda^2)$ times. Each iteration calls the procedure EDGE-SKIPPING(). The while loop in line 3 of procedure EDGE-SKIPPING() iterates once per generated edge. If m edges are generated, the while loop takes $O(m)$ time in total. Therefore, the algorithm takes $O(\Lambda + \Lambda^2 + m) = O(m + \Lambda^2)$ time.

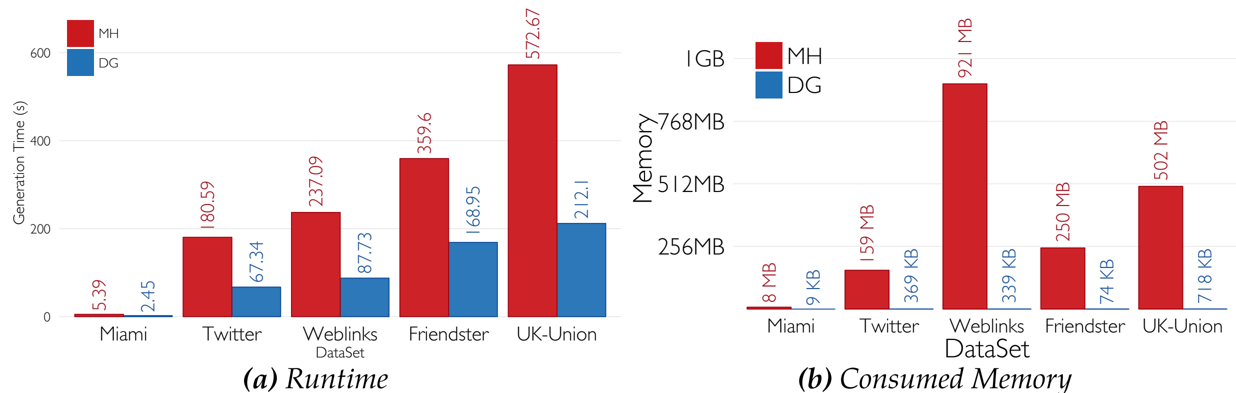
When $\Lambda^2 < O(m)$, our algorithm is asymptotically as good as the MH algorithm. In fact experimental results show that our algorithm about 3~4 times faster than the MH algorithm. Note that the CL model is only applicable when $w_{\max}^2 < S$, where $w_{\max}$ is the maximum expected degree. For integral expected degrees, we have $\Lambda < w_{\max}$, i.e., $\Lambda^2 < S = O(m)$. Therefore, whenever the CL model is applicable, our algorithm has a run time of $O(m)$. Clearly, we will use the CL model only when it is applicable. In fact, in most of the real-world graphs Λ^2 is significantly smaller than m as shown in Table 5.2. Even for power-law networks that have very skewed degree distributions and few vertices with very high degrees, the maximum degree is $O(\sqrt[\gamma]{n})$ where γ is the power-law exponent [6]. Typical values of γ is between 2 and 3.

Experimental Evaluation. Now we experimentally evaluate the performance of our algorithm against the MH algorithm [105], which is the best known sequential algorithm, using both real-world and synthetic networks. We extracted the degree sequences of these networks, and then generated new graphs from these degree sequences. Figure 5.4

Table 5.2: *Number of distinct degrees in real-world graphs*

Network	Type	n	m	Λ	$\frac{\Lambda^2}{m}$
Miami [16]	Contact	2M	51M	398	0.003
Weblinks	Real-world	276M	1B	14K	0.190
Twitter [86]	Real-world	41M	1B	20K	0.207
Friendstar [161]	Real-world	65M	2B	3K	0.005
UK-Union [22]	Real-world	131M	4B	30K	0.201
Erdős-Rényi (ER)	Synthetic	1M	200M	117	0.00007
Power-Law (PL)	Synthetic	1B	249B	10K	0.0004

demonstrates the run time and memory required by the algorithms. We observe that our DG algorithm is approximately 3 times faster than the MH algorithm as we discussed before. A huge improvement made by our DG algorithm is on the memory requirement, by a factor of 440-3474 for the networks shown in Figure 5.4. Thus, the DG algorithm is more efficient in both time and space requirements. Moreover, our DG algorithm leads to a better parallel algorithm, which is presented in the next section.



(a) *Runtime* ***(b)*** *Consumed Memory*

Figure 5.4: *Performance of the sequential algorithms*

We also compare the performance of our algorithm against the Graph500 generator (Version 2.1.4), one of the most used random graph generators [65] for benchmarking HPC systems. Generating a graph with $1B$ edges requires 650.46 seconds using the sequential Graph500 generator. In contrast, our algorithm takes only 67.34 seconds. Further, the Graph500 generator requires $\Theta(m)$ space whereas our algorithm requires $\Theta(\Lambda)$ space, where $\Lambda \ll m$. Moreover, to fit the degree distribution of an existing real-world network, the probability

matrix of the Graph500 generator has to be determined using a maximum likelihood based fitting algorithm (KronFit), which can take a significant amount of time, and often the fit is not perfect [93, 117]. Fitting the degree distribution of a real-world network (75K vertices, 508K edges) requires approximately 45 minutes in KronFit [93]. Our algorithm requires only the degree distribution that can be extracted in a few seconds.

5.3 Parallelization of the DG Algorithm

In this section, we present the parallelization of the DG algorithm. We assume that the input degree distribution of the graph is available for every processor. Let P be the number of processors. Efficient parallelization of Algorithm 8 requires:

- Computing the starting id (λ_i) of each group V_i
- Computing the sum S in parallel
- Generating the edges using the P processors with good load balancing

Each processor computes the starting id of each group in $O(\Lambda)$ time. The sum S is efficiently computed using a parallel sum operation in $O(\frac{\Lambda}{P} + \log P)$ time. We divide the work of generating edges into many independent tasks. Let $\mathcal{T}_{i,j}$ be the task of generating edges between groups V_i and V_j, where $d_i, d_j \in \mathbb{D}$. Note that all the tasks are mutually independent, i.e., for any $1 \leq i, i', j, j' \leq \Lambda$ such that $i \neq i'$ or $j \neq j'$, tasks $\mathcal{T}_{i,j}$ and $\mathcal{T}_{i',j'}$ can be executed independently by two different processors. Also, notice that when $i = j$, task $\mathcal{T}_{i,i}$ generates intra edges, otherwise $\mathcal{T}_{i,j}$ produces inter edges. There is a total of Λ and $\binom{\Lambda}{2}$ tasks for intra and inter edges, respectively. Let $\tau = \Lambda + \binom{\Lambda}{2} = \frac{\Lambda(\Lambda+1)}{2}$ be the number of tasks. In the next section, we describe how the τ tasks are executed by the P processors such that the loads are well balanced.

5.3.1 Task Distribution and Load Balancing

To distribute the tasks with good load balancing, we need an accurate estimation of the computational cost of each task. Estimating costs and distributing tasks to get the best load balancing are the most challenging parts of our parallel algorithm. For the best speedup, estimation and distribution must also be done in parallel that are non-trivial problems.

Computational Cost. Let $c_{i,j}$ be the computational cost of executing task $\mathcal{T}_{i,j}$. Assume that α unit of time is required to initialize a task and β unit of time to generate an edge. Let $m_{i,j}$

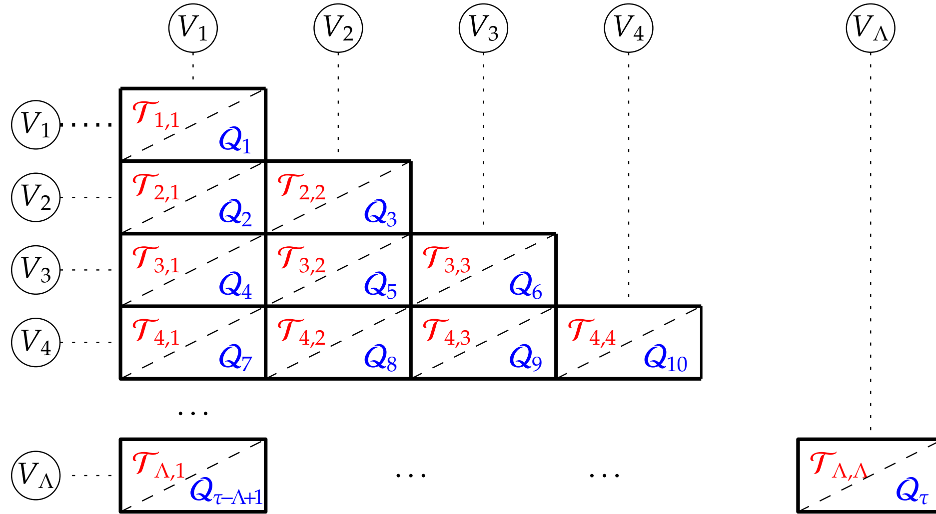
be the expected number of edges generated by task $\mathcal{T}_{i,j}$. Then the expected computation cost for $\mathcal{T}_{i,j}$ is

$$E[c_{i,j}] = \alpha + \beta m_{i,j} = \begin{cases} \alpha + \beta \frac{n_i(n_i-1)}{2} \frac{d_i^2}{S}, & i = j \\ \alpha + \beta n_i n_j \frac{d_i d_j}{S}, & i \neq j. \end{cases} \tag{5.3}$$

Therefore, the total expected computational cost C is given by

$$C = \sum_{1 \leq j \leq i \leq \Lambda} E[c_{i,j}] = \sum_{1 \leq j \leq i \leq \Lambda} (\alpha + \beta m_{i,j}) = \alpha\tau + \beta m \tag{5.4}$$

where m is the expected number of generated edges. For the optimal load balancing, the tasks need to be distributed in such a way that each processor has a computational cost of $\hat{C} = \frac{C}{P}$.



(a) All Tasks

$\mathcal{T}_{1,1}$	$\mathcal{T}_{2,1}$	$\mathcal{T}_{2,2}$	$\mathcal{T}_{3,1}$	$\mathcal{T}_{3,2}$	$\mathcal{T}_{3,3}$	...	$\mathcal{T}_{\Lambda,\Lambda}$
Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	...	Q_τ

(b) Ordering and Relabeling the Tasks

Figure 5.5: *Relabeling the tasks. The red and blue texts represent the original and new labels of the tasks respectively.*

Task Relabeling. So far, we used two indices i, j for a task $\mathcal{T}_{i,j}$. To simplify the discussion and implementation, we relabel the tasks from two indices to a single natural *task number*. Let Q_x be the new label of the task $\mathcal{T}_{i,j}$ where $x = \frac{i(i-1)}{2} + j$. Let Π be the set of tasks using

the new labels, i.e., $\Pi = \{Q_1, Q_2, Q_3, \ldots, Q_\tau\}$. The relabeled task Q_x can be converted to the original label $\mathcal{T}_{i,j}$ using the functions:

$$i = \left\lceil \frac{-1 + \sqrt{1 + 8x}}{2} \right\rceil \quad \text{and} \quad j = x - \frac{i(i-1)}{2}. \tag{5.5}$$

Task relabeling is depicted visually in Figure 5.5. Let c_x be the cost of the task Q_x, i.e., $c_x = c_{i,j}$ for the original task $\mathcal{T}_{i,j}$.

Task Distribution. To generate the edges in parallel, first, the set of tasks Π is divided into P disjoint subsets $\Pi_0, \Pi_1, \ldots, \Pi_{P-1}$; i.e., $\Pi_k \subset \Pi$, such that for any $k \neq l$, $\Pi_k \cap \Pi_l = \emptyset$ and $\bigcup_k \Pi_k = \Pi$. Each processor $\mathcal{P}_k$ is assigned the subset Π_k and executes the tasks $\{Q_x \in \Pi_k\}$. The computational cost of a processor $\mathcal{P}_k$ is given by $c(\mathcal{P}_k) = \sum_{Q_x \in \Pi_k} c_x$. We need to find the subsets Π_k such that each processor has almost equal cost, i.e., $c(\mathcal{P}_k) \approx \hat{C}$. Finding such subsets is a well-known problem called *chains-on-chains partitioning* (CCP) problem [102, 110, 116]. For better speedup the CCP problem has to be computed in parallel. In [7], the authors presented an efficient $O(\frac{\tau}{P} + P)$ time distributed memory parallel algorithm, called the *uniform cost partitioning* (UCP) for the CCP problem. Their algorithm uses cumulative cost $C_x = \sum_{i=1}^{x} c_x$ for each task Q_x for distributing tasks. Let any subset Π_k starts with the task Q_{q_k} and ends with the task $Q_{q_{k+1}-1}$ where q_k is called the *lower boundary* of Π_k. The lower boundary q_k satisfies the following condition: $C_{q_k - 1} < k\hat{C} \leq C_{q_k}$ for $0 < k \leq P - 1$. Then $q_k = \arg\min_x (C_x \geq i\hat{C})$, i.e., a task Q_x is executed by the processor $\mathcal{P}_k$ where $k = \left\lfloor \frac{C_x}{\hat{C}} \right\rfloor$. However, a task in the UCP algorithm is non-divisible, i.e., the entire task is assigned to a processor. If some boundary tasks are very large, it can lead to imbalanced loads. If we can break those tasks into arbitrary smaller subtasks, we can achieve a fine-grained granularity on load balancing. In fact, we can show a task Q_x can be divided into arbitrary smaller subtasks. As a result, we present an extension of the UCP algorithm, called UCP-DIV that achieves fine-grained load balancing by dividing the tasks.

Dividing Tasks. Using the edge-skipping technique, a task repeatedly computes the skip lengths with a probability p (using the geometric distribution) and generates edges from a sequence of potential edges (see Figure 5.6(a)). We can also divide the sequence into two arbitrary disjoint sub-sequences and apply the edge skipping technique on those sub-sequences individually (see Figure 5.6(b)), with the same result and effect. Both of these two processes are stochastically equivalent due to the memoryless property of the geometric random variable.

Consider two edges e' and e'' in Figure 5.6(a). In the original sequence, both e' and e'' are selected with probability p regardless of their distance from the beginning edge of the sequence. In other words, regardless where the sequence begins, every edge is selected

(a) *Original sequence* ***(b)*** *Two sub-sequences*

Figure 5.6: *Dividing a task into multiple sub-tasks*

with probability p. That is, we can arbitrarily break the sequence into two subsequences at any edge k and use that as the beginning of the second subsequence (Figure 5.6(b)). Any edge $e'' \geq k$ will still be selected with probability p if the edge skipping technique is applied on that subsequence. In fact, any sequence can be divided into any number of subsequences.

Uniform Cost Partitioning with Task Division. Now, we present the UCP-DIV algorithm. Similar to the original UCP algorithm, we also find the lower boundary x of a partition Π_k such that $C_{x-1} < k\hat{C} \leq C_x$. Instead of assigning the entire boundary task Q_x to processor $\mathcal{P}_k$ where $k = \left\lfloor \frac{C_x}{\hat{C}} \right\rfloor$ (as done in the UCP algorithm), we break it into two or more subtasks (see Figure 5.7). Let $Q_{x,s,t}$ be a subtask of the task Q_x with the subsequence starting from edge number s to t. Note that the subtask $Q_{x,1,M}$ represents the entire task Q_x where M is the number of potential edges in Q_x.

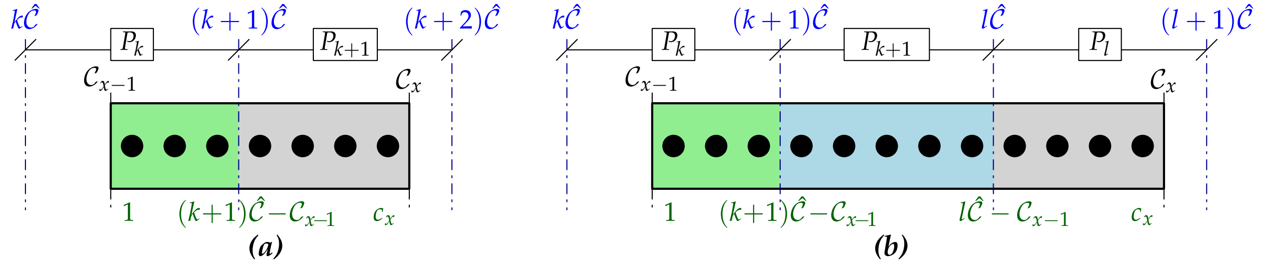


Figure 5.7: *Dividing the boundary tasks. The blue and green texts represent the cost boundaries among the processors and the subtask partitions within the task respectively.*

The first part of the boundary task Q_x is executed by $\mathcal{P}_k$ where $k = \left\lfloor \frac{C_{x-1}}{\hat{C}} \right\rfloor$. To make $c(\mathcal{P}_k) = \hat{C}$, we assign $(k+1)\hat{C} - C_{x-1}$ more loads to $\mathcal{P}_k$. Therefore, we divide the task Q_x into a subtask $Q_{x,1,t}$ such that $t = \left\lfloor \frac{(i+1)\hat{C}-C_{x-1}}{c_x} M \right\rfloor$ (see Figure 5.7).

Now, if the remaining part of the task $C_x - (k+1)\hat{C} \leq \hat{C}$ (see Figure 5.7(a)), $\mathcal{P}_{k+1}$ executes the last part of the task, i.e., the subtask $Q_{x,t+1,M}$ is assigned to $\mathcal{P}_{k+1}$. Otherwise, we divide the remaining part of the task again (see Figure 5.7(b)). Let $\mathcal{P}_l$ be the processor that executes the last part of the task Q_x where $l = \left\lfloor \frac{C_x}{\hat{C}} \right\rfloor$. Each processor from $\mathcal{P}_{k+1}$ to $\mathcal{P}_{l-1}$ executes $\hat{C}$ amounts of loads, i.e., $\mathcal{P}_z$ is assigned with the subtask Q_{x,s_z,t_z} where $s_z = \left\lfloor \frac{z\hat{C}-C_{x-1}}{c_x} M \right\rfloor + 1$

and $t_z = \left\lfloor \frac{(z+1)\hat{C} - C_{x-1}}{c_x} M \right\rfloor$ for $z = (k+1), \ldots (l-1)$ (see Figure 5.7(b)). The last part of the task, i.e., the subtask $Q_{x,s_l,M}$ is assigned to the processor $\mathcal{P}_l$ where $s_l = \left\lfloor \frac{C_x - l\hat{C}}{c_x} M \right\rfloor + 1$. Thus, the algorithm leads to optimal load balancing.

5.3.2 Parallel Implementation

Now, we present the implementation details of the parallel DG algorithm for the CL model. First, we show the parallel algorithm for the UCP-DIV method for task distribution. Then we present the parallel DG algorithm for the CL model.

Parallel UCP-DIV Algorithm. The pseudocode of the parallel UCP-DIV algorithm is presented in Algorithm 9. The procedure `UCP-Div()` computes the task boundaries using the procedure `Find-Boundaries()`. The algorithm assumes that the cumulative costs C_x and average cost per processor $\hat{C}$ are already computed (discussed in the next section).

Each processor $\mathcal{P}_k$ executes the procedure `UCP-Div()` in-parallel. $\mathcal{P}_k$ is responsible for finding the boundary tasks among $\frac{\tau}{P}$ tasks from $\frac{k\tau}{P} + 1$ to $\frac{(k+1)\tau}{P}$. A boundary task is a task Q_x where $\lfloor \frac{C_{x-1}}{\hat{C}} \rfloor \neq \lfloor \frac{C_x}{\hat{C}} \rfloor$. The boundary tasks are found using the procedure `Find-Boundaries()` (line 3). The procedure `Find-Boundaries()` takes parameters b, e and finds all the boundaries among the tasks from Q_b to Q_e using a recursive divide and conquer based algorithm. Each boundary task Q_x is divided into subtasks and assigned to processors as discussed earlier. Let $Q_{x,s,t}$ be such a subtask assigned to processor $\mathcal{P}_z$. A message $\langle \texttt{type}, x, s, t \rangle$ representing the subtask $Q_{x,s,t}$ is sent to $\mathcal{P}_z$ (from $\mathcal{P}_k$), where $\texttt{type}, x, s, t$ represents the type of subtask (either `start` or `end`), the task number, the starting edge, and the ending edge of the task, respectively. Note that each processor $\mathcal{P}_z$ receives two such messages. The pair of the messages is returned as output (Line 6). The run time of the algorithm is $O\left(\frac{\tau}{P} + P\right)$ in the worst case as shown in Theorem 14.

Theorem 14. *The parallel UCP-DIV algorithm to distribute τ tasks into P processors runs in $O\left(\frac{\tau}{P} + P\right)$ time.*

Proof. It is easy to see that for each processor $\mathcal{P}_k$, the run time of the algorithm is dictated by the number of task boundaries found in the range $\left[\frac{k\tau}{P} + 1, \frac{(k+1)\tau}{P}\right]$. Finding a boundary on these $\frac{\tau}{P}$ tasks require $O\left(\log \frac{\tau}{P}\right)$ time. If the range has η boundaries, then it takes $O\left(\min\left\{\frac{\tau}{P}, \eta \log \frac{\tau}{P}\right\}\right)$ time. For each subtask, exactly two messages are sent to corresponding processors. There are at most P boundaries in $\left[\frac{i\tau}{P} + 1, \frac{(i+1)\tau}{P}\right]$. Thus, in the worst case, a processor may need to send at most $2P$ messages taking $O(P)$ time. Therefore, the total time in the worst case is $O\left(\min\left\{\frac{\tau}{P}, \eta \log \frac{\tau}{P}\right\} + P\right) = O\left(\frac{\tau}{P} + P\right)$. □

Algorithm 9: Finding task boundaries using UCP-DIV

```
1  Procedure UCP-Div()
2      k ← Processor Id
       /* Executed by processor P_k in parallel                */
3      Find-Boundaries(kτ/P + 1, (k+1)τ/P)
4      A ← Receive Message ⟨start, q_k, s, t⟩
5      B ← Receive Message ⟨end, q_{k+1} − 1, s′, t′⟩
6      return ⟨A, B⟩

7  Procedure Find-Boundaries(b, e)
8      if b > e then // No boundary
9          return
10     x ← (b+e+1)/2
11     M ← # of potential edges in Q_x
12     l ← C_x / Ĉ
13     k ← C_{x−1} / Ĉ
14     if k ≠ l then
15         t ← ⌊((k+1)Ĉ − C_{x−1}) / c_x · M⌋
16         Send Message ⟨start, x, 1, t⟩ to P_k
17         for z ← k + 1 to l − 1 do
18             s ← ⌊(zĈ − C_{x−1}) / c_x · M⌋ + 1
19             t ← ⌊((z+1)Ĉ − C_{x−1}) / c_x · M⌋
20             Send Message ⟨start, x, s, t⟩ to P_z
21             Send Message ⟨end, x, s, t⟩ to P_z
22         s ← ⌊(lĈ − C_{x−1}) / c_x · M⌋ + 1
23         Send Message ⟨end, x, s, M⟩ to P_l
24     Find-Boundaries(b, x − 1)
25     Find-Boundaries(x + 1, e)
```

Parallel DG Algorithm. The pseudocode of the parallel DG algorithm using the UCP-DIV algorithm is presented in Algorithm 10. The procedure PARALLEL-DG-CL() is executed by each processor $\mathcal{P}_k$. It takes a degree distribution as input and generates the edges in parallel. $\mathcal{P}_k$ computes the sum (line 3), starting labels of each group (lines 4-6), and the total cost and average computational costs (line 7-11) in parallel. Next, the procedure UCP-Div() (Line 12) is called which return the pair of messages $\langle A, B \rangle$. Recall that A and B represents

Algorithm 10: Parallel DG Algorithm for the CL Model

```
1  Procedure PARALLEL-DG-CL(𝔻, {n_k}_{k∈𝔻})
2      k ← Processor Id
       /* Executed by processor 𝒫_k in parallel                */
3      In-Parallel: Compute S = ∑_{l=1}^{Λ} n_l d_l
4      λ_1 = 1
5      for l ← 2 to Λ do
6          λ_l ← λ_{l−1} + n_l
7      for x ← kτ/P + 1 to (k+1)τ/P do
8          C_x ← C_{x−1} + c_x
9      z_k ← C_{(k+1)τ/P}
10     In Parallel: C ← ∑_{l=0}^{P−1} z_l
11     Ĉ ← C/P
12     ⟨A, B⟩ ← UCP-DIV()
13     for Q_{x,s,t} = A to B do
14         i = ⌈(−1+√(1+8x))/2⌉
15         j = x − i(i−1)/2
16         EDGE-SKIPPING(i, j, d_i d_j / S, s, t)
```

two subtasks $Q_{x,s,t}$ and $Q_{x',s',t'}$ assigned to $\mathcal{P}_k$. $\mathcal{P}_k$ executes the subtasks, and all the tasks from Q_{x+1} to $Q_{x'}$ using the procedure EDGE-SKIPPING() from Algorithm 7 (line 13).

The run time of the parallel Algorithm 10 is $O\left(\frac{m+\Lambda^2}{P} + \Lambda + P\right)$ w.h.p. as shown in Theorem 16. With $\Lambda^2 = O(m)$, the run time of the algorithm is $O\left(\frac{m}{P} + \Lambda + P\right)$. To prove Theorem 16, we need a bound on computational cost shown in Theorem 15.

Theorem 15. *The computational cost in each processor is* $O\left(\frac{m+\tau}{P}\right)$ *w.h.p.*

Proof. Let x be the number of potential edges processed in processor $\mathcal{P}_k$, and these are denoted by $g_1, g_2, \ldots, g_x$ (in any arbitrary order). Let X_i be an indicator random variable such that $X_i = 1$ if $\mathcal{P}_k$ creates g_i and $X_i = 0$ otherwise. Then the number of edges created by $\mathcal{P}_k$ is $X = \sum_{i=1}^{x} X_i$. As discussed in Section 5.2.2, generating the edges efficiently by grouping and applying the edge skipping technique is stochastically equivalent to generating each edge (u, v) independently with probability $p_{u,v} = \frac{w_u w_v}{S}$. Let μ_k be the expected number of edges generated by $\mathcal{P}_k$, i.e., $\mu = E[X] = m_k$. Using the standard Chernoff bound for independent indicator random variables for any $0 < \delta < 1$ with $\delta = \frac{1}{2}$

we have:

$$\Pr\left[X \geq (1+\delta)\mu\right] \leq e^{-\delta^2\frac{\mu}{3}}$$
$$\Pr\left[X \geq \frac{3}{2}m_k\right] \leq e^{-\frac{m_k}{12}} \leq \frac{1}{m_k^3}$$

for any $m_k \geq 189$. We assume $m \gg P$ and consequently $m_k > P$ for all k. Now using the union bound,

$$\Pr\left[X \geq \frac{3}{2}m_k\right] \leq m_k \frac{1}{m_k^3} = \frac{1}{m_k^2}$$

for all k simultaneously. Then with probability at least $1 - \frac{1}{m_k^2}$, the computation cost $\beta X + \alpha|\Pi_k|$ is bounded by $\frac{3}{2}\beta m_k + \frac{3}{2}\alpha|\Pi_k| = \frac{3}{2}\left(\beta m_k + \alpha|\Pi_k|\right)$, where α, β are constants. By construction of the partitions by our algorithm, we have $(\beta m_k + \alpha|\Pi_k|) = \left(\frac{m+\tau}{P}\right)$. Therefore, the computational cost in all processors is $O\left(\frac{m+\tau}{P}\right)$ w.h.p. □

Theorem 16. *Our parallel algorithm using the UCP-DIV scheme for generating random graphs with the CL model runs in* $O\left(\frac{m+\Lambda^2}{P} + \Lambda + P\right)$ *time w.h.p.*

Proof. Computing the sum S in parallel takes $O\left(\frac{\Lambda}{P} + \log P\right)$ time. Computing the starting labels of each group requires $O(\Lambda)$ time. Computing the costs require $O\left(\frac{\tau}{P} + \log P\right)$ time. Using the UCP-DIV algorithm, task distribution takes $O\left(\frac{\tau}{P} + P\right)$ time (Theorem 14). In the UCP-DIV scheme, each partition has $O\left(\frac{m+\tau}{P}\right)$ computation cost w.h.p. (Theorem 15). Thus creating edges using procedure EDGE-SKIPPING() requires $O\left(\frac{m+\tau}{P}\right)$ time, and the total time is $O\left(\frac{m+\tau}{P} + P + \Lambda\right) = O\left(\frac{m+\Lambda^2}{P} + P + \Lambda\right)$ w.h.p. □

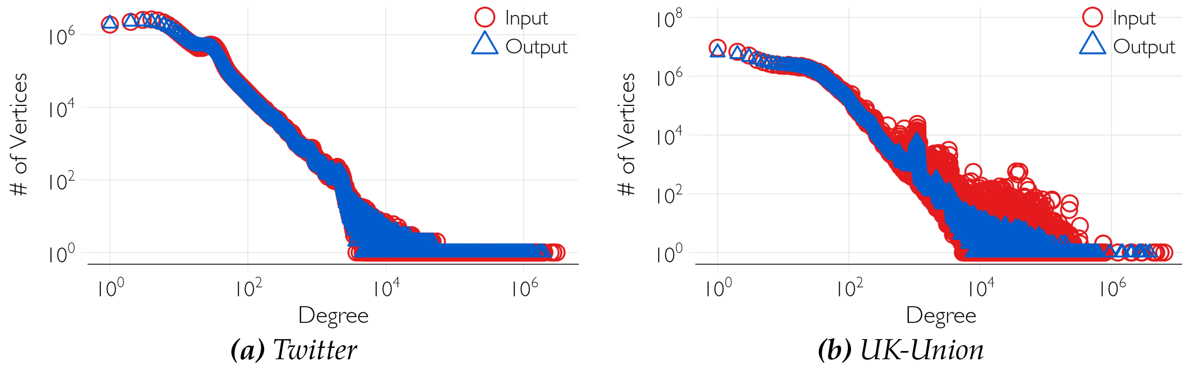
5.3.3 Experimental Results

Now, we experimentally evaluate the accuracy and performance of our algorithm. We show that the graphs generated by our algorithm closely match the input degree distribution. We also present the scalability and load balancing capabilities.

Experimental Setup. We used an 81-node HPC cluster for the experiments. Each node has two octa-core SandyBridge E5-2670 2.60GHz (3.3GHz Turbo) processors with 64GB RAM. We used MPICH2 (v1.7) for the algorithm. The run time does not include the disk I/O time to write the graph.

Degree Distribution. Figure 5.8 shows the input and generated degree distributions of Twitter and UK-Union graphs (see Table 5.2). As observed from the figures, the generated

degree distributions closely follow the input, which visually validates the correctness of our parallel algorithm. Some variations in the distribution are due to the randomness. We also experimented with several other graphs and observed the same result.



(a) *Twitter* **(b)** *UK-Union*

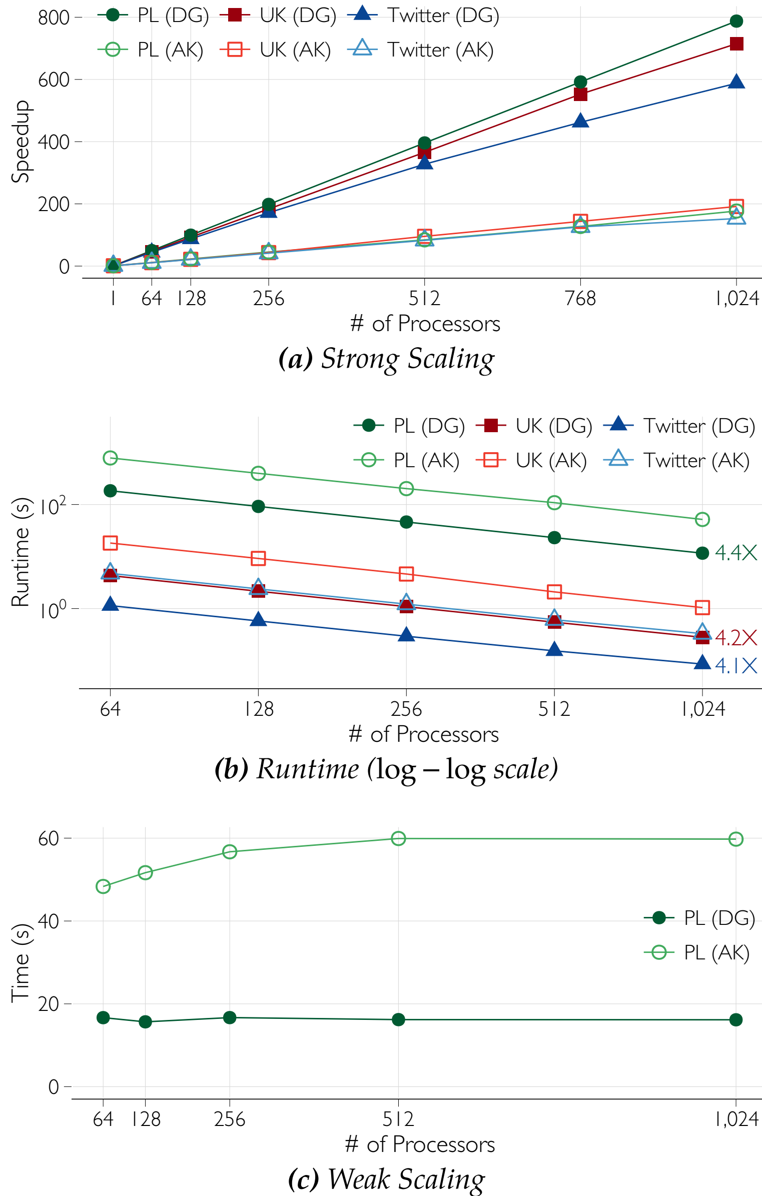
Figure 5.8: *Degree distributions of input and generated graph*

As a formal test, we use the Kullback-Leibler (KL) divergence [82] to compute the statistical difference between the input and output degree distributions. The KL divergence measures the difference between two probability distributions Q (input) and R (output) as information gain defined as:

$$D_{\mathrm{KL}}(Q\|R) = \sum_i Q(i) \log \frac{Q(i)}{R(i)}. \tag{5.6}$$

In other words, it measures the amount of information lost (in number of bits) when the output distribution R is used in place of the input distribution Q. The average minimum number of bits needed for each entry of input distribution Q are 6.12777 and 6.69488 for Twitter and UK-Union networks respectively. The KL divergences between the input and output degree distributions of our parallel algorithm for Twitter and UK-Union networks are 0.00693 and 0.01607, respectively. That accounts for a difference of number of bits required in percentage of 0.11% (Twitter) and 0.24% (UK-Union), which are negligible and expected due to the randomness of the network model. In fact, the original sequential algorithm for the Chung-Lu model also produces outputs with very similar KL divergences (0.00700 (0.11%) for Twitter and 0.01604 (0.24%) for UK-Union).

Strong and Weak Scaling. Strong scaling of a parallel algorithm shows its performance with the increasing number of processors while keeping the problem size fixed. Figure 5.9(a) shows the speedup of our parallel algorithm along with the best known parallel algorithm [7] (referred as the AK algorithm) for a massive synthetic (PL) and two large real-world graphs (Twitter and UK-Union). Speedups are measured as $\frac{T_s}{T_p}$, where T_s and T_p are the



(a) *Strong Scaling*

(b) *Runtime* (log – log *scale)*

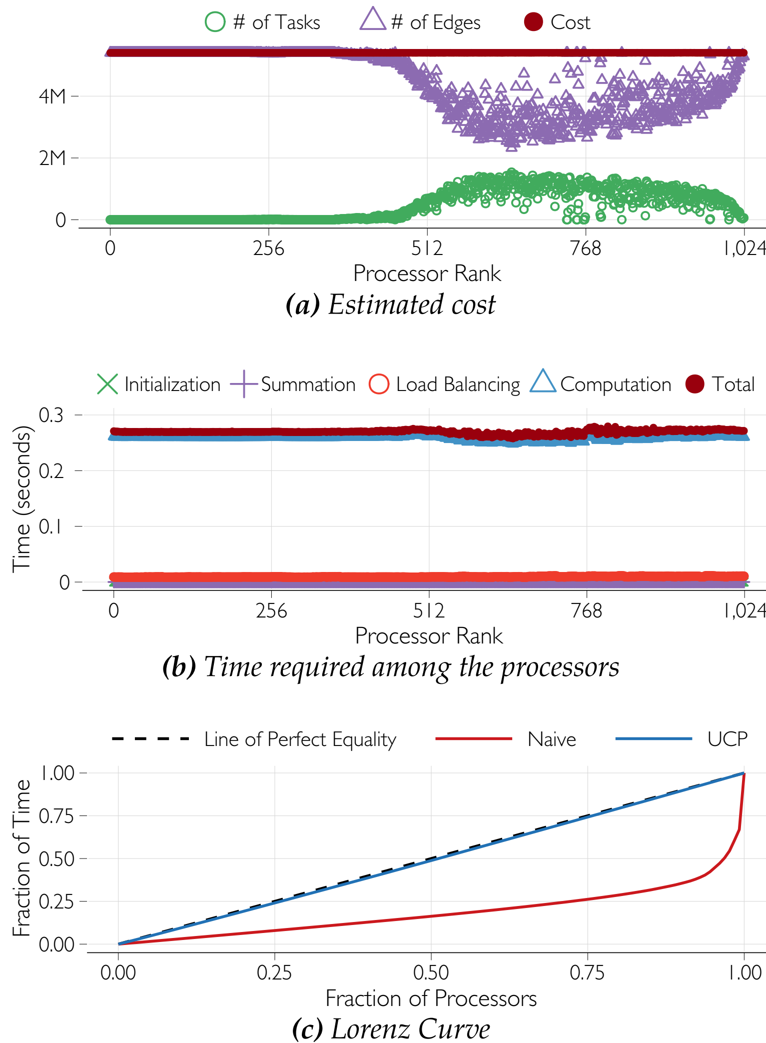
(c) *Weak Scaling*

Figure 5.9: *Strong and weak scaling of the parallel algorithms*

running time of the sequential and the parallel algorithm, respectively. The number of processors is varied from 1 to 1024. As shown in Figure 5.9(a), our algorithm achieves almost linear speedup for each graph. The AK algorithm also has a linear speedup. But our algorithm is approximately four times faster than the AK algorithm (see Figure 5.9(b)). Moreover, our algorithm requires less memory ($O(\Lambda)$ memory) than the AK algorithm ($O(n)$ memory). For example, for the Twitter, UK-Union, and PL graphs, the DG algorithm takes about 440, 716, and 16000 times less memory than the AK algorithm, respectively.

Thus, our algorithm scales to a large number of processors.

The weak scaling measures the performance of a parallel algorithm when the input size per processor remains constant. For this experiment, we varied the number of processors from 64 to 1024. For P processors, a PL graph with 10^6P vertices and 10^8P edges is generated. Note that weak scaling can only be performed on artificial graphs. Figure 5.9(c) shows that our algorithm also achieves very good weak scaling compared to the AK algorithm with almost constant run time.



***(a)** Estimated cost*

***(b)** Time required among the processors*

***(c)** Lorenz Curve*

***Figure 5.10:** Load balancing of the parallel DG algorithm*

Load Balancing. Our parallel algorithms provide very good load balancing. Figure 5.10 demonstrates the quality of our load balancing approaches described in Section 5.3.1. We formally quantify the quality of load balancing using Lorenz curves and Gini coefficients

as described below. *Lorenz curves* [99], often used in economics for representing inequality of the distribution of wealth, visualizes the load disparities [118]. In a Lorenz curve, the cumulative proportion of a distribution is plotted against the cumulative proportion of ordered individuals. In our context, the ordered individuals are the processors ordered by their computational time. Let the computational times of the P processors are denoted and ordered by y_i, where $1 \leq i \leq P$ and $y_i \leq y_{i+1}$. Then the Lorenz curve is the continuous piecewise linear function joining the points $\left(\frac{x}{P}, \frac{\mathcal{Y}_x}{\mathcal{Y}_P}\right)$, where $x = 0, 1, 2, \ldots, P$, $\mathcal{Y}_0 = 0$, and $\mathcal{Y}_x = \sum_{i=1}^{x} y_i$. If all processors require the same amount of time, the Lorenz curve is a straight line called *the line of perfect equality*. For imbalanced loads, the curve falls below that line. The *Gini coefficient* $G \in [0, 1]$ [62] is defined as:

$$G = \left(\frac{2\sum_{i=1}^{P} i y_i}{P\sum_{i=1}^{P} y_i} - \frac{P+1}{P}\right). \tag{5.7}$$

For balanced loads, the Gini coefficient is closer to 0, whereas, with increasing imbalanced load, it gradually reaches to 1.

Balancing the workloads for the real-world graphs are more challenging than the synthetic graphs. Therefore, in this experiment we demonstrate for the UK-Union graph, the largest public real-world graph [22] using 1024 processors. We experimentally determined $\alpha = 2$ and $\beta = 1$ for the cost function (see Equation 5.4), which achieves the best load-balancing. Figure 5.10(a) shows the number of tasks, edges, and the estimated cost of each processor. Note that the estimated cost is almost the same in each processor. Figure 5.10(b) demonstrates the time required by each processor for initialization, summation, load-balancing and graph computation steps. Figure 5.10(c) also shows the Lorenz curve based on the total time required by each processor. Our algorithm using the UCP-DIV task distribution scheme has a Gini coefficient of 0.015 indicating near perfect load balancing. In contrast, using a naïve scheme where each processor executes equal number of tasks, has a Gini coefficient of 0.63. The results strongly favor our choice of the cost function and the UCP-DIV task distribution algorithm. Thus, our algorithm achieves very good load-balancing, where each processor spends almost an equal amount of time.

5.4 Block Two-level Erdős-Rényi

The Block Two-Level Erdős-Rényi (BTER) is another model to generate random graphs using two fundamental properties: degree distribution and clustering coefficients [80, 134]. The clustering coefficient C_u of a vertex u is defined as the ratio of the number of edges among its neighbors to the maximum number of all possible such edges. More formally,

$C_u = \frac{|\{(v,w) \in E : v,w \in \mathcal{N}_u\}|}{\binom{\delta_u}{2}}$, where $\mathcal{N}_u$ is the set of neighbors, and δ_u is the degree of u. The BTER model takes as input (1) the desired degree distribution $\{n_d\}_{d \in \mathbb{D}}$, and (2) the desired average clustering coefficients by degree $\{c_d\}_{d \in \mathbb{D}}$ where $c_d = \frac{1}{n_d} \sum_{\{u:\delta_u = d\}} C_u$. First, the vertices are divided into many blocks called *affinity blocks* based on their expected degrees. An affinity block with degree d contains $d + 1$ vertices (except the last block). Typically, there are many small blocks with a low degree and a few large blocks with high degree vertices. Next, the edges are generated in two phases. In Phase 1, edges within each block are generated. Phase 1 generates triangle rich non-overlapping communities. Each block is represented by an ER model with probability p. For a block involving degree-d vertices, the probability is given as $p = \sqrt[3]{c_d}$. In Phase 2, edges across the blocks are created. Consider some vertex i with expected degree δ_i. Suppose, in Phase 1, the vertex created δ'_i edges. Then, $w_i = \delta_i - \delta'_i$ denotes the excess degree of the vertex i. To get the desired degree δ_i of a vertex i, w_i more edges need to be incident on i. The Chung-Lu (CL) model is applied to the expected degree sequence $\{w_i\}_{1 \le i \le n}$ to get the desired degree distribution.

A complete, scalable implementation of BTER model is given in [80]. We analyzed the BTER implementation and observed that both Phase 1 and Phase 2 can be improved significantly incorporating the DG method. To generate edges in both Phase 1 and Phase 2, the implementation uses sampling based edge generation similar to the approach proposed in [117]. In the sampling based algorithm, each edge is generated by randomly choosing two end-points based on their degrees. Therefore, there is a possibility that an edge is selected multiple times. Consider an affinity block b with n_b vertices and edge probability p_b in Phase 1. The expected number of edges generated by the ER model for the block b is $m_b = p_b\binom{n_b}{2}$. To ensure that m_b distinct edges are generated, their algorithm samples $w_b = \binom{n_b}{2} \ln \frac{1}{1-p_b}$ edges [80]. Note that $w_b \ge m_b$, because $\ln \frac{1}{1-p_b} \ge p_b$ for $0 \le p_b \le 1$. When the edge probability is relatively high w_b can be several factor larger than m_b. For example, during the generation of UK-Union graph, Phase 1 produces approximately 3.2B distinct edges. To generate those many edges, the original BTER implementation generates 8.9B edges, which is about 2.8 times more than the required number of edges (about 64% edges are discarded). Moreover, as the number of duplicate edges is very large, removing duplicates edges becomes very costly. In contrast, the edge skipping technique requires no unnecessary operation. All the edges generated in Phase 1 are distinct. Therefore, duplicate removal step is not required. Similar arguments can also be made for Phase 2. Moreover, as we saw earlier, edge skipping technique is highly scalable due to the ability to break a large task into multiple subtasks.

5.4.1 A DG-based Algorithm for the BTER Model

In this section, we present the DG-based BTER algorithm by incorporating the DG method.

Sequential Algorithm. The sequential algorithm runs in three steps: a) Initialization, b) Phase 1, and c) Phase 2. In the initialization step, the affinity blocks are created using the procedure `BTER_Setup()` (Algorithm 1 in [80]). In Phase 1, edges are produced using the edge skipping technique using only the ER model. Next, in Phase 2, the DG algorithm for the CL model is applied. Here, the vertices are grouped based on their excess degrees as defined earlier. Note that the expected excess degrees can contain fractions. After grouping, the rest of the algorithm is similar to Algorithm 8.

Parallel Algorithm. We can parallelize the Phase 1 and Phase 2 of our algorithm using the parallel framework used in Section 5.3. The task partitioning and load balancing are done independently in each phase with different cost functions. Phase 1 consists of a number of affinity blocks each executing the ER model independently. Therefore, each affinity block represents an independent task. To compute the computational cost of the task, we assign α_1 unit of time for processing an affinity block and β_1 unit of time for generating an edge. We apply the UCP-DIV algorithm to distribute the tasks into P processors. Parallelization of Phase 2 is quite similar to the parallel DG algorithm for the CL model. In this case, we assume α_2 unit of time for processing a task and β_2 unit of time processing an edge. The suitable values of $\alpha_1, \alpha_2, \beta_1, \beta_2$ are determined experimentally.

5.4.2 Experimental Evaluation

In this section, we evaluate the accuracy and performance of both our sequential and parallel algorithms with the original BTER implementation [80]. For fairness, we used the same set of graphs used in the original paper listed in Table 5.3. Henceforth, we refer BTER as the original implementation [80], and DG-BTER as our algorithm.

Performance of the Sequential Algorithm. The run times of the BTER and the sequential DG-BTER algorithms are also shown in Table 5.3. Runtimes for the LJournal and Hollywood graphs are collected from the MATLAB-based implementation of BTER. Twitter and UK-Union graphs are generated by scalable BTER on a Hadoop cluster with 32 computing nodes [80]. Our sequential DG-BTER algorithm not only outperforms the MATLAB-based BTER, it also outperforms the Hadoop-based scalable BTER implementation.

Degree Distribution and Average Clustering Coefficients. We demonstrate the input and generated degree distributions and average clustering coefficients for four real-world

Table 5.3: *Performance of the sequential algorithm for BTER*

			Runtime (s)	
Graph	Vertices	Edges	BTER [80]	DG-BTER
LJournal [22]	5M	49M	160.21	1.99
Hollywood [22]	2M	115M	450.79	5.34
Twitter [22, 86]	41M	1.2B	230.00	48.38
UK-Union [22]	131M	4.6B	1350.00	209.74

graphs LJournal, Hollywood, Twitter, and UK-Union in Figure 5.11. The graphs are generated using our parallel DG-BTER algorithm. As observed from the plots, both the generated degree distributions and average clustering coefficients closely follow the input. Our experiments with other networks also give similar results. The corresponding plots in the BTER paper [80] are exactly the same as ours, verifying the correctness of DG-BTER.

Load Balancing. Figure 5.12 demonstrates the load balancing performance of our parallel BTER algorithm for the UK-Union graph. We experimentally determined the parameters of the cost functions where $\alpha_1 = 2, \alpha_2 = 1.5, \beta_1 = \beta_2 = 1$. Figure 5.12(a) shows that the costs are distributed uniformly among the processors for both phases. Figure 5.12(b) demonstrates that each processor takes almost the same amount of time in every step of the algorithms, i.e., loads are well balanced.

Strong Scaling. Figure 5.13 shows the speedups of our parallel BTER algorithm. As Figure 5.13 demonstrates, we also achieve a linear speedup for a large number of processors. For example using 1024 processors, our algorithm took only 0.37 seconds for the UK-Union graph, with a speed-up of about 572 (in contrast to the 1350 seconds required by the BTER algorithm). Also, note that the speed up increases with graph size. Therefore, our algorithm is suitable for fast generation of massive graphs, significantly faster than the existing algorithms.

5.5 Stochastic Block Model

The stochastic block model (SBM) is another popular model first studied in mathematical sociology [73, 152]. A stochastic block model is defined with the following three parameters: *1)* a set of n vertices, *2)* k disjoint subset of vertices $\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_k$, called communities, and

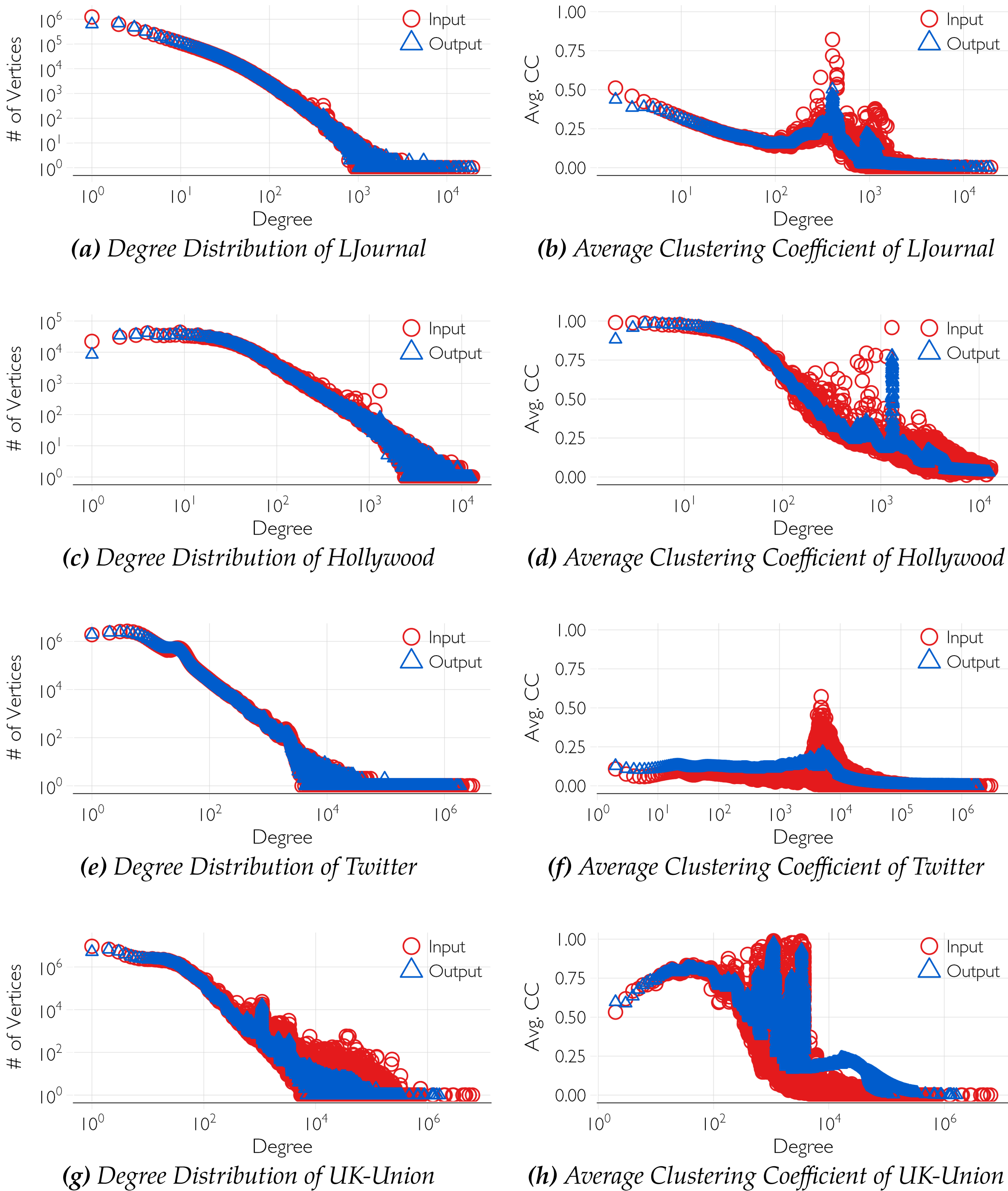
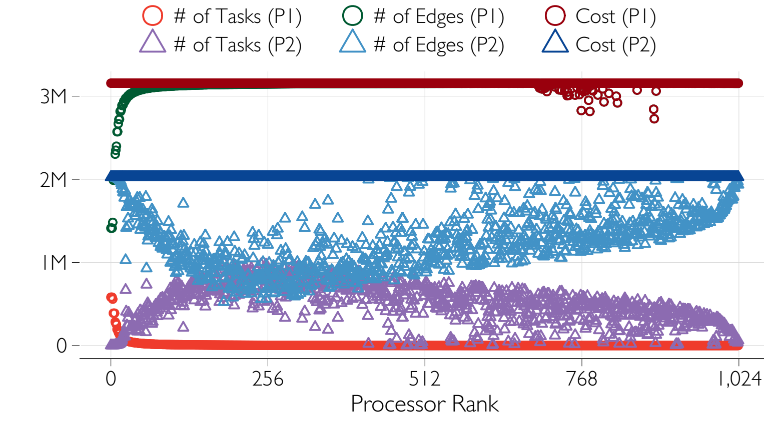
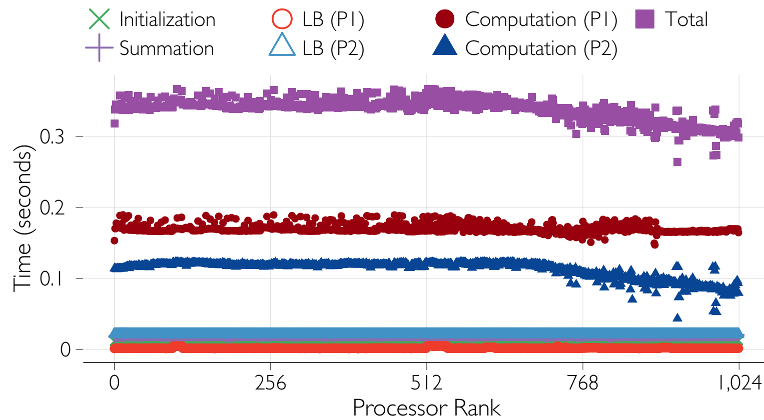


Figure 5.11: *Degree distributions and average clustering coefficient per degree of input and generated networks using the BTER model*



(a) *Estimated Cost*



(b) *Time (per processor)*

Figure 5.12: *Load Balancing of Parallel BTER. P1 and P2 denotes Phase 1 and Phase 2 respectively.*

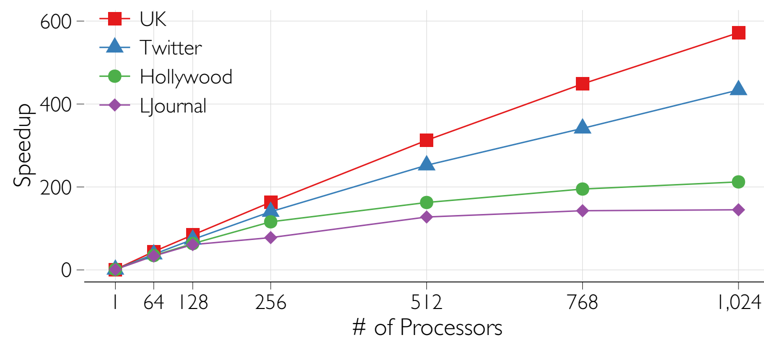


Figure 5.13: *Strong scaling of the parallel algorithms for BTER*

3) a $k \times k$ matrix M, where $M_{i,j}$ denotes the probability that a vertex of community $\mathbb{C}_i$ is connected to a vertex of community $\mathbb{C}_j$. With a given set of parameters n, $\{\mathbb{C}_i\}_{1 \le i \le k}$,

and M the edges are created as follows: any two vertices $u \in \mathbb{C}_i, v \in \mathbb{C}_j$ is connected with probability $M_{i,j}$. Note that for any pair of communities $\{\mathbb{C}_i, \mathbb{C}_j\}$, any possible edge $(u, v) : u \in \mathbb{C}_i, v \in \mathbb{C}_j$ is created with probability $p_{u,v} = M_{i,j}$, i.e., all potential edge in a pair of communities are independent and identically distributed. Observe that the groups in the DG algorithm are remarkably similar to the SBM communities. The main difference is that in the SBM model the probability p of an edge between two communities is provided in M, whereas in the DG algorithm, the probability depends on the degree of the groups. As a result, we can use the Algorithm 8 for generating edges efficiently using the SBM model by replacing the lines 9 and 11 with $\textsc{Edge-Skipping}(i, i, M_{i,i}, 1, \binom{|\mathbb{C}_i|}{2})$ and $\textsc{Edge-Skipping}(i, j, M_{i,j}, 1, |\mathbb{C}_i||\mathbb{C}_j|)$ respectively. Note that the parallel algorithm for the CL model can be applied to the SBM model in a similar fashion.

Performance of the Parallel Algorithm. Figure 5.14 shows the speedup of our parallel algorithm for generating edges using the SBM model for two graphs with 300 and 1000 communities with 2.4B and 17.8B edges, respectively. We can see that the strong scaling of the algorithm is also linear. Therefore, our algorithm is efficient and scalable to a large number of processors. Our algorithm also achieves very good load balancing.

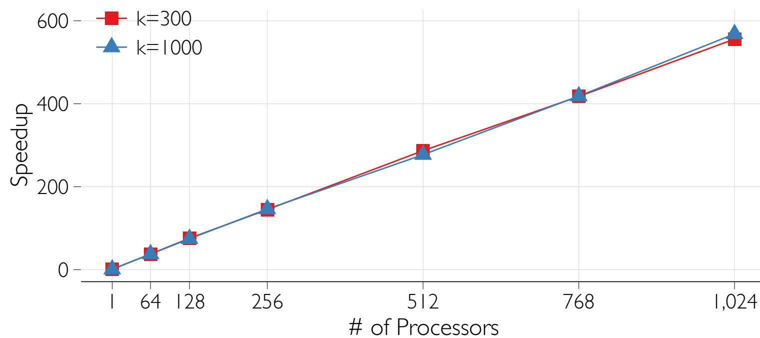


Figure 5.14: *Strong scaling of the parallel algorithm for the SBM*

5.6 Conclusion

Our DG method leads to novel algorithms with significantly improved space and time efficiency for generating random graphs using the CL, BTER, and SBM models, compared to the state-of-the-art algorithms for these models. Our algorithms are exact, in the sense that they generate graphs with the precise probability distribution, and improve on all prior algorithms with respect to rigorous theoretical guarantees, as well as their experimental performances. Further, the DG method leads to better parallel algorithms with optimal load balancing. The parallel algorithms scale very well to a large number of processors

and allows us to generate very-large scale random graphs. Extending our DG method to other random graph models with additional constraints is an interesting open direction.

Acknowledgment

This work has been partially supported by NSF DIBBs Grant ACI-1443054, DTRA Grant HDTRA1-11-1-0016, DTRA CNIMS Contract HDTRA1-11-D-0016-0001, and NSF NetSE Grant CNS-1011769.

Part II

GPU-Based Algorithms for Network Analysis and Data Mining Problems

Chapter 6

GPU-based Efficient Index Searching

6.1 Introduction

6.1.1 In-Memory Databases

In the age of Big Data, modern data systems need to support high throughput, low latency operations. Of late, main memory capacity is adequate enough to store the entire contents of some databases in the computer's main random access memory (RAM). In line with Jim Gray's observation, "Memory is the new disk, disk is the new tape", RAM has become sufficiently inexpensive to make in-memory databases a viable choice for large data-intensive enterprise applications [122]. Responding to these trends, commercial in-memory systems have emerged, such as HyPer [76], SAP HANA [56], and Microsoft Hekathon [46], to name a few. These systems are heavily used in data-intensive tasks in the fields of information retrieval, machine learning, data mining and analysis [72, 132, 141].

Index search is one of the most critical parts of in-memory databases. It has been shown that the most amount of time for a query operation in in-memory database systems is the time spent on searching the index [164]. Therefore, the best performance of such database systems depends on efficient index searching. The efficiency of index searching largely depends on the choice of index data structures. Traditional disk-optimized schemes for index searching are either inefficient or not easily generalized for different search types. For example, binary search trees are inefficient, and hash-tables cannot support range queries.

Modern systems incorporate accelerators such as graphical processing units (GPUs) to offload computationally intensive work from main processors that offer great potential for fast index-searching. Because such GPU accelerators are specialized for compute-intensive,

data-parallel computation compared to traditional multi-core CPU systems, some data intensive tasks have been ported to GPU platforms in the past [29, 159]. In the same fashion, index searching operations can exploit the GPU platform to improve bulk index lookup if appropriate algorithms are designed and implemented efficiently.

6.1.2 Index Search Approaches

Indexes are used in modern database systems to facilitate faster lookups of keys for data. In these systems, a list of $\langle key, id \rangle$ tuples are organized in such a way that, given a specific key $key(i)$ or a range of keys $[key(i), key(j)]$, the index search subsystem returns the corresponding identifier(s) $id(s)$ of the key(s). The goal of any index searching algorithm is to deliver the identifiers as fast as possible, corresponding to a given set of keys.

Broadly speaking, there are two main approaches to fast index searching: (1) hash-based approaches and (2) tree-based approaches. Within the latter, an important sub-category is the radix tree-based approach.

Hash-based Systems: Hash-based systems are mainly used in key-value storage systems such as Memcached [57], Redis [129], and RAM-Cloud [124]. Hash-based systems are very efficient and perform well for point queries. However, these systems do not support range queries, which is a crucial functionality in many systems such as finding the transactions between two dates, finding top-k values within the range, etc. These systems also do not have good cache utilization in general [55].

Tree-based Systems: Tree-based systems, such as B+-tree, have been the de-facto standard for traditional disk-based databases [40] designed to obtain the best I/O throughput. However, the B+-tree does not utilize the increasing cache sizes properly and is deemed unusable for in-memory systems [91]. Consequently, many variations of B+-tree have been proposed, such as T-Tree [90], Cache Sensitive Search Trees (CSS-Trees) [119], Cache Sensitive B+-Trees (CSB+-Trees) [120], Δ-Tree [42], BD-Tree [43], Fast Architecture Sensitive Tree (FAST) [78], and Bw-tree [97]. Although these systems achieve better cache utilization, they are computationally expensive. Furthermore, the order of comparison operations in tree structures is harder to predict and result in undesirable stalling in processor pipelines [91].

Radix Tree-based Systems: Radix trees, such as the Judy Tree [18] and the Adaptive Radix Tree (ART) [91], do not rely on hashing or comparison. Instead, they create index trees based on the digital representation of search keys. One key property of searching with the radix tree is that the search operation requires $O(k)$ operations, where k is the length of the key, and it is independent of the number of keys to index. A radix tree has many

advantages over comparison-based trees:

- The height of a radix tree depends on the length of the keys, not on the number of keys
- Keys are kept in a lexicographic order, therefore any order of insertions results into the same tree
- Keys can be retrieved in sorted order, allowing fast range queries
- Keys are not stored explicitly, thus giving memory savings, yet can be re-constructed by tree traversal.

6.1.3 Organization

The rest of the chapter is organized as follows. The details of GPU-based radix tree implementation for index searching is discussed in Section 6.2. Benchmarks for the performance evaluation are discussed in Section 6.3. Experimental results are presented in Section 6.4. Finally, we summarize and conclude in Section 6.5.

6.2 GPU-based Radix Tree for Index Searching

To the best of our knowledge, radix-tree based index searching system has not been implemented and/or evaluated on the GPU. In this chapter, we implement and evaluate the performance of a radix tree-based index searching on the GPU. Furthermore, our implementation also supports range queries on the GPU, for which we identify radix trees as ideally suited. We present GRT (GPU-based Radix Tree), an efficient index searching implementation based on radix trees customized and optimized for single instruction multiple data (SIMD) operation, different key lengths, various key distributions, and point queries as well as range queries.

The adaptive radix-tree (ART) is the most efficient radix-tree based index searching system in the CPU [91]. ART has been demonstrated to be very memory efficient and yields high lookup throughput [91]. Our basic approach builds on the structures used by ART and adapts them for modern GPUs. A comprehensive performance study of GRT is presented here using benchmarks that index some data sets containing millions of keys. We also compare the run time performance against other existing competitive CPU and GPU-based methods. Performance evaluation demonstrates the high throughput achieved by our

implementation, clocking over 100 million lookup operations per second on large data sets of over 64 million 32-bit keys. We provide benchmarks and detailed run time data to support the results and findings.

6.2.1 Radix Tree Nodes

A radix tree has two types of nodes: inner nodes and leaf nodes. Inner nodes map partial keys to other nodes. Each leaf node represents a distinct key and stores the index value of the corresponding key. A lookup operation on a radix tree starts from the root. For each inner level, a partial key is extracted from the given key, and it is used to navigate to the child node of the next level. This process is repeated until a leaf node is reached. Therefore the lookup operation depends on the number of levels of the radix tree.

The number of levels of a radix tree depends on the size of the partial keys. If the size of a partial key is s and the size of a key is k, then the radix tree requires $\lceil \frac{k}{s} \rceil$ levels. For example, for 32-bit keys, if the size of a partial key is 4, then it would require 8 levels. Therefore, for faster lookup, it is desirable that the length of the partial key is large.

For a partial key of size s-bits, an inner node may store up to 2^s child pointers. In conventional radix trees, all inner nodes store the same number of pointers, that is, all inner nodes have 2^s pointers. When most of the child pointers are not used, the space required by conventional radix trees can be excessive [91]. To reduce the space required and utilize the benefits of radix tree, Leis et al. proposed an adaptive radix tree (ART) [91], which adaptively uses different sizes of inner nodes with the same partial key size. In their implementation, they used 8-bit partial keys and four different types of inner nodes with fan out of 4, 16, 48, and 256 branches. On CPUs, ART has been demonstrated to deliver high throughput lookups in high performance in-memory database systems such as HyPer [76]. However, adaptive radix trees have never been implemented or evaluated on GPUs. To exploit the performance gain from adaptivity, in our implementation, we build on adaptive radix trees as our index searching data structure and map them to the GPU architectural elements characterized by memory idiosyncrasies and single instruction multiple data (SIMD) processing style.

We implemented index searching on modern GPU architectures, with support for three basic operations of index searching: (1) inserting $\langle key, id \rangle$ tuples in a bulk mode, (2) performing the search for a single given key $\langle key \rangle$, a list of keys $\langle key_1, key_2, \ldots key_n \rangle$, or a range of keys $\left[key_{\min}, key_{\max} \right]$ and (3) modifying any given tuple $\langle key, id \rangle$ by changing the id of an already present key.

6.2.2 Benefits and Challenges of Using GPUs

GPUs are highly parallel, multi-threaded, many-core processors and are widely used for general purpose computing. Usage of GPU is prevalent in scientific computation and complex simulations. GPUs are also being increasingly used in big data analytics, machine learning, and data mining fields [52, 72, 132, 141]. A GPU contains streaming multiprocessor (SM), where SM is a group of core processors. Each core processor executes only one thread at a time. All the core processors can execute their corresponding threads simultaneously. If some threads perform operations with high latencies, those are put into waiting state and other pending threads are executed. Therefore GPUs increase throughput by keeping the processors busy. All thread management, including creating and scheduling threads, are performed entirely in hardware with negligible overheads.

Although GPUs are capable of accelerating data-intensive applications greatly, the idiosyncrasies of the memory architecture make GPU programming challenging.

Challenge 1: *Limited Memory Capacity* The capacity of GPU memory is much smaller than that of CPU main memory [149]. For example, the memory size of a commodity GPU is less than a dozen gigabytes, while that of a commodity server can be hundreds of gigabytes. Therefore efficient memory usage is a still a challenge for GPU programming, relative to CPU programming.

Challenge 2: *Data Structure and Algorithm Optimizations on GPUs* For the best performance, the data transfer between CPU and GPU memory must be minimal. GPU-based execution also works best when the memory access is coalesced and bank-conflict-free. Hence, complex CPU based data structures are not well suited for the GPU. For example, a simple two-dimensional array works best when it is converted into a large one-dimensional array.

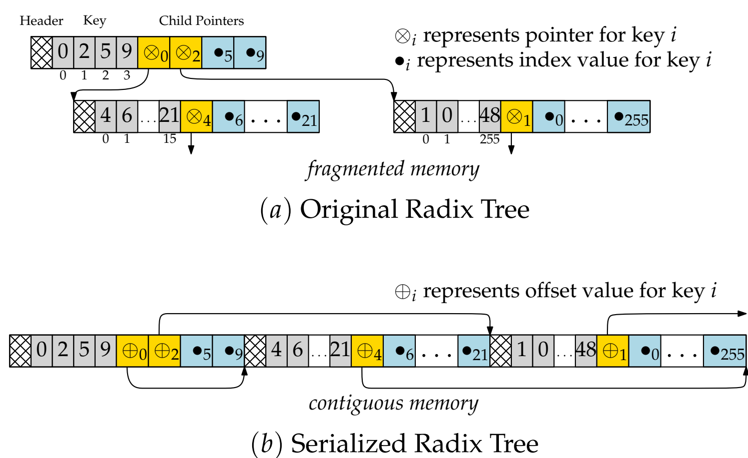
6.2.3 Tree Interface and Serialization

Before using the radix tree for searching, the tree has to be built from the keys and stored. The tree can be built with the CPU in main memory or by the GPU in GPU (device) memory. Building the radix-tree in the CPU is relatively easier as CPU efficiently deals with dynamic memory allocation. After the radix-tree is built on the CPU, it must be transformed into another memory efficient structure, suitable for efficient GPU lookup operations. The original ART structure uses memory pointers, which is inefficient to use in GPU. For efficient memory accesses, we use offset values instead of pointers as shown in Figure 6.1. The offset value denotes how far the child is located from the start of the pointer. Therefore, the tree traversal operations ($\texttt{root} \rightarrow \texttt{child}[i]$) become simpler ($\texttt{root} + \texttt{offset}$)

operations on the GPU, where `root` is the starting point of the tree. The tree is serialized using a depth-first search (DFS) traversal into a contiguous byte array for efficient lookup operations on the GPU. Therefore, in our first implementation, the main instance of the radix tree is stored in main memory and a GPU-efficient copy is stored in a contiguous chunk of the GPU memory. However, whenever a new item is inserted in the radix-tree, we have to serialize the structure again and copy it to the GPU memory. For applications where insertions and deletions are frequent, it may take a long amount of time. To avoid this, we implemented another efficient version where the radix-tree is built only using the offset-based structure. In this case, we allocate a large chunk of GPU memory in advance and build the radix-tree in this chunk of memory. We can determine the approximate amount of memory required to store the radix-tree based on the key distribution as shown in the experimental evaluation section.

For efficiency, the full tree is copied to the GPU memory. As the first level of the tree is always accessed for any lookup query, many concurrent lookups would cause memory contentions. Such memory contentions are avoided by caching the first level of the tree in the shared memory at the block level. The individual threads work on different parts of memory, and, therefore, do not have contentions. Since the shared memory of GPU is significantly faster than the global memory, the lookup time is reduced.

Our GPU implementation is designed primarily for lookup operations, which are typically the most computationally intensive portion, and more numerous than insert/update operations on the tree. Therefore, optimization of lookup operations is the primary goal of this work.



(a) Original Radix Tree

(b) Serialized Radix Tree

Figure 6.1: *Serializing the radix tree for GPU*

6.2.4 Query Types

There are two main query categories that require index searching support.

Point Queries: The most typical index searching operation in a database system is to find the index from a single key, known as point queries. When tree-based structures are used for point query searching, the average run time for a search operation is proportional to the depth of the tree. For faster searching, hash tables are often used, which have an amortized $\mathcal{O}(1)$-time search. The actual performance of the hash tables is highly dependent on the table structure and hash functions [10, 91]. Another popular method for point queries is radix-tree which offers $\mathcal{O}(k)$-time search operations, where k is the size of the key.

Range Queries: A range query is a searching operation that retrieves all indices where the value of the key is between an upper and lower boundary. Range queries appear in many applications, such as using timestamps as keys and finding the records contained within two timestamps.

Many indexing systems that have been tuned to deliver fast point queries are unsuitable for range queries. For example, hash-based systems support very fast point queries, but they have limited support for range queries. One simple way to perform range queries is to repeat the point query operation for all keys in the range. This may yield acceptable performance where the range is small, but for large ranges, this will not do well because it wastes computation in the form of many duplicate traversals of the tree. Tree-based systems such as comparison-based and radix-based systems use an ordering of keys and therefore support efficient range queries.

In many cases, such as for finding names or phone numbers with a common prefix, a variation of range queries called prefix queries is also used. As the name suggests, prefix query searches for the indices of keys with a common prefix. Hash-based and comparison-based tree systems do not suit well for prefix queries. Only radix tree-based systems natively support prefix queries.

6.3 Benchmarks

To evaluate the performance of index searching on GPUs, we vary the key sizes, key distributions, and lookup workload patterns. In this section, an overview of the key and lookup benchmarks is provided, followed by additional details of the different lookup benchmarks.

6.3.1 Overview

The following are the key and lookup variants used for experiments in the performance study.

- **Key Variants.** The lookup operations of any radix-based tree is dependent on the key structure. In this work, we considered multiple variations of key structure in terms of lengths and values. We considered both synthetic and real-world key values to measure the performance of GRT.

  GRT is capable of handling any key length. However, for the purposes of the performance study, we considered three commonly used key lengths in our experiments: 4-byte, 8-byte, and 16-byte keys. Most of the previous works on index searching supported 4-byte keys for benchmarking, therefore we follow the convention and exercise 4-byte keys for the benchmarking. We also used 8-byte keys for benchmarking to compare with [164], which uses 8-byte keys. Many real-world keys are larger than these sizes. To measure the performance of real-world keys on the GPU, we also evaluated 16-bytes keys. We varied the number of keys across experimental scenarios. Four distinct set of keys were considered in this work: 64 thousand, 1 million, 16 million, and 64 million keys. All sizes are similar to the ones covered in the prior literature [78, 91, 164]. A summary of the keys used in the experiments is shown in Table 6.1.

  Table 6.1: *Datasets used in the evaluation*

Key Type	# of keys	Length
Sparse	64**K**, 1**M**, 16**M**, 64**M**	4 Bytes
Dense	64**K**, 1**M**, 16**M**, 64**M**	4 Bytes
ISBN	7**M**	8 Bytes
Music Identifiers	9**M**	16 Bytes

- **Lookup Variants.** To measure the performance of GRT, multiple benchmarks are considered. Three types of lookup operations are covered in these benchmarks: (1) a singular lookup for a key, (2) a bulk lookup for an array of keys, and (3) a range lookup between two keys. Singular lookup is useful in applications residing completely in the GPU. Bulk lookup is useful in applications residing either on GPU or CPU. Singular lookup provides the best latency while bulk lookup provides the best throughput. Range lookup is used in many database applications, and it can be effectively utilized on both CPU and GPU. A summary of all the benchmarks is provided in Table 6.2.

Table 6.2: *List of Benchmarks*

Code	Description
1S	**Bulk** (**1**) lookup with ***S****parse* keys
1D	**Bulk** (**1**) lookup with ***D****ense* keys
1A	**Bulk** (**1**) lookup with 8-byte (**A**) *real-world* keys
1B	**Bulk** (**1**) lookup with 16-byte (**B**) *real-world* keys
2S	**Singular** (**2**) lookup with ***S****parse* keys
2D	**Singular** (**2**) lookup with ***D****ense* keys
3S	**Hierarchical** (**3**) lookup with ***S****parse* keys
3D	**Hierarchical** (**3**) lookup with ***D****ense* keys
4S	**Range** (**4**) lookup with ***S****parse* keys
4D	**Range** (**4**) lookup with ***D****ense* keys

The main performance metric used here is the throughput, measured as the number of lookups completed per second. Average time per lookup can be readily inferred as the inverse of the throughput.

6.3.2 Lookup Benchmarks

- **Bulk Lookup Benchmarks.** The bulk lookup is used to compare the performance of GRT with other efficient implementations reported in the literature. In the bulk lookup, the indices to be looked up are provided in an array of keys. The following variations of these bulk lookup arrays are considered: (1) *Sparse distribution*, where each key is unique and chosen uniformly at random from $[1, 2^k)$ where k is the size of the key in bits, and (2) *Dense distribution*, where every key is in the range $[0, 1, 2, \dots, n-1]$ where n is the number of keys to index.

  Further, in our performance evaluation with the bulk lookup benchmarks, we have also exercised two real-world keysets: (1) ISBN numbers from a book database [111] and (2) Music Identifiers from a music database [107].

- **Singular and Hierarchical Lookup Benchmarks.** In many GPU-based parallel applications, the GPU threads independently performs many lookup operations to fetch data. The singular lookup benchmarks evaluate the lookup performance of such systems. In this benchmark, each GPU thread independently generates many random keys and

performs point queries for each key. We also use both sparse and dense keys in this experiment.

So far, the keys are considered to be unrelated to each other. But in many applications, such as genealogy [35], communication threads in social network analysis, phylogenetic tree [113], the keys have a hierarchical relationship resulting into self-referenced foreign keys. Such applications require the information or data from its parents leading up to the root of the tree. In such cases, an efficient lookup is needed to trace back the parent using a single composite lookup rather than having multiple singular lookups to achieve the same. To evaluate such scenarios, we introduce an additional benchmark that exercises the hierarchical lookups. In this benchmark, let $\mathcal{K}$ be the set of keys. Then, for any key $k \in \mathcal{K}$, we have a parent key $P(k) \in \mathcal{K}$. The root key does not have a parent key. Since we use an m-ary tree, each key has m children keys. Given a key $k \in \mathcal{K}$, the hierarchical benchmark traces every node in the path back to the root of the m-ary tree.

- **Range Lookup Benchmarks.** In a range lookup, when the user provides a pair of keys k_{min} and k_{max}, all the indices of keys k are returned such that $k_{\text{min}} \leq k \leq k_{\text{max}}$, where the size of range r is defined as $r = k_{\text{max}} - k_{\text{min}} + 1$. In this experiment, the number of keys is kept to the maximum available and the range size is varied. We consider both the *sparse distribution* and *dense distribution* distribution of keys, as mentioned earlier. We also measure the performance of range search as the number of range queries processed per second and also in terms of throughput as the number of indices retrieved per second.

6.4 Performance Study

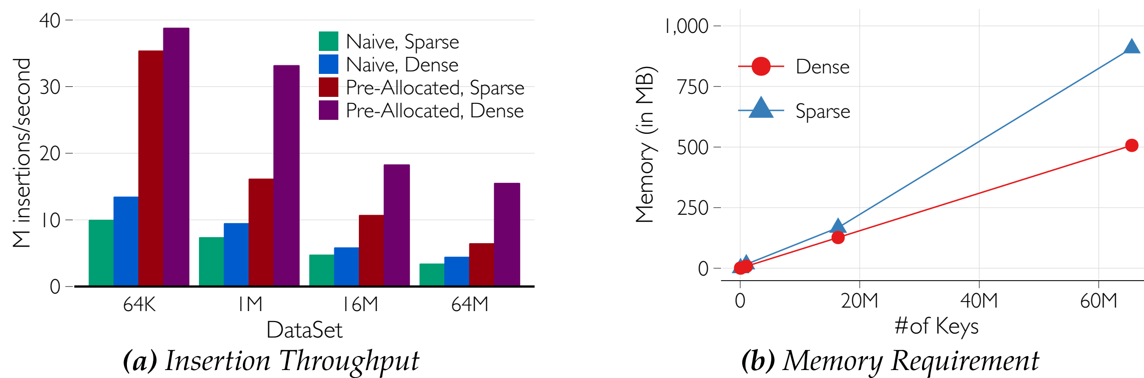
In this section, we evaluate the performance of GRT under different workloads and configurations, and observe that GRT achieves excellent throughput for all workloads.

In the experiments, the CPU is a *Xeon E5* with a 2.5 GHz clock and a memory of 16 GB with a clock rate of 2.1 GHz with a peak bandwidth of 68 GB/s. The GPU is a *nVidia Tesla K80* with processor clock rate of 562MHz, memory rate of 2.5 GHz and peak bandwidth of 240 GB/s. The operating system is *Ubuntu 12.04.5 LTS*, and all software on this machine was compiled with GNU *gcc 4.6.3* with optimization flags *-O3*. The *CUDA compilation tools V6.5* were used for the GPU code along with *nvcc* compiler. All CPU based experiments were performed using a single core.

6.4.1 GPU Overhead

Here we present performance results regarding the creation and serialization performance of GRT. As mentioned earlier, we use two approaches to building the GPU radix tree. In the naive approach, the pointer based radix-tree is built on the CPU and then serialized and copied to the GPU memory. In the pre-allocation based approach, we allocate a big chunk of memory in the GPU and directly build the radix tree on that memory, which supports further insertions and deletions of indices in the GPU. However, our experiments suggest that building the tree only using the GPU memory takes more time than using the CPU memory. For example, creating a radix tree with $16M$ sparse keys require about 1.54 seconds using the CPU memory, whereas using only the GPU memory requires about 16 seconds. Therefore, we use a hybrid approach, where the initial pointer less offset-based radix-tree is built in the CPU and copied to the GPU memory. Future updates to the radix-tree could be performed directly in the GPU memory without using the CPU.

Insertion Throughput. Figure 6.2(a) demonstrates the insertion throughput of building the radix tree using both approaches for sparse and dense keys. Note that the insertion throughput of the naive approach is similar to the one reported in [91]. The pre-allocate based hybrid approach yields better throughput than the naive scheme for all datasets. Also, note that dense keys require less time to build the tree.



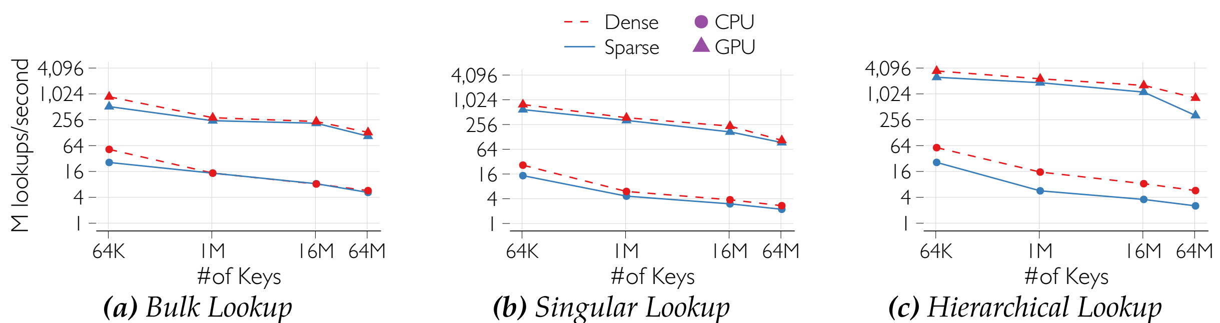
(a) *Insertion Throughput* **(b)** *Memory Requirement*

Figure 6.2: *Replicating Radix Tree for the GPU*

Memory requirement. Figure 6.2(b) shows the memory required to store the radix tree in the GPU. The required memory increases linearly with the data size. Adaptive radix trees use lazy expansion; for this reason, dense keys expand slower and have better space efficiency. For 64M entries, about 900 MB GPU memory is required for sparse keys, which is about 14 bytes per search key. However, for dense keys, only 531 MB GPU memory is required, which is about 8 bytes per key. For the pre-allocation based approach, we use 16 and 8 bytes per sparse and dense keys respectively.

6.4.2 Bulk Lookup

We performed bulk lookup operations on the sparse and dense keys* from the datasets listed in Table 6.1. The index searching on the CPU with adaptive radix trees was performed using the unaltered source code of the ART system [91]. We used 512 threads for the GPU implementations. In each experiment, more than 100 million lookup operations were performed, in order to eliminate noise. The results of the experiments are shown in Figure 6.3(a).



***(a)** Bulk Lookup* ***(b)** Singular Lookup* ***(c)** Hierarchical Lookup*

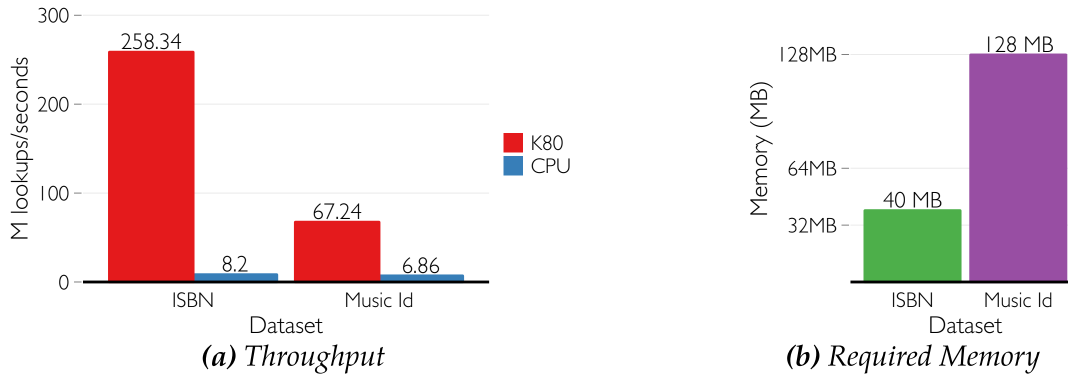
***Figure 6.3:** Bulk, Singular, and Hierarchical lookup throughputs for 4-byte keys*

From the figure, it is clear that the dense key lookups have better throughput in most of the cases. With the increase in the number of keys, sparse keys become dense, and this advantage diminishes. For example, for 64K dense keys, only two-byte levels are sufficient for all the search operations (when bytes are accessed in little endian format), whereas many sparse keys would require more than two levels. Therefore, the performance of dense keys is significantly better. For 64 million entries, we achieve a throughput of about 100 million lookups per second for the sparse keys and 130 million lookups per second for the dense keys.

6.4.3 Bulk Lookup with Real-World Keys

To test the performance of GRT, we performed bulk lookup operations on real world keys. We collected two sets of keys from two popular open source databases. The first set of keys comes from OpenLibrary.com [111] containing around 7 million book entries containing ISBN-13 numbers. A 13-digit ISBN number fits into an 8-byte integer. We also collected a set of keys from MusicBrainz.com [107] database, which contains around 9 million song entries. Each entry has a 16-byte id.

*dense keys are shuffled randomly for the experiments



***(a)** Throughput* ***(b)** Required Memory*

***Figure 6.4:** Bulk lookup for ISBN (8 Bytes) and Music Id keys (16 Bytes)*

Figure 6.4 exhibits the throughput and required memory for the lookup operations. As observed from the figures, we achieved a throughput of 258.34 million lookups per second on the Tesla K80 GPU.

ISBN-13 keys are very structured. The first three digits represent a prefix (either 978 or 979) as per the standard ISBN specification. The next two digits represent the registration group element. The four digits next to it represent the publisher identifier. The title and check digit is represented by the next three digits and one digit respectively. For these kinds of structured keys, the radix-based approach is well suited.

We expect similarly suitable lookup behavior for analogous key structures found in many other domains such as international article number (EAN) barcodes, electronic product codes (EPC), stock keeping unit (SKU), social security numbers (SSN), telephone numbers, and so on.

The music-identifying keys are randomly generated unique identifiers, and the distribution of keys in the lookups is very sparse. As a result, we observe only around 67 million lookups per second. Nevertheless, GRT still performs better than the CPU version, providing a speedup of around 10×.

6.4.4 Singular and Hierarchical Lookup

In the singular lookup benchmark, each GPU kernel thread generates a random key and perform a point query. In this experiment, each GPU thread performs millions of point queries to eliminate noise. Figure 6.3(b) exhibits the singular lookup throughputs for both sparse and dense keys. The throughputs are similar to the bulk lookup because the singular lookup simply generates random keys instead of using a given array in the bulk

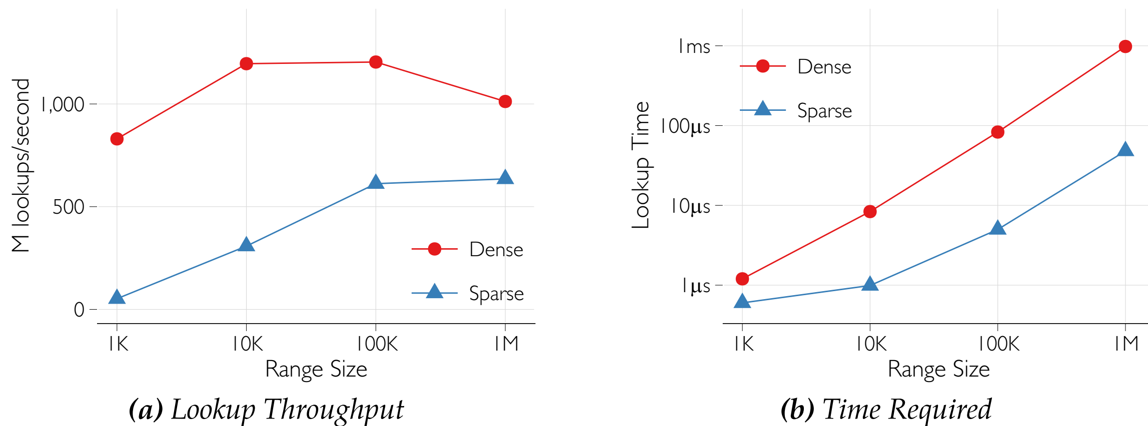
lookup.

In the hierarchical lookup benchmark, we assume that there is an m-ary parent-child relationship among the keys. Therefore each parent has m children keys, and so on. In this experiment we set the value $m = 10$. Each GPU thread randomly selects a key and repeatedly performs searches towards the parent, until the root key is reached. An m-ary tree has the depth of $O(\log_m N)$ for N keys. Hence, in the worst case, there would be $O(\log_m n)$ such search operations per key. To identify how the hierarchical relationship affects the search performance, we investigate yet another lookup variant. In this variant, each GPU thread selects a random key and searches the index of the key.

Figure 6.3(c) shows that the hierarchical lookup achieves about two times faster throughput than the singular lookup operations. This behavior is observed across both CPU and GPU as a similar trend. The main reason is the way searches are performed. The keys closer to the root are searched more often than the keys which are closer to the leaves. Therefore, they are stored in the cache for a longer time, effectively improving the throughput.

6.4.5 Range Lookup

In this experiment, we evaluate the performance of range queries of GRT. To the best of our knowledge, there is no range query implementation on the GPU. All the experiments were performed with 64 million keys. We used both sparse and dense keys for this experiment. Here we used a set of four range sizes $\mathcal{R} = \{10^3, 10^4, 10^5, 10^6\}$. In each GPU thread, a random key k is picked and a range query operation is performed over the range $[k, k + r - 1]$ for each $r \in \mathcal{R}$.



(a) *Lookup Throughput* **(b)** *Time Required*

Figure 6.5: *Performance of range queries*

Figure 6.5(a) shows the throughput performance of range queries on GPU. As the figures

show, the throughput of dense keys is much higher than the corresponding throughputs of sparse keys. The range queries are implemented using depth first search tree traversal. For any given range, the inner nodes traversed by the algorithm for dense and sparse keys are approximately the same. However, as the dense distributions contain more keys (leaf nodes) than the sparse distributions in the same range, dense distributions yield better throughput. Also observe that for sparse keys, throughput increases when the range is increased, but for the largest range (1M) the throughput does not increase much. A similar trend is also observed in dense keys, but in this case, the throughput decreases by a significant amount for the largest range. This is due to the fact that, the traversal algorithm has to process more inner nodes compared to the number of leaf nodes. For range size of 10^5, range query operation processes an average 394 inner nodes for dense keys, whereas, for range size of 10^6, the operation requires processing 3925 inner nodes.

Figure 6.5(b) exhibits the time required to perform a single range query by our algorithm. The number of range queries per second decreases with increasing range size as expected. However, range queries for sparse keys run much faster. This is due to the fact that, sparse distributions contain less number of keys in a given range as compared to dense distributions, and therefore have to process fewer leaf nodes.

As seen from the above figures, our algorithm is capable of performing fast range queries with high throughput, which is not possible for hash-based systems.

6.4.6 Update Operations

Our pre-allocation based approach provides support for delete, modify, and insert operations directly on the GPU. However, if one wants to update/insert a lot of indices, the CPU should be used in conjunction with the GPU. Our experiments show that for an already constructed $16M$ sparse keys, we achieve an update throughput of 0.8 million index operations per second using only the GPU, whereas along with the CPU yields a throughput of 8 million index operations per second.

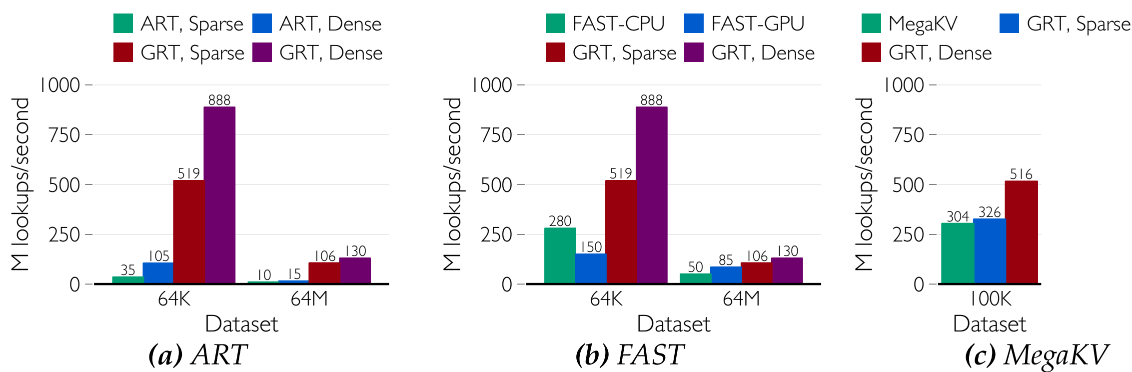
6.4.7 Comparing GRT with Other Index Searching Systems

We compare the throughput of GRT with existing CPU and GPU based index searching systems. Here we only consider ART [91], FAST [78], and Mega-KV [164]. Due to the unavailability of source codes of these systems, we use the best performance numbers from their corresponding articles.

We show the performance results on three datasets with 64K, 100K, and 64M keys. The 64K

Table 6.3: *Machine configurations of existing index searching systems*

	CPU Specifications				GPU Specifications		
	ART	FAST	MegaKV	GRT	FAST	MegaKV	GRT
Processor	Core i7	Core i7	Xeon E5	Xeon E5	GTX 280	GTX 780	Tesla K80
Clock Rate (GHz)	3.2	3.2	2.6	2.5			
Peak BW (GB/s)	51.2	30	59.7	68	141.7	288.4	240
Peak GFLOPs					933.3	3977	4370



(a) *ART* ***(b)*** *FAST* ***(c)*** *MegaKV*

Figure 6.6: *Comparing GRT with other systems*

and 64M datasets consist of 4-byte keys; they are used in performance evaluation on both ART and FAST. Therefore, the comparison remains fair. We also consider both the sparse and dense key distribution as was done in the previous literature. The 100K keys with a key length of 8-byte dataset is used specifically to compare with Mega-KV, considering that the largest block size in Mega-KV is also the same.

GRT vs. ART. Figure 6.6(a) shows the throughput comparison between ART and GRT. GRT obtained a magnitude of 15× and 8× higher throughput over ART. We also observe a similar pattern for sparse and dense keys, as this pattern emerges because of the radix-tree structure.

GRT vs. FAST. Figure 6.6(b) exhibits the throughput comparison between GRT and FAST (both CPU and GPU). For smaller key sizes, the CPU implementation of FAST is faster than the GPU version because of caching effects. Both the sparse and dense key versions of GRT have about two to three fold performance gain over the FAST implementations. For larger key sizes, such as 64M keys, however, the performance of GRT decreases rapidly compared to the FAST version. Nevertheless, the actual throughput of GRT always remains better than that of FAST. Note that the comparison may not be entirely fair here: as the source

codes for FAST on the GPU is not available online, we were unable to determine how FAST would perform in modern GPUs.

GRT vs. Mega-KV. Mega-KV is the latest GPU-based system that uses a hash table for index searching. Figure 6.6(c) shows the throughput comparison between GRT and Mega-KV. For 100K keys, both systems have similar throughput, with GRT performing better than Mega-KV by a small margin. Note that both the corresponding GPUs for GRT and Mega-KV have almost similar FLOPs and memory bandwidth, making the comparison fair.

In additional experiments, we used three different GPUs that differ in their capacities in terms of processor clock rate, memory capacity, and peak GFLOPs. The GTX 580 model was released in 2010, the Tesla K20X in 2012, and the Tesla K80 in 2014. The only common feature among the models is the bandwidth, as the bandwidth of GPU has not increased significantly over the years. We obtained nearly similar throughput rates from all of the GPUs in the bulk lookup performance. The same phenomenon is observed in Figure 6.3. Although the throughput is almost doubled for the singular lookup performance, this improvement happened in every other GPU. Therefore, the difference in GPU architecture is not contributing significantly in index searching. This is probably because the index searching systems require extensive memory access operations and are memory-bound by nature. Therefore the determining factor for index searching operation seems to be the bandwidth. Also, note that dense keys use less memory and therefore perform better.

6.5 Conclusion and Future Work

In this work, we developed an efficient GPU-based Radix Tree (GRT) for index searching with high throughput on GPUs. We serialized of the radix tree to a low memory footprint data-structure to achieve extremely efficient lookup operations on GPUs. We augmented additional index searching performance benchmarks to the set of well-known existing ones and, evaluated GRT on all the benchmarks. The performance study included a comprehensive analysis, taking into account the diversity of keys, lookup operations and run time effects with real-world data. We compared GRT to the most recent and the best available index searching systems. Our experiments indicate that GRT meets or betters the performance of other index searching systems currently available both on the CPU and the GPU. For a large dataset of 64 million keys, GRT achieved a throughput of 106 million lookups per second for sparse keys and 130 million lookups per second for dense keys. For the same dataset, excellent range query performances are achieved, yielding over 1000 million lookups per second for sparse keys and over 600 million lookups per second for

large range sizes for dense keys.

Since index searching is a bottleneck for databases, especially for main memory systems, GRT can be directly applied to such systems to achieve significant performance gains. Additionally, it is also possible for GRT to be extended to high-performance key-value storage systems, where values are stored in memory in conjunction with the keys.

In our study, the implementation was restricted to a single GPU for storing and managing index structures that fit within a GPU's memory. In future, we plan to explore index searching operations on larger datasets with a greater number of keys on multiple GPUs.

Acknowledgment

This chapter has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Dept. of Energy. Accordingly, the U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

Chapter 7

Concluding Remarks

We presented HPC-based parallel algorithms for generating massive random networks using the preferential attachment, Chung-Lu, block two-level Erdős-Rényi, and stochastic block models. The algorithms produce large in-memory networks with millions to billions of edges in few minutes. Random networks play an important role in designing and developing networking analysis and mining algorithms, simulation, and benchmarking. Our algorithms are highly efficient and scalable to a large number of processors. We extensively analyzed our algorithms and provided theoretical bounds on run time and memory. We designed and developed novel partitioning and load balancing schemes to achieve the best performance. These partitioning techniques are quite general and can be applied to other problems where the computational cost is well defined. Our parallel load balancing algorithm has also been applied successfully to develop other network analysis algorithms for better distribution of work loads. We also presented an efficient GPU-based algorithm for the index searching problem. Many graph database systems require fast lookup throughputs. Our GPU-based algorithm is able to match the demand. Further, we also developed fast range queries using radix-trees and showed its usefulness. We achieved the faster throughputs ever reported in literature for both point and range queries on the GPU.

Bibliography

[1] James Abello, Mauricio G. C. Resende, and Sandra Sudarsky. "Massive Quasi-Clique Detection." In: *LATIN: Theoretical Informatics*. Springer, 2002, pp. 598–612 (page 1).

[2] Abhijin Adiga, Anil Kumar S. Vullikanti, and Dante Wiggins. "Subgraph Enumeration in Dynamic Graphs." In: *IEEE International Conference on Data Mining*. IEEE, 2013, pp. 11–20 (page 1).

[3] Charu C. Aggarwal and Haixun Wang. *Managing and Mining Graph Data*. 1st. Vol. 40. Advances in Database Systems. Springer US, 2010, p. 600 (page 5).

[4] Charu C. Aggarwal, Na Ta, Jianyong Wang, Jianhua Feng, and Mohammed J. Zaki. "Xproj: A Framework for Projected Structural Clustering of Xml Documents." In: *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2007, pp. 46–55 (page 1).

[5] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. "A Scalable Processing-In-Memory Accelerator for Parallel Graph Processing." In: *Annual International Symposium on Computer Architecture*. ACM. ACM Press, 2015, pp. 105–117 (page 5).

[6] William Aiello, Fan Chung, and Linyuan Lu. "A Random Graph Model for Power Law Graphs." In: *Experimental Mathematics* 10.1 (2001), pp. 53–66 (page 78).

[7] Maksudul Alam and Maleq Khan. "Parallel Algorithms for Generating Random Networks With Given Degree Sequences." In: *International Journal of Parallel Programming*. Springer US, 2015, pp. 1–19 (pages 5, 71, 82, 88).

[8] Maksudul Alam, Maleq Khan, and Madhav V. Marathe. "Distributed-Memory Parallel Algorithms for Generating Massive Scale-Free Networks Using Preferential Attachment Model." In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM Press, 2013, pp. 1–12 (page 5).

[9] Réka Albert, Hawoong Jeong, and Albert-László Barabási. "Error and Attack Tolerance of Complex Networks." In: *Nature* 406.6794 (2000), pp. 378–382 (pages 4, 71).

[10] Victor Alvarez, Felix Martin Schuhknecht, Jens Dittrich, and Stefan Richter. "Main Memory Adaptive Indexing for Multi-Core Systems." In: *International Workshop on Data Management on New Hardware*. ACM Press, 2014, pp. 1–10 (page 106).

[11] Renzo Angles and Claudio Gutierrez. "Survey of Graph Database Models." In: *ACM Computing Surveys* 40.1 (2008), pp. 1–39 (page 5).

[12] David A. Bader and Kamesh Madduri. "Parallel Algorithms for Evaluating Centrality Indices in Real-World Networks." In: *International Conference on Parallel Processing*. 2006, pp. 539–547 (page 4).

[13] Eman Badr and Lenwood S. Heath. "CoSREM: A Graph Mining Algorithm for the Discovery of Combinatorial Splicing Regulatory Elements." In: *BMC Bioinformatics* 16.1 (2015), p. 285 (page 1).

[14] Grey Ballard, Tamara G. Kolda, and Todd Plantenga. "Efficiently Computing Tensor Eigenvalues on a GPU." In: *IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum* (2011), pp. 1340–1348 (page 5).

[15] Albert-László Barabási and Réka Albert. "Emergence of Scaling in Random Networks." In: *Science* 286.5439 (1999), pp. 509–12 (pages 1, 3, 14, 71).

[16] Christopher L. Barrett, Richard J. Beckman, Paula Elaine Stretz, and Bryan Lewis. "Generation and Analysis of Large Synthetic Social Contact Networks." In: *Simulation*. 2009, pp. 1003–1014 (pages 63, 79).

[17] Christopher L. Barrett, Stephen G. Eubank, Anil Kumar S. Vullikanti, and Madhav V. Marathe. "Understanding Large-Scale Social and Infrastructure Networks: A Simulation Based Approach." In: *SIAM News* 37.4 (2004), pp. 1–5 (page 3).

[18] Doug Baskins. *Judy Arrays* (page 101).

[19] Vladimir Batagelj and Ulrik Brandes. "Efficient Generation of Large Random Networks." In: *Physical Review E* 71.3 Pt 2A (2005), p. 036113 (pages 1, 3–5, 15, 45, 73, 75).

[20] Edward A. Bender and E. Rodney Canfield. "The Asymptotic Number of Labeled Graphs With Given Degree Sequences." In: *Journal of Combinatorial Theory, Series A* 24.3 (1978), pp. 296–307 (page 72).

[21] Stefano Berretti, Alberto Del Bimbo, and Enrico Vicario. "Efficient Matching and Indexing of Graph Models in Content-Based Retrieval." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23.10 (2001), pp. 1089–1105 (page 1).

[22] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. "A Large Time-Aware Web Graph." In: *ACM SIGIR Forum* 42.2 (2008), p. 33 (pages 79, 91, 94).

[23] Paolo Boldi and Sebastiano Vigna. "The WebGraph Framework I: Compression Techniques." In: *International World Wide Web Conference*. ACM. 2004, pp. 595–601 (page 1).

[24] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. "Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge." In: *ACM SIGMOD International Conference on Management of Data*. ACM. 2008, pp. 1247–1250 (page 5).

[25] Karsten M. Borgwardt, Cheng Soon Ong, Stefan Schönauer, S. V. N. Vishwanathan, Alex J. Smola, and Hans-Peter Kriegel. "Protein Function Prediction via Graph Kernels." In: *Bioinformatics* 21.Suppl 1 (2005), pp. i47–i56 (page 1).

[26] Robert A. Bridges, John P. Collins, Erik M. Ferragut, Jason A. Laska, and Blair D. Sullivan. "Multi-Level Anomaly Detection on Time-Varying Graph Data." In: *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. ASONAM '15. Paris, France: ACM, 2015, pp. 579–583 (page 71).

[27] Sergey Brin and Lawrence Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine." In: *Computer Networks and ISDN Systems* 30.1-7 (1998), pp. 107–117 (page 1).

[28] Jan F. Broenink. "Introduction to Physical Systems Modelling With Bond Graphs." In: *SiE Whitebook on Simulation Methodologies*. 1999, pp. 1–31 (page 1).

[29] John Canny and Huasha Zhao. "Big Data Analytics With Small Footprint: Squaring the Cloud." In: *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2013, pp. 95–103 (page 101).

[30] Jean M. Carlson and John Doyle. "Highly Optimized Tolerance: A Mechanism for Power Laws in Designed Systems." In: *Physical Review E* 60.2 (1999), p. 1412 (pages 1, 3, 71).

[31] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. "R-MAT: A Recursive Model for Graph Mining." In: *SIAM International Conference on Data Mining*. 2004, pp. 442–446 (pages 1, 3, 4, 71).

[32] Deepayan Chakrabarti, Yang Wang, Chenxi Wang, Jurij Leskovec, and Christos Faloutsos. "Epidemic Thresholds in Real Networks." In: *ACM Transactions on Information and System Security* 10.4 (2008), pp. 1–26 (page 1).

[33] David P. Chassin and Christian Posse. "Evaluating North American Electric Grid Reliability Using the Barabási-Albert Network Model." In: *Physica A* 355.2-4 (2005), pp. 667–677 (pages 3, 4).

[34] Amlan Chatterjee, Sridhar Radhakrishnan, and John K. Antonio. "Counting Problems on Graphs: GPU Storage and Parallel Computing Techniques." In: *IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE Computer Society, 2012, pp. 804–812 (page 5).

[35] Feng Chen, Aaron J. Mackey, Christian J. Stoeckert, and David S. Roos. "OrthoMCL-DB: Querying a Comprehensive Multi-Species Collection of Ortholog Groups." In: *Nucleic Acids Research* 34.1 (2006), pp. D363–D368 (page 109).

[36] Hsinchun Chen, Roger HL Chiang, and Veda C Storey. "Business Intelligence and Analytics: From Big Data to Big Impact." In: *MIS Quarterly: Management Information Systems* 36.4 (2012), pp. 1165–1188 (page 1).

[37] Prasann Choudhari, Eikshith Baikampadi, Paresh Patil, and Sanket Gadekar. "Parallel and Improved PageRank Algorithm for GPU-CPU Collaborative Environment." In: *International Journal of Computer Science and Information Technologies* (2015) (page 5).

[38] Fan Chung and Linyuan Lu. "Connected Components in Random Graphs With Given Expected Degree Sequences." In: *Annals of Combinatorics* 6.2 (2002), pp. 125–145 (pages 71, 73).

[39] Fan Chung and Linyuan Lu. "The Average Distances in Random Graphs With Given Expected Degrees." In: *National Academy of Sciences of the United States of America* 99.25 (2002), pp. 15879–15882 (pages 1, 3, 4, 6, 44, 45, 71, 73).

[40] Douglas Comer. "Ubiquitous B-Tree." In: *ACM Computing Surveys* 11.2 (1979), pp. 121–137 (page 101).

[41] Alain Cosnuau. "Computation on GPU of Eigenvalues and Eigenvectors of a Large Number of Small Hermitian Matrices." In: *Procedia Computer Science* 29 (2014), pp. 800–810 (page 5).

[42] Bin Cui, Beng Chin Ooi, Jianwen Su, and Kian-Lee Tan. "Contorting High Dimensional Data for Efficient Main Memory KNN Processing." In: *ACM SIGMOD International Conference on Management of Data*. ACM, 2003, pp. 479–490 (page 101).

[43] Bin Cui, Beng Chin Ooi, Jianwen Su, and Kian-Lee Tan. "Main Memory Indexing: The Case for BD-tree." In: *IEEE Transactions on Knowledge and Data Engineering* 16.7 (2004), pp. 870–874 (page 101).

[44] Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis. "Frequent Substructure-Based Approaches for Classifying Chemical Compounds." In: *IEEE Transactions on Knowledge and Data Engineering* 17.8 (2005), pp. 1036–1050 (page 1).

[45] Michael Dewing. *Social Media: An Introduction Social Media*. 2010-03-E. Cambridge University Press, 2012 (page 5).

[46] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. "Hekaton: SQL Server's Memory-Optimized OLTP Engine." In: *International Conference on Management of Data*. ACM. ACM Press, 2013, p. 1243 (pages 6, 100).

[47] Hristo Djidjev, Guillaume Chapuis, Rumen Andonov, Sunil Thulasidasan, and Dominique Lavenier. "All-Pairs Shortest Path Algorithms for Planar Graph for GPU-accelerated Clusters." In: *Journal of Parallel and Distributed Computing* 85 (2015), pp. 91–103 (page 5).

[48] Sergey N. Dorogovtsev and José Fernando Ferreira Mendes. "Evolution of Networks." In: *Advances in Physics*. Vol. 51. 4. 2002, pp. 1079–1187 (page 37).

[49] Sergey N. Dorogovtsev, José Fernando Ferreira Mendes, and Alexander N. Samukhin. "Principles of Statistical Mechanics of Uncorrelated Random Networks." In: *Nuclear Physics B* 666.3 (2003), pp. 396–416 (page 12).

[50] Nhat Tan Duong, Quang Anh Pham Nguyen, Anh Tu Nguyen, and Huu-Duc Nguyen. "Parallel PageRank Computation Using GPUs." In: *Symposium on Information and Communication Technology*. ACM, 2012, p. 223 (page 5).

[51] Paul Erdős and Alfréd Rényi. "On the Evolution of Random Graphs." In: *Publications of the Mathematical Institute of the Hungarian Academy of Sciences* (1960), pp. 17–61 (pages 1, 3, 71).

[52] Ugo Erra, Sabrina Senatore, Fernando Minnella, and Giuseppe Caggianese. "Approximate TF-IDF Based on Topic Extraction From Massive Message Stream Using the GPU." In: *Information Sciences* 292 (2015), pp. 143–161 (pages 3, 104).

[53] Tanja Falkowski, Jörg Bartelheimer, and Myra Spiliopoulou. "Mining and Sisualizing the Evolution of Subgroups in Social Networks." In: *IEEE/WIC/ACM International Conference on Web Intelligence*. IEEE Computer Society. 2007, pp. 52–58 (page 1).

[54] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. "On Power-Law Relationships of the Internet Topology." In: *ACM SIGCOMM Computer Communication Review*. Vol. 29. 4. ACM Press, 1999, pp. 251–262 (pages 1, 3, 71).

[55] Bin Fan, David G. Andersen, and Michael Kaminsky. "MemC3: Compact and Concurrent Memcache With Dumber Caching and Smarter Hashing." In: *USENIX Conference on Networked Systems Design and Implementation*. Section 3. USENIX Association, 2013, pp. 371–385 (page 101).

[56] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. "SAP HANA Database." In: *ACM SIGMOD Record* 40.4 (2012), p. 45 (pages 6, 100).

[57] Brad Fitzpatrick. *Memcached - A Distributed Memory Object Caching System*. Online. 2009 (page 101).

[58] Gary William Flake, Robert E. Tarjan, and Kostas Tsioutsiouliklis. "Graph Clustering and Minimum Cut Trees." In: *Internet Mathematics* 1.4 (2004), pp. 385–408 (page 1).

[59] Ove Frank and David Strauss. "Markov Graphs." In: *Journal of the American Statistical Association* 81.395 (1986), p. 832 (page 71).

[60] King-Sun Fu. "A Step Towards Unification of Syntactic and Statistical Pattern Recognition." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8.3 (1986), pp. 398–404 (page 1).

[61] Zhisong Fu, Michael Personick, and Bryan Thompson. "MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs." In: *Workshop on GRAph Data-management Experiences and Systems*. ACM Press, 2014, pp. 1–6 (page 5).

[62] Corrado Gini. "Variabilità e Mutabilità." In: *Memorie di Metodologica Statistica* (1912) (page 91).

[63] Michelle Girvan and Mark E. J. Newman. "Community Structure in Social and Biological Networks." In: *National Academy of Sciences* 99.12 (2002), pp. 7821–7826 (pages 1, 3, 71).

[64] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Vol. 2. Addison-Wesley Longman Publishing Co., Inc., 1989, pp. xiii + 625 (page 31).

[65] *Graph 500*. 2010 (pages 4, 71, 79).

[66] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. "Fast Triangle Counting on the GPU." In: *Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 2014, pp. 1–8 (page 5).

[67] Daniel Gruhl, Ramanathan Guha, David Liben-Nowell, and Andrew S. Tomkins. "Information Diffusion Through Blogspace." In: *International Conference on World Wide Web*. May 2004. ACM. 2004, pp. 491–501 (page 1).

[68] Aric Hagberg, Daniel Schult, and Pieter Swart. "Exploring Network Structure, Dynamics, and Function Using NetworkX." In: *Python in Science Conference*. 2008, pp. 11–15 (pages 15, 41).

[69] Jiawei Han. *Mining Frequent Patterns, Associations, and Correlations: Basic Concepts and Methods*. Elsevier, 2006, pp. 4–5 (page 1).

[70] Huahai He and Ambuj K. Singh. "Graphs-At-A-Time: Query Language and Access Methods for Graph Databases." In: *Language*. Vol. L. Springer, 2008, pp. 405–418 (page 1).

[71] Lenwood S. Heath and Nidhi Parikh. "Generating Random Graphs With Tunable Clustering Coefficient Generating Random Graphs With Tunable Clustering Coefficient." In: *Physica A* 390.23–24 (2011), pp. 4577–4587 (page 1).

[72] Konrad Herbst, Cindy Fähnrich, Mariana L. Neves, and Matthieu-P. Schapranow. "Applying In-Memory Technology for Automatic Template Filling in the Clinical Domain." In: *CEUR Workshop Proceedings*. Vol. 1180. 2014, pp. 91–102 (pages 3, 6, 100, 104).

[73] Paul W. Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. "Stochastic Blockmodels: First Steps." In: *Social Networks* 5.2 (1983), pp. 109–137 (pages 6, 71, 94).

[74] Song Huang, Shucai Xiao, and Wu Chun Feng. "On the Energy Efficiency of Graphics Processing Units for Scientific Computing." In: *IEEE International Symposium on Parallel & Distributed Processing*. IEEE. IEEE, 2009, pp. 1–8 (page 3).

[75] Krzysztof Kaczmarski, Piotr Przymus, and Paweł Rzążewski. "Improving High-Performance GPU Graph Traversal With Compression." In: *Advances in Intelligent Systems and Computing*. Vol. 312. Springer, 2015, pp. 201–214 (page 5).

[76] Alfons Kemper and Thomas Neumann. "HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots." In: *International Conference on Data Engineering*. IEEE Computer Society, 2011, pp. 195–206 (pages 6, 100, 103).

[77] Robert Robest Kessl, Nilothpal Talukder, Pranay Anchuri, and Mohammed J. Zaki. "Parallel Graph Mining With GPUs." In: *International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*. 2014, pp. 1–16 (page 5).

[78] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. "FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs." In: *International Conference on Management of Data*. ACM Press, 2010, p. 339 (pages 101, 107, 114).

[79] Jon M. Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S. Tomkins. "The Web as a Graph: Measurements, Models, and Methods." In: *Annual International Conference on Computing and Combinatorics*. Tokyo, Japan: Springer-Verlag, 1999, pp. 1–17 (page 15).

[80] Tamara G. Kolda, Ali Pinar, Todd Plantenga, and Comandur Seshadhri. "A Scalable Generative Graph Model With Community Structure." In: *SIAM Journal on Scientific Computing* 36.5 (2014), pp. C424–C452 (pages 71, 91–94).

[81] Chris J. Kuhlman, Anil Kumar S. Vullikanti, and Sekharipuram Subramaniam Ravi. "Controlling Opinion Bias in Online Social Networks." In: *Annual ACM Web Science Conference* 57.10 (2012), pp. 165–174 (page 1).

[82] Solomon Kullback and Richard Leibler. "On Information and Sufficiency." In: *The Annals of Mathematical Statistics* (1951) (pages 63, 88).

[83] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tomkins, and Eli Upfal. "Stochastic Models for the Web Graph." In: *Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc, 2000, pp. 57–65 (pages 9, 15).

[84] Tarun Kumar, Parikshit Sondhi, and Ankush Mittal. "Parallelization of PageRank on Multicore Processors." In: *International Conference on Distributed Computing and Internet Technology*. Vol. 7154 LNCS. Springer Berlin Heidelberg, 2012, pp. 129–140 (page 5).

[85] Michihiro Kuramochi and George Karypis. "Frequent Subgraph Discovery." In: *IEEE International Conference on Data Mining*. IEEE Computer Society, 2001, pp. 313–320 (page 1).

[86] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. "What Is Twitter , a Social Network or a News Media?" In: *International World Wide Web Conference Committee*. 2010, pp. 1–10 (pages 1, 3, 71, 79, 94).

[87] Matthieu Latapy. "Main-Memory Triangle Computations for Very Large (Sparse (Power-Law)) Graphs." In: *Theoretical Computer Science* 407.1-3 (2008), pp. 458–473 (page 1).

[88] Vito Latora and Massimo Marchiori. "Vulnerability and Protection of Critical Infrastructures." In: *Physical Review E* 71.1 (2004), p. 4 (page 3).

[89] Mong Li Lee, Liang Huai Yang, Wynne Hsu, and Xia Yang. "XClust: Clustering XML Schemas for Effective Integration." In: *International Conference on Information and Knowledge Management*. Vol. 117543. 065. ACM. 2002, p. 292 (page 1).

[90] Tobin J. Lehman and Michael J. Carey. "A Study of Index Structures for Main Memory Database Management Systems." In: *Vldb*. Morgan Kaufmann Publishers Inc., 1986, pp. 294–303 (page 101).

[91] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases." In: *IEEE International Conference on Data Engineering*. IEEE, 2013, pp. 38–49 (pages 101–103, 106, 107, 110, 111, 114).

[92] Jure Leskovec. "Dynamics of Large Networks." PhD thesis. Carnegie Mellon University, 2008 (pages 1, 3, 71).

[93] Jure Leskovec. "Kronecker Graphs: An Approach to Modeling Networks." In: *Journal of Machine Learning Research* 11 (2010), pp. 985–1042 (pages 1, 3, 4, 71, 80).

[94] Jure Leskovec and Christos Faloutsos. "Scalable Modeling of Real Graphs Using Kronecker Multiplication." In: *International Conference on Machine Learning*. 2007, pp. 497–504 (pages 1, 3, 4, 71).

[95] Jure Leskovec and Eric Horvitz. "Planetary-Scale Views on a Large Instant-Messaging Network." In: *International Conference on World Wide Web*. ACM Press, 2008, p. 915 (page 4).

[96] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne Van-Briesen, and Natalie Glance. "Cost-Effective Outbreak Detection in Networks." In: *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2007, pp. 420–429 (page 1).

[97] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. "The Bw-Tree: A B-Tree for New Hardware Platforms." In: *IEEE International Conference on Data Engineering*. IEEE, 2013, pp. 302–313 (page 101).

[98] Xiaojie Lin, Rui Zhang, Zeyi Wen, Hongzhi Wang, and Jianzhong Qi. "Efficient Subgraph Matching Using GPUs." In: *Australasian Database Conference*. Vol. 8506 LNCS. Springer, 2014, pp. 74–85 (page 5).

[99] Max Lorenz. "Methods of Measuring the Concentration of Wealth." In: *Journal of the American Statistical Association* (1905) (page 91).

[100] David Luebke. "CUDA: Scalable Parallel Programming for High-Performance Scientific Computing." In: *IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. IEEE. 2008, pp. 836–838 (page 3).

[101] Benjamin Machta and Jonathan Machta. "Parallel Dynamics and Computational Complexity of Network Growth Models." In: *Physical Review E* 71.2 (2005), p. 26704 (page 12).

[102] Fredrik Manne and Tor Sorevik. "Optimal Partitioning of Sequences." In: *Journal of Algorithms* 19.2 (1995), pp. 235–249 (pages 44, 51, 82).

[103] Madhav V. Marathe and Anil Kumar S. Vullikanti. "Computational Epidemiology." In: *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 56.7 (2014), pp. 1969–1969 (page 1).

[104] Adam McLaughlin and David A. Bader. "Scalable and High Performance Betweenness Centrality on the GPU." In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. Vol. 2015-Janua. January. IEEE Press. 2014, pp. 572–583 (page 5).

[105] Joel C. Miller and Aric Hagberg. "Efficient Generation of Networks With Given Expected Degrees." In: *International Workshop on Algorithms and Models for the Web-Graph*. Vol. 6732 LNCS. 2011, pp. 115–126 (pages 5, 45, 47, 71, 73, 74, 78).

[106] Michael Molloy and Bruce Reed. "A Critical Point for Random Graphs With a Given Degree Sequence." In: *Random Structures & Algorithms* 6.2-3 (1995), pp. 161–180 (page 72).

[107] MusicBrainz. *MusicBrainz Database - MusicBrainz* (pages 108, 111).

[108] Mark E. J. Newman. "Finding Community Structure in Networks Using the Eigenvectors of Matrices." In: *Physical Review E* 74.3 (2006), p. 036104 (page 1).

[109] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. "Fast Random Graph Generation." In: *International Conference on Extending Database Technology*. 2011, p. 331 (page 5).

[110] Bjorn Olstad and Fredrik Manne. "Efficient Partitioning of Sequences." In: *IEEE Transactions on Computers* 44.11 (1995), pp. 1322–1326 (pages 44, 51, 82).

[111] OpenLibrary. *Open Library Data Dumps* (pages 108, 111).

[112] ORNL. *Titan*. 2012 (page 3).

[113] Guy Perrière and Manolo Gouy. "WWW-query: An On-Line Retrieval System for Biological Sequence Banks." In: *Biochimie* 78.5 (1996), pp. 364 –369 (page 109).

[114] Kalyan S. Perumalla. "Discrete-Event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs)." In: *Workshop on Principles of Advanced and Distributed Simulation*. Vol. 2006. 2006, pp. 74–81 (pages 3, 5).

[115] Kalyan S. Perumalla, Brandon G. Aaby, Srikanth B. Yoginath, and Sudip K. Seal. "GPU-based Real-Time Execution of Vehicular Mobility Models in Large-Scale Road Network Scenarios." In: *Workshop on Principles of Advanced and Distributed Simulation*. 2009, pp. 95–103 (pages 3, 5).

[116] Ali Pinar and Cevdet Aykanat. "Fast Optimal Load Balancing Algorithms for 1D Partitioning." In: *Journal of Parallel and Distributed Computing* 64.8 (2004), pp. 974–996 (pages 44, 51, 82).

[117] Ali Pinar, Comandur Seshadhri, and Tamara G. Kolda. "The Similarity Between Stochastic Kronecker and Chung-Lu Graph Models." In: *SIAM International Conference on Data Mining*. 2011, pp. 1071–1082 (pages 4, 71, 73, 74, 80, 92).

[118] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. "Replication, Load Balancing and Efficient Range Query Processing in DHTs." In: *International Conference on Extending Database Technology*. Springer. 2006, pp. 131–148 (page 91).

[119] Jun Rao and Kenneth A. Ross. "Cache Conscious Indexing for Decision-Support in Main Memory." In: *International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1998, pp. 78–89 (page 101).

[120] Jun Rao and Kenneth A. Ross. "Making B+- Trees Cache Conscious in Main Memory." In: *ACM SIGMOD Record*. Vol. 29. 2. 2000, pp. 475–486 (page 101).

[121] Matthew J. Rattigan, Marc Maier, and David Jensen. "Graph Clustering With Network Structure Indices." In: *Annual International Conference on Machine Learning*. ACM. 2006, pp. 783–790 (page 1).

[122] Steven Robbins. *RAM Is the New Disk...* Online. 2008 (page 100).

[123] Garry Robins, Pip Pattison, Yuval Kalish, and Dean Lusher. "An Introduction to Exponential Random Graph (p*) Models for Social Networks." In: *Social Networks* 29.2 (2007), pp. 173–191 (pages 1, 3, 71).

[124] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. "Log-Structured Memory for DRAM-based Storage." In: *USENIX Conference on File and Storage Technologies*. USENIX Association, 2014, pp. 1–16 (page 101).

[125] Arnon Rungsawang and Bundit Manaskasemsak. "Fast PageRank Computation on a GPU Cluster." In: *Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. 2012, pp. 450–456 (page 5).

[126] Karl Rupp, Philippe Tillet, Barry F. Smith, Tibor Grasser, and Ansgar Ju ngel. "A Note on the GPU Acceleration of Eigenvalue Computations." In: *AIP Conference Proceedings* 1558 (2013), pp. 1536–1539 (page 5).

[127] Pratha Sah, Lisa O. Singh, Aaron Clauset, and Shweta Bansal. "Exploring Community Structure in Biological Networks With Random Graphs." In: *BMC Bioinformatics* 15.1 (2014), p. 220 (page 1).

[128] Peter Sanders and Jesper Larsson Träff. "Parallel Prefix (Scan) Algorithms for MPI." In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, 2006, pp. 49–57 (page 53).

[129] Salvatore Sanfilippo and Pieter Noordhuis. *Redis*. Online. 2009 (page 101).

[130] Ahmet Erdem Sariyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. "Regularizing Graph Centrality Computations." In: *Journal of Parallel and Distributed Computing* 76 (2015), pp. 106–119 (page 5).

[131] Atish Das Sarma, Anisur Rahaman Molla, Gopal Pandurangan, and Eli Upfal. "Fast Distributed PageRank Computation." In: *Theoretical Computer Science* 561, Part B (2015), pp. 113 –121 (page 5).

[132] David Schwalb, Martin Faust, Jens Krueger, and Hasso Plattner. "Leveraging In-Memory Technology for Interactive Analyses of Point-Of-Sales Data." In: *IEEE International Conference on Data Engineering Workshops*. IEEE, 2014, pp. 97–102 (pages 3, 6, 100, 104).

[133] Hyunseok Seo, Jinwook Kim, and Min-Soo Kim. "GStream: A Graph Streaming Processing Method for Large-Scale Graphs on GPUs." In: *ACM SIGPLAN Notices*. Vol. 50. 8. ACM. 2015, pp. 253–254 (page 5).

[134] Comandur Seshadhri, Tamara G. Kolda, and Ali Pinar. "Community Structure and Scale-Free Collections of Erdős-Rényi Graphs." In: *Physical Review E* 85.5 (2012), p. 056109 (pages 6, 71, 91).

[135] *Shadowfax | Virginia Bioinformatics Institute*. 2015 (page 5).

[136] Yilun Shang. "Groupies in Random Bipartite Graphs." In: *Applicable Analysis and Discrete Mathematics* 4.2 (2010), pp. 278–283 (page 76).

[137] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. "Out-Of-Core GPU Memory Management for MapReduce-based Large-Scale Graph Processing." In: *IEEE International Conference on Cluster Computing*. IEEE. 2014, pp. 221–229 (page 5).

[138] Georgos Siganos, Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. "Power Laws and the As-Level Internet Topology." In: *IEEE/ACM Transactions on Networking* 11.4 (2003), pp. 514–524 (pages 1, 3, 71).

[139] Raffaele Solca, Azzam Haidar, Stanimire Tomov, Thomas C. Schulthess, and Jack Dongarra. "A Novel Hybrid CPU-GPU Generalized Eigensolver for Electronic Structure Calculations Based on Fine Grained Memory Aware Tasks." In: *SC Companion: High Performance Computing, Networking Storage and Analysis* 28.2 (2012), pp. 1338–1339 (page 5).

[140] Isabelle Stanton and Ali Pinar. "Constructing and Sampling Graphs With a Prescribed Joint Degree Distribution." In: *Journal of Experimental Algorithmics* 17.1 (2012), p. 3.1 (pages 6, 71).

[141] Matthias Steinbrecher and Joos-Hendrik Boese. "Real-Time Data Mining With In-Memory Database Technology." In: *Computational Intelligence in Intelligent Data Analysis*. Vol. 445. Springer Berlin Heidelberg, 2013, pp. 275–284 (pages 3, 6, 100, 104).

[142] Jimeng Sun, Christos Faloutsos, Spiros Papadimitriou, and Philip S. Yu. "GraphScope: Parameter-Free Mining of Large Time-Evolving Graphs." In: *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. ACM Press, 2007, p. 687 (page 1).

[143] Ning Tian, Longjiang Guo, Chunyu Ai, Meirui Ren, and Jinbao Li. "GPU Acceleration of Finding Maximum Eigenvalue of Positive Matrices." In: *International Conference on Algorithms and Architectures for Parallel Processing*. 2014, pp. 231–244 (page 5).

[144] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. "Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems." In: *Parallel Computing* 36.5-6 (2010), pp. 232–240 (page 5).

[145] Aparna S. Varde, Fabian M. Suchanek, Richi Nayak, and Pierre Senellart. "Fast Subgraph Matching on Large Graphs Using Graphics Processors." In: *International Conference on Database Systems for Advanced Applications*. Vol. 5463. Springer. 2009, pp. 784–788 (page 5).

[146] Richard L. Villars and Carl W. Olofson. "Big Data: What It Is and Why You Should Care." In: *IDC White Paper* (2011) (page 1).

[147] Vasily Volkov and James W. Demmel. *Using GPUs to Accelerate the Bisection Algorithm for Finding Eigenvalues of Symmetric Tridiagonal Matrices*. Tech. rep. UCB/EECS-2007-179. University of California at Berkeley, 2008 (page 5).

[148] Nikil Wale, Xia Ning, and George Karypis. "Trends in Chemical Graph Data Mining." In: *Managing and Mining Graph Data*. Vol. 40. Springer US, 2010, pp. 581–606 (page 1).

[149] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. "GDM: Device Memory Management for GPGPU Computing." In: *Sigmetrics*. ACM. 2014, p. 13 (page 104).

[150] Yang Wang, Deepayan Chakrabarti, Chenxi Wang, and Christos Faloutsos. "Epidemic Spreading in Real Networks: An Eigenvalue Viewpoint." In: *International Symposium on Reliable Distributed Systems*. IEEE Comput. Soc, 2003, pp. 25–34 (page 1).

[151] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. "Gunrock: A High-Performance Graph Processing Library on the GPU." In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM. 2015, pp. 265–266 (page 5).

[152] Yuchung J Wang and George Y Wong. "Stochastic Blockmodels for Directed Graphs." In: *Journal of the American Statistical Association* 82.397 (1987), p. 8 (page 94).

[153] Duncan J. Watts and Steven H. Strogatz. "Collective Dynamics of 'Small-World' Networks." In: *Nature* 393.6684 (1998), pp. 440–2 (pages 1, 3, 4, 71).

[154] Jim Webber. "A Programmatic Introduction to Neo4J." In: *Annual Conference on Systems, Programming, and Applications: Software for Humanity*. ACM. 2012, pp. 217–218 (page 5).

[155] Benjamin Welton and Barton P. Miller. "The Anatomy of Mr. Scan: A Dissection of Performance of an Extreme Scale GPU-Based Clustering Algorithm." In: *Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. IEEE. 2014, pp. 54–60 (page 5).

[156] Douglas Brent West. *Introduction to Graph Theory*. Upper Saddle River (N. J.): Prentice Hall, 2001 (page 7).

[157] David W. Williams, Jun Huan, and Wei Wang. "Graph Database Indexing Using Structured Graph Decomposition." In: *IEEE International Conference on Data Engineering*. IEEE. IEEE, 2007, pp. 976–985 (page 6).

[158] Robin J. Wilson. *Introduction to Graph Theory*. New York, NY, USA: John Wiley & Sons, Inc., 1986 (page 7).

[159] Ren Wu, Bin Zhang, and Meichun Hsu. "GPU-Accelerated Large Scale Analytics." In: *Development* HPL-2009-38 (2009), p. 10 (page 101).

[160] Bo Yang, Kai Lu, Ying Hui Gao, Xiao Ping Wang, and Kai Xu. "GPU Acceleration of Subgraph Isomorphism Search in Large Scale Graph." In: *Journal of Central South University* 22.6 (2015), pp. 2238–2249 (page 5).

[161] Jaewon Yang and Jure Leskovec. "Defining and Evaluating Network Communities Based on Ground-Truth." In: *Knowledge and Information Systems*. Vol. 42. 1. 2015, pp. 181–213 (pages 63, 71, 79).

[162] Jaewon Yang and Jure Leskovec. "Patterns of Temporal Variation in Online Media." In: *ACM International Conference on Web Search and Data Mining*. Vol. 468. Hong Kong, China: ACM, 2011, pp. 177–186 (page 63).

[163] Andy Yoo and Keith Henderson. "Parallel Generation of Massive Scale-Free Graphs." In: *Computing Research Repository* abs/1003.3 (2010), pp. 1–13 (pages 4, 12).

[164] Tao Zhang, Jingjie Zhang, Wei Shu, Min-You Wu, and Xiaoyao Liang. "Efficient Graph Computation on Hybrid CPU and GPU Systems." In: *The Journal of Supercomputing*. Vol. 71. 4. 2015, pp. 1563–1586 (pages 5, 6, 100, 107, 114).

[165] Jianlong Zhong and Bingsheng He. "Towards GPU-accelerated Large-Scale Graph Processing in the Cloud." In: *International Conference on Cloud Computing Technology and Science*. Vol. 1. IEEE. 2013, pp. 9–16 (page 5).