Parallel Algorithms for Switching Edges and Generating Random Graphs from Given Degree Sequences using HPC Platforms

Md Hasanuzzaman Bhuiyan

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

Doctor of Philosophy
in
Computer Science and Applications

Madhav V. Marathe, Chair
Maleq Khan, Co-Chair
Anil K. Vullikanti, Member
Lenwood S. Heath, Member
S. S. Ravi, Member

August 11, 2017
Blacksburg, Virginia

Keywords: Network Science, Parallel Algorithms, High Performance Computing, Edge Switch, Random Networks

Parallel Algorithms for Switching Edges and Generating Random Graphs from Given Degree Sequences using HPC Platforms

Md Hasanuzzaman Bhuiyan

(ABSTRACT)

Networks (or graphs) are an effective abstraction for representing many real-world complex systems. Analyzing various structural properties of and dynamics on such networks reveal valuable insights about the behavior of such systems. In today's data-rich world, we are deluged by the massive amount of heterogeneous data from various sources, such as the web, infrastructure, and online social media. Analyzing this huge amount of data may take a prohibitively long time and even may not fit into the main memory of a single processing unit, thus motivating the necessity of efficient parallel algorithms in various high-performance computing (HPC) platforms. In this dissertation, we present distributed and shared memory parallel algorithms for some important network analytic problems.

First, we present distributed memory parallel algorithms for switching edges in a network. Edge switch is an operation on a network, where two edges are selected randomly, and one of their end vertices are swapped with each other. This operation is repeated either a given number of times or until a specified criterion is satisfied. It has diverse real-world applications such as in generating simple random networks with a given degree sequence and in modeling and studying various dynamic networks. One of the steps in our edge switch algorithm requires generating multinomial random variables in parallel. We also present the first non-trivial parallel algorithm for generating multinomial random variables.

Next, we present efficient algorithms for assortative edge switch in a labeled network. Assuming each vertex has a label, an assortative edge switch operation imposes an extra constraint, i.e., two edges are randomly selected and one of their end vertices are swapped with each other if the labels of the end vertices of the edges remain the same as before. It can be used to study the effect of the network structural properties on dynamics over a network. Although the problem of assortative edge switch seems to be similar to that of (regular) edge switch, the constraint on the vertex labels in assortative edge switch leads to a new difficulty, which needs to be addressed by an entirely new algorithmic approach. We first present a novel sequential algorithm for assortative edge switch; then we present an efficient distributed memory parallel algorithm based on our sequential algorithm.

Finally, we present efficient shared memory parallel algorithms for generating random networks with exact given degree sequence using a direct graph construction method, which involves computing a candidate list for creating an edge incident on a vertex using the Erdős-Gallai characterization and then randomly creating the edges from the candidates.

Parallel Algorithms for Switching Edges and Generating Random Graphs from Given Degree Sequences using HPC Platforms

Md Hasanuzzaman Bhuiyan

(GENERAL AUDIENCE ABSTRACT)

Network analysis has become a popular topic in many disciplines including social sciences, epidemiology, biology, and business as it provides valuable insights about many real-world systems represented as networks. The recent advancement of science and technology has resulted in a massive growth of such networks, and mining and processing such massive networks poses significant challenges, which can be addressed by various high-performance computing (HPC) platforms. In this dissertation, we present parallel algorithms for a few network analytic problems using HPC platforms.

Random networks are widely used for modeling many complex real-world systems such as the Internet, biological, social, and infrastructure networks. Most prior work on generating random graphs involves sequential algorithms, and they can be broadly categorized in two classes: (***i***) edge switching and (***ii***) stub-matching. We present parallel algorithms for generating random graphs using both the edge switching and stub-matching methods. Our parallel algorithms for switching edges can generate random networks with billions of edges in a few minutes with 1024 processors. We have studied several load balancing methods to equally distribute workload among the processors to achieve the best performance. The parallel algorithm for generating random graphs using the stub-matching method also shows good speedup for medium-sized networks. We believe the proposed parallel algorithms will prove useful in analyzing and mining of emerging networks.

Dedication

Dedicated to my mother, Marium Begum, and the loving memory of my father, Md Shamsuzzaman Bhuiyan, for their selfless love, support, and encouragement.

Acknowledgments

It was an amazing, humbling, and enriching journey throughout my Ph.D. life at Virginia Tech. I am grateful to have many wonderful people in my professional and personal life, who helped me to complete this dissertation. I express my deepest appreciation and acknowledgment for their support, encouragement, and love.

First of all, I would like to express my sincerest gratitude to my advisors Drs. Madhav Marathe and Maleq Khan, for their continuous support, constant encouragement, insightful advice, and persistent effort in guiding and advising me throughout my graduate study. I am especially grateful to Dr. Madhav Marathe for introducing me to network science and big data analytics, providing me thoughtful suggestions to improve my presentation and writing skills, giving invaluable career advice through sharing his professional experience, and providing constructive feedback on my research problems. He always managed time from his busy schedule, including even when he was traveling to attend a conference or a meeting, to discuss any problems and help them sort out. I thank him for continuously inspiring and motivating me during the hardest time in my Ph.D. life. I am indebted to him for providing me the opportunity to work in his lab and funding me throughout my entire graduate study. I am immensely honored to have an excellent advisor like him.

I would like to thank Dr. Maleq Khan for introducing me to the research areas of Graph Theory and High-Performance and Distributed Computing through sharing his remarkable expertise in these fields. He spent a substantial amount of time by his enthusiastic and active participation in developing the research ideas in this dissertation. He mentored me in every aspect of my academic endeavor, including improving my writing, problem solving, presentation, and analytical skills. He has been a responsible mentor, concerning friend, and rigorous researcher. He has always been accessible when I needed his guidance. I could not have completed this dissertation without his helpful advice and continuous support. I am very grateful to him for his invaluable contribution in my professional and personal growth. I am very fortunate to have an excellent mentor and advisor like him.

I would like to thank the other committee members, Drs. Lenwood S. Heath, Anil Vullikanti,

and S. S. Ravi for their valuable feedback to improve this dissertation. I am indebted to Dr. Heath for carefully and thoroughly reviewing the draft of this dissertation and providing constructive feedback. I am thankful to Dr. Vullikanti for his insightful comments and suggestions in one of our research papers. I am also grateful to Dr. Ravi for offering me useful insights about analytical approaches for several problems.

I am specially thankful to Drs. Madhav Marathe, Maleq Khan, and S. S. Ravi for their advice and guidance during my job search. I also thank Drs. Madhav Marathe, Maleq Khan, and Chris Kuhlman for writing recommendation letters for me. I also thank the anonymous reviewers of the papers [16–19], which are the basis of this dissertation. Their suggestions and scholarly comments helped greatly in improving the presentation of the papers. I also acknowledge each of my co-authors in the research papers resulted from this dissertation.

I acknowledge my lab-mates in the Network Dynamics and Simulation Science Lab (NDSSL) for the wonderful interactions and discussions. I am grateful to the Bangladeshi community at Virginia Tech for their support, friendship, and for making my stay a pleasant one. I also thank all my friends at home and abroad for always believing in me and inspiring me in my tough times.

Last but not the least, my heartfelt gratitude goes to my family for their endless and selfless love, support, encouragement, and motivation throughout my life. I am thankful to my wonderful siblings, Faria Tasnim and Sumaiya Zaman, for their selfless support, enthusiasm, and inspiration, and for continuously taking care of our family, which greatly helped me to focus on my Ph.D. research. I would like to particularly mention my father, who taught me the math and science related subjects in my early stages of education and greatly helped me to build a strong foundation in those subjects, which served as a backbone of my higher studies. He continuously inspired me to pursue my major in Computer Science during my undergraduate study and highly motivated me to further strengthen my expertise through a Ph.D program. Without his support, inspiration, and blessing, I would not have been able to embark this place. I would also like to express my sincerest gratitude to my mother for teaching me to practice patience and resilience in tough times. She has been a role model to me for always being cool, conscientious, and positive even in an overwhelmingly frustrating and adversarial environment. Thank you, *Ammu* and *Abbu*, for making me who I am, and I miss you a lot, *Abbu*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many complex and real-world systems can be represented through networks (graphs) for further analyses. Analyzing the dynamics on and structural properties of networks became a central topic of research as they provide valuable insights into real-world systems [30]. Application of network analyses spans a wide range of areas including biology [64], epidemiology [98], social sciences [49, 69], economics [42], business, and finance [9]. The recent growth of real-world data due to the rapid progression of science and technology poses significant challenges towards efficient processing of massive networks. For instance, social media and web companies like Yahoo, Facebook, Google, and Twitter have to analyze networks having billions to trillions of edges for diverse real-world applications such as content ranking and recommendation systems [22, 23, 30, 31]. Dealing with these huge amounts of data efficiently and effectively motivates the need for parallel computing. In addition to the efficient computing issue, the massive networks may not fit into the memory of a single computing machine, thus posing another difficulty of dealing with larger memory requirements. These challenges are discussed in detail in a recent study [30], where Facebook shared their experiences about facing these challenges in the production level while processing networks having trillions of edges.

To address these issues, various high-performance computing (HPC) platforms, such as the distributed memory, shared memory, and the graphics processing unit (GPU) based systems can be a suitable choice. In the distributed memory parallel platforms, the input data and tasks are typically partitioned and distributed among the processors. Examples of such computation models include MPI [53–55, 105], MapReduce [38], Hadoop [113], Pregel [79], Giraph [7, 30], and Spark [8, 119]. Although the processors do not have any shared memory among them in a distributed memory parallel system, they can communicate and exchange data with each other through message passing, which is

useful to deal with the computational dependencies in the parallel algorithms. Distributed memory parallel algorithms often require a large amount of implementation effort to achieve efficiency through a careful design; yet it is worth the investment to design such algorithms since the same algorithm can scale from thousands to millions of processors and therefore can efficiently deal with massive-scale data, which is the ultimate goal of parallel computation. Thus distributed memory parallel systems offer a potential solution to the scalability related challenges associated with the processing of big data. In a shared memory parallel system [4, 65], all the processors share the same memory and can pass data between processors by reading and writing to the global address space asynchronously, thus avoiding the complexity and synchronization involved with the message passing in the distributed memory parallel systems. The recent advancement of science and technology made shared memory parallel systems with very large memory easily affordable and very popular. It is also easy to write code in a shared memory parallel system. However, asynchronous concurrent memory access issues may lead to race conditions, which are dealt with via complex mechanisms like locks and semaphores. OpenMP [28, 29, 36] and Pthreads [72, 90] are among the popular shared memory parallel programming platforms. On the other hand, GPU-based parallelism [89, 94, 96, 101] offers data-level parallelism, which can be faster for some special classes of problems, but it is often hard to program in this paradigm for many graph algorithms. Examples of this paradigm include OpenCL [83, 109] and CUDA [91, 92].

In this dissertation, we present distributed and shared memory parallel algorithms for several important network analytic problems. We present algorithms developed in the message passing interface (MPI) [53–55, 105] and OpenMP [28, 29, 36], which are very popular, widely used, and general purpose distributed and shared memory parallel programming platforms, respectively. In both MPI and OpenMP, an algorithm needs to be implemented from scratch by specifying the details of data and task partitioning or sharing, data structures, synchronization among the processors, and terminating condition. On the other hand, these lower level complexities and implementation details are hidden from the developers in more abstract parallel computing platforms like Pregel [79], Giraph [7, 30], and GraphX [51]. Although it requires a large amount of implementation effort in MPI and OpenMP compared to the abstract platforms, they offer the flexibility of designing efficient algorithms and applying various optimization techniques, which consequently often lead towards a faster execution of the parallel algorithms. We develop our parallel algorithms in MPI and OpenMP due to their superior performance with optimized code [26, 102].

First, we present MPI-based distributed memory parallel algorithms for switching edges in a network. Edge switch, also known as edge swap, edge rewiring, and edge flip, is an operation on a network where two edges are selected randomly and the end vertices

are swapped with each other. There are many variations [34, 45, 48, 61, 63, 97, 106, 107, 110] of this problem. In the most commonly used edge switch operation, two edges (u_1, v_1) and (u_2, v_2) are selected at random and replaced by (u_1, v_2) and (u_2, v_1). It is repeated either a given number of times or until a specified criterion is satisfied. Note that this operation preserves the degree of each vertex. It has many diverse real-world applications, such as in generating random networks from a given degree sequence [34, 45, 48], in studying and modeling various dynamic networks [45], and in studying the dynamics over a network such as disease dynamics over a social contact network [44]. The Havel-Hakimi method [57] can generate a deterministic network from a given degree sequence. By combining the Havel-Hakimi method with the edge switch process, a random network can be generated with a prescribed degree sequence [34, 45, 48]. The sequential algorithm for edge switch is quite straightforward. An edge switch operation selects two edges uniformly at random and swaps the end vertices if the resulting network remains simple (i.e., no loops or parallel edges). However, parallelizing the simple edge switch operation seems to be a non-trivial problem for the following reasons: a) the sequential algorithm selects and switches a sequence of pairs of edges, one pair after another, whereas in the parallel algorithm, multiple pairs of edges are selected and switched simultaneously by different processors, thus posing a significant challenge in designing a parallel algorithm by maintaining an equivalent stochastic process as that of the sequential one, and b) the requirement of keeping the network simple entails complex synchronization and communication among the processors, which in turn makes it difficult to achieve a good speedup by parallelization. Addressing these issues requires carefully designed data structures, complex synchronization, and communication among the processors. We also study several partitioning schemes in conjunction with the algorithms and present the trade-offs. A maximum speedup of 115 is achieved with 800 processors on a network with 2 million vertices and 52 million edges. One of the steps in the parallel algorithms requires generating multinomial random variables in parallel. We also present the first non-trivial parallel algorithm for generating multinomial random variables, which achieves a speedup of 925 with 1024 processors. Moreover, we present a shared memory parallel algorithm for 1-Flipper, which is an edge switch operation maintaining the connectivity of the given graph. The parallel algorithm achieves a speedup of 15 with 32 cores.

Next, we present efficient algorithms for assortative edge switch in a labeled network, where a vertex u has an associated label $L(u)$. Such labels can be discrete characteristics (e.g., language, race, gender in social networks) or scalar properties (e.g., age, degree) of the vertices. An assortative edge switch operation imposes an extra constraint on the vertex labels, where two edges (u_1, v_1) and (u_2, v_2) are randomly selected, and they are replaced with edges (u_1, v_2) and (u_2, v_1), if $L(u_1) = L(u_2)$, $L(v_1) = L(v_2)$, and the resulting network remains simple. For each vertex u, the degree as well as the distribution of the labels of

adjacent vertices of u, remain invariant under an assortative edge switch process. It has many important applications in graph theory and network analysis. Examples include studying the mixing pattern and dynamic behavior of social networks, modeling and analyzing dynamic networks, and generating random networks from a given network by preserving the same degree sequence and assortative coefficient (a measure of the tendency of the vertices to associate with similar or dissimilar vertices) [27, 84, 85, 88]. Recall that the (regular) edge switch algorithm selects two edges randomly from the entire network, irrespective of the end vertex labels, which can result in many failed attempts for an assortative edge switch operation due to dissatisfying the constraint on vertex labels. As a result, we need a completely new algorithmic approach. We first present an efficient sequential algorithm for assortative edge switch using a novel algorithmic approach. Then we present an efficient MPI-based distributed memory parallel algorithm, which allows working more efficiently with very large networks. To the best of our knowledge, both our sequential and parallel algorithms are the first non-trivial efficient algorithms for this problem. The dependencies among successive assortative edge switch operations, the requirement of maintaining the same assortative coefficient and keeping the network simple during the assortative edge switch process, and designing a good partitioning scheme to achieve a well-balanced computation load among the processors pose non-trivial challenges in designing a parallel algorithm. Our parallel algorithm addresses all these challenges, scales well to a large number of processors, and provides a speedup of 68–772 with 1024 processors for a wide variety of synthetic and real-world networks with diverse distributions of the labels of the vertices.

Finally, we present an OpenMP-based shared memory parallel algorithm for generating a random network with a prescribed degree sequence. There has been a significant increase of interest in generating random networks due to its popularity in modeling and simulating many complex real-world systems [21]. Degree sequence is one of the most important aspects of these systems [15, 58, 75]. Random networks with a given degree sequence can capture many characteristics like dependent edges and non-binomial degrees that are absent in many classical models such as the Erdős-Rényi [24] network model [20]. In addition, they are also widely used in uniform sampling of random networks as well as in counting the number of networks having the same degree sequence [14, 20, 33, 39]. Blitzstein et al. [20] presented such a sequential importance sampling algorithm for generating random networks with a prescribed degree sequence. This approach first creates all the edges incident on the vertex having the minimum degree in the sequence, then moves to the next vertex having the minimum remaining degree to create its incident edges, and so on. To create an edge incident on a vertex u, a candidate list C is computed using the Erdős-Gallai characterization [20] such that after adding an edge by connecting u to any candidate vertex v from the list C, the residual degree sequence remains graphical (i.e., there exists a

graph with that degree sequence) and the graph remains simple. Then an edge (u, v) is created by choosing v from the candidate list C with the probability proportional to the remaining degree of v at the time of the edge creation. The advantage of this method is that it never gets stuck or creates loops or parallel edges like many other methods (e.g., the configuration model [86]) do, which is often dealt with by restarting the graph generation process. Moreover, the algorithm generates every possible random network for a given degree sequence with a positive probability, and the distribution of the generated networks can be estimated as well. A parallel algorithm for generating random networks with expected degree sequence has been presented in [5]. However, there is no existing parallel algorithm for generating random network following an exact degree sequence. We present an efficient shared memory parallel algorithm for the problem based on the sequential algorithm presented in [20], which achieves a maximum speedup of 20.5 with 32 cores. The dependencies among assigning edges to vertices in a particular order to ensure the algorithm never gets stuck, the requirement of keeping the graph simple, maintaining an exact stochastic process as that of the algorithm in [20], and concurrent writing by multiple cores in the shared memory lead to significant challenges in designing an efficient parallel algorithm. One of the steps of the algorithm requires checking the graphicality of a given degree sequence using the Erdős-Gallai characterization in parallel. We present here a non-trivial parallel algorithm for checking the Erdős-Gallai characterization, which achieves a speedup of 23 using 32 cores. Moreover, we present a comparative study of several structural properties of the random networks generated by our parallel algorithm with that of the real-world networks and random networks generated by the edge switching method.

The remainder of the dissertation is organized as follows. We define some basic terminologies used in the dissertation in Chapter 2. We present the distributed memory parallel algorithms for switching edges and generating multinomial random variables and the shared memory parallel algorithm for 1-Flipper in Chapter 3. The sequential and distributed memory parallel algorithms for assortative edge switch are discussed in Chapter 4. The shared memory parallel algorithms for generating a random network with a prescribed degree sequence and checking the Erdős-Gallai characterization are presented in Chapter 5. Finally, we conclude in Chapter 6.

Chapter 2

Preliminaries

In this chapter, we define some basic terminologies in network (or graph) and probability theory, which are used throughout the remainder of the dissertation. For convenience, we define some terminologies later as they are needed. Interested readers are referred to [25, 41, 46, 112] for detailed texts on network and probability theory.

2.1 Network

A *network* or *graph* $G = (V, E)$ is a mathematical structure consisting a set of *vertices* or *nodes* V and a set of *edges* E. A vertex represents an entity such as a person in a social contact network, and an edge represents a relationship between two entities such as a friendship between two persons in a social contact network. If E is a set of unordered pairs of vertices $\{u, v\}$, where u and v are vertices in V, then the network is known as an *undirected network*. An edge between vertices u and v is denoted by (u, v) and is said to be *incident* on u and v. For an edge $(u, v) \in E$, we say u and v are *neighbors* of each other. The *adjacency list* of a vertex u contains all the neighbors of u and is denoted by $N(u)$, i.e., $N(u) = \{v \in V | (u, v) \in E\}$. The *degree* of u is $d_u = |N(u)|$.

2.2 Simple Network

A *self-loop* is an edge from a vertex to itself. *Parallel edges* are two or more edges connecting the same pair of vertices. A *simple network* is an undirected network with no self-loops or parallel edges. Throughout this dissertation, we consider simple networks.

2.3 Degree Distribution

Let n_k denote the number of vertices of degree k in an undirected network $G = (V, E)$. Then the *degree distribution* DD is defined by a sequence

$$DD = \{P_k \mid k \geq 0\}, \tag{2.1}$$

where P_k is the fraction of vertices of degree k, i.e.,

$$P_k = \frac{n_k}{|V|}. \tag{2.2}$$

In other words, if a vertex is chosen uniformly at random from G, then the probability of the vertex having degree k is P_k. Here, P_k is normalized so that $\sum_k P_k = 1$.

2.4 Connected Component

A *connected component* of an undirected network $G = (V, E)$ is a maximal subgraph $S \subseteq G$, where any two vertices in S are reachable from each other, i.e., there is a path between any two vertices in S. A network can have one or more connected components. A network in which every two vertices are reachable from each other has only one connected component consisting of the entire network and the network is connected.

2.5 Triangle

In an undirected network $G = (V, E)$, if three vertices u, v, and w are connected by edges $(u, v), (v, w)$, and (w, u), then these three edges form a *triangle*. More formally, the number of triangles $\Delta(v)$ incident on a vertex v is

$$\Delta(v) = |\{(u, w) \in E \mid u, w \in N(v)\}|. \tag{2.3}$$

As each triangle appears on each of the three nodes it is incident on, the number of triangles $\Delta(G)$ in G is

$$\Delta(G) = \frac{1}{3} \sum_{v \in V} \Delta(v). \tag{2.4}$$

2.6 Clustering Coefficient

The *local clustering coefficient* $C(v)$ of a vertex v in an undirected network $G = (V, E)$ is the ratio of the triangles incident on v to the maximum number of such triangles, i.e.,

$$C(v) = \begin{cases} \frac{2\ \Delta(v)}{d_v(d_v-1)} & \text{if } d_v \geq 2, \\ 0 & \text{otherwise.} \end{cases} \tag{2.5}$$

The *degree-dependent clustering coefficient* $\overline{C}(k)$ is the average of the local clustering coefficient of all vertices of degree k, i.e.,

$$\overline{C}(k) = \begin{cases} \frac{2\ \Delta_k}{n_k k(k-1)} & \text{if } k \geq 2, \\ 0 & \text{otherwise,} \end{cases} \tag{2.6}$$

where n_k is the number of vertices of degree k and Δ_k is the number of triangles incident on vertices of degree k, i.e.,

$$\Delta_k = \sum_{v:d_v=k} \Delta(v). \tag{2.7}$$

The *average clustering coefficient* $C(G)$ of a network is the average of the local clustering coefficient of all of its vertices, i.e.,

$$C(G) = \frac{1}{n} \sum_{v \in V} C(v). \tag{2.8}$$

2.7 Shortest Path Distance

The *distance* $d(u, v)$ between two vertices u and v in an undirected network $G = (V, E)$ is the number of edges in a shortest path connecting them if u and v are reachable from each other, i.e., there is a path between u and v. Note that there may be no path connecting two vertices if they belong to different components, or there may be multiple shortest paths connecting u and v. The *average shortest path distance* or *length* $d_{avg}(G)$ in G is the average of the distances between all pairs of vertices which are reachable from each other, i.e.,

$$d_{avg}(G) = \frac{\sum_{u,v \in V | u \rightsquigarrow v} d(u, v)}{|\{(u, v) \mid u, v \in V \wedge u \rightsquigarrow v\}|}, \tag{2.9}$$

where $u \rightsquigarrow v$ means there is a path between u and v.

2.8 Assortativity Coefficient

Assortativity or the *assortative coefficient* of an undirected network $G = (V, E)$ measures the tendency of vertices to associate with similar or dissimilar vertices [85, 88]. In other words, assortativity measures the correlation of vertices based on vertex labels such as degree of a vertex or age of a person. Let e_{xy} be the fraction of all edges in G that connect the vertices with values x and y for any scalar properties such as age. Then it satisfies

$$\sum_{x,y} e_{xy} = 1, \tag{2.10}$$

$$\sum_{y} e_{xy} = a_x, \tag{2.11}$$

$$\sum_{x} e_{xy} = b_y, \tag{2.12}$$

where a_x and b_y are the fraction of edges that start and end at vertices with values x and y, respectively. The assortativity coefficient r for mixing by a scalar property of a vertex is then defined as:

$$r = \frac{\sum_{x,y} xy(e_{xy} - a_x b_y)}{\sigma_a \sigma_b}, \tag{2.13}$$

where σ_a and σ_b are the standard deviations of the distributions a_x and b_y, respectively. For the practical purpose of measuring the degree-based assortativity coefficient in a network, it can be redefined as:

$$r = \frac{m^{-1} \sum_i j_i k_i - \left[m^{-1} \sum_i \frac{1}{2}(j_i + k_i)\right]^2}{m^{-1} \sum_i \frac{1}{2}(j_i^2 + k_i^2) - \left[m^{-1} \sum_i \frac{1}{2}(j_i + k_i)\right]^2}, \tag{2.14}$$

where m is the number of edges in G and j_i, k_i are the degrees of the end vertices of the i-th edge with $i = 1, 2, \ldots, m$. The value of the assortativity coefficient r lies in the range $[-1, 1]$. The values $r = 1$ and $r = -1$ indicate perfect assortativity and disassortativity, respectively. Considering the degree of a vertex as its label, for example, social (e.g., friendship) networks exhibit positive assortativity ($r > 0$)—more association among similar types of vertices—whereas technological (e.g., Internet) and biological (e.g., food-web) networks show negative assortativity ($r < 0$)—more association among dissimilar types of vertices [84].

2.9 Binomial Distribution

Suppose that N independent trials are to be performed, where each trial results in a success with probability q and in a failure with probability $(1 - q)$. If X represents the number of successes that occur among N trials, then X is said to be a binomial random variable. The distribution of X is a binomial distribution with parameters N and q and denoted as follows:

$$X \sim \mathcal{B}(N, q). \tag{2.15}$$

The probability of getting exactly i successes in N trials is given by

$$\Pr\{X = i\} = \binom{N}{i} q^i (1 - q)^{N-i}. \tag{2.16}$$

2.10 Multinomial Distribution

Let N be the number of independent trials to be performed, where each trial has ℓ possible outcomes $0, 1, \ldots, \ell - 1$ with probability $q_0, q_1, \ldots, q_{\ell-1}$ respectively, such that $q_i \geq 0$ for $0 \leq i \leq \ell - 1$ and $\sum_i q_i = 1$. Let the random variable X_i indicate the number of times the outcome i appears among N independent trials. Then $X = \langle X_0, X_1, \ldots, X_{\ell-1} \rangle$ has a multinomial distribution with parameters $N, q_0, q_1, \ldots, q_{\ell-1}$, and denoted as follows:

$$\langle X_0, X_1, \ldots, X_{\ell-1} \rangle \sim \mathcal{M}(N, q_0, q_1, \ldots, q_{\ell-1}). \tag{2.17}$$

Chapter 3

Parallel Algorithms for Switching Edges in Heterogeneous Networks

3.1 Introduction

Edge switch, also known as edge swap, edge flip, edge shuffle, and edge rewiring, is an operation that swaps the end vertices of the edges in a network. Many variations of this problem have been studied [34, 45, 48, 61, 63, 97, 106, 107, 110] with diverse real-world applications. In the most commonly used edge switch operation, two randomly selected edges (a, b) and (c, d) are replaced with edges (a, d) and (c, b) respectively, i.e., the end vertices of the selected edges are swapped with each other. This operation is repeated either a given number of times or until a specified criterion is satisfied. It is easy to see that an edge switch operation preserves the degree of each vertex.

This problem has many important applications. It can be used in generating random networks with a given degree sequence. There has been a lot of work on random graph generation, because of the popularity of network models in diverse applications. Most of the prior work involves sequential algorithms, and much of it is restricted to regular graphs; we briefly summarize the main approaches here. A popular method for random graph generation is the *configuration model* (also referred to as the "pairing" model) [20, 86, 114], which involves creating stubs for vertices, choosing pairs of stubs at random, and then connecting them by edges. Unfortunately, this leads to parallel edges, unless the degrees are very small. This basic approach has been modified in various ways to avoid parallel edges in the case of regular graphs [67, 108, 114] (see [20] for a good discussion). Blitzstein et al. [20] give a simple algorithm for generating random graphs with a given degree sequence using sequential importance sampling, based on the Erdős-Gallai characterization.

By using the Havel-Hakimi method [57], a network can be generated following a given degree sequence. Since it is deterministic, this method generates the same network each time it is run with the same degree sequence whereas there can be many different networks with the same degree distribution. However, edge switch can be combined with the Havel-Hakimi method to generate a random network with a given degree sequence [34, 45, 48]. Once a network is generated using the Havel-Hakimi method, by randomly switching the edges we can generate a random network with the same degree sequence. The mixing time was shown to be bounded by a large polynomial by Cooper et al. [34] and extended by Feder et al. [45] to variants of the edge switch process.

Edge switch is also used in modeling and studying various dynamic networks such as peer-to-peer networks [45]. Other applications of edge switch include the generation of randomly labeled bipartite graphs with a given degree sequence [63], independent realizations of graphs with a prescribed joint degree distribution using a Markov chain Monte Carlo approach [97], and studying the sensitivity of network topology on dynamics over a network such as disease dynamics over a social contact network [44].

Edge switch can be paired with additional constraints such as imposing a connectivity requirement, allowing or not allowing parallel edges and loops. NetworkX [56] has a sequential implementation of edge switch that does not allow parallel edges, but allows loops, and provides the option of imposing connectivity constraints on the graph. A connectivity constraint requires a graph to remain connected after an edge switch operation. Some theoretical studies of edge switch for restricted graph classes can be found in the literature, such as the study of mixing time of the Markov chain introduced by this operation [34, 48]. However, no effort was given to design parallel algorithms for switching edges in a graph. For smaller graphs, sequential implementation of edge switch suffices, but this may not work for massive networks for the following reasons: (*i*) a massive network with billions of edges simply may not fit in the memory of a single computing machine and (*ii*) a sequential algorithm may take a prohibitively long time. These issues can be addressed by a distributed memory parallel algorithm where the network is partitioned into disjoint subsets of edges or blocks and each processor contains one such block.

In this chapter, we present distributed memory parallel algorithms for switching edges in massive graphs with the constraint that the graph remains simple. The dependencies among successive edge switch operations and the requirement of keeping the graph simple lead to significant challenges in designing a parallel algorithm. Dealing with these requires complex synchronization and communication among the processors, which in turn makes it challenging to gain any speedup by parallelization. The performance of the algorithms also depends on the partitioning of the graph. We study several partitioning schemes in conjunction with the algorithms and present their respective trade-offs. A harmonic mean

speedup (compared to the sequential algorithm's runtime) of 73.25 is achieved on eight different networks with 1024 processors. The algorithms require generating multinomial random variables in parallel, which is also a non-trivial problem. To the best of our knowledge, there is no existing parallel algorithm for the problem, and we present here a novel parallel algorithm for generating multinomial random variables, which achieves a speedup of 925 using 1024 processors. Finally, we present a shared memory parallel algorithm for 1-Flipper, i.e., switching edges by imposing the connectivity constraint, which achieves a speedup of 15 using 32 cores.

The rest of the chapter is organized as follows. Section 3.2 describes the preliminaries and notations used in the chapter. The edge switch problem and the sequential algorithm are briefly explained in Section 3.3. We present our main parallel algorithms for switching edges in Section 3.4. The parallel algorithm for generating multinomial random variables is presented in Section 3.5. We present the shared memory parallel algorithm for 1-Flipper in Section 3.6. Finally, we conclude in Section 3.7.

3.2 Preliminaries

Below are the notations and definitions used in this chapter.

Notations. We are given a simple graph $G = (V, E)$, where V is the set of vertices, and E is the set of edges. There are a total of $n = |V|$ vertices, labeled as $0, 1, 2, \ldots, n-1$, and $m = |E|$ edges in the graph G. The adjacency list and degree of a vertex $u \in V$ are denoted as $N(u)$ and d_u, respectively. The terms *node* and *vertex*, *graph* and *network*, *neighbor list* and *adjacency list*, *loop* and *self-loop*, *label* and *vertex-id* are used interchangeably throughout the chapter. We use H, K, M, and B to denote hundreds, thousands, millions, and billions, respectively; e.g., $1M$ stands for one million. For the parallel algorithms, let p be the number of processors, and let P_i be the processor with rank i.

Edge Switch. An edge switch operation replaces two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$, selected uniformly at random from E, by new edges $e_3 = (u_1, v_2)$ and $e_4 = (u_2, v_1)$, as shown in Figure 3.1. If $u_1 = v_2$ or $u_2 = v_1$, then the above edge switch creates self-loops. The edge switch creates parallel edges, if edge (u_1, v_2) or (u_2, v_1) already exists in the graph.

Note that the set of edges E changes dynamically during the course of an edge switch process and the edges are selected from the current set of edges at a given time. During an edge switch operation, a selected edge can be categorized as one of the following two types. (*i*) *Original edge:* an edge that has not participated in any of the previous edge

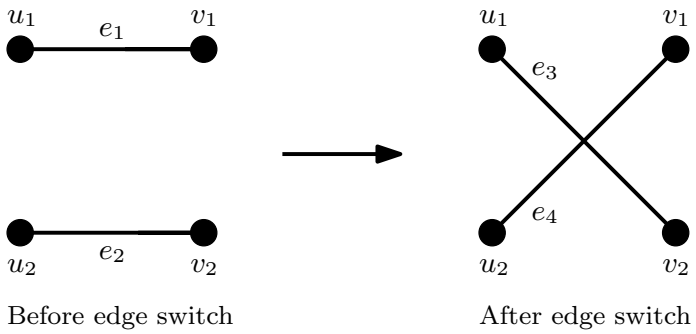


Figure 3.1: *An edge switch operation replaces two randomly selected edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ by new edges $e_3 = (u_1, v_2)$ and $e_4 = (u_2, v_1)$.*

switch operations and is still unchanged. (*ii*) *Modified edge:* any edge participating in an edge switch operation is replaced by a new edge, and such a new edge (e.g., e_3 or e_4 in Figure 3.1) is called a modified edge.

Visit Rate. We define the *visit rate* as the ratio of the number of modified edges to the total number of edges in G. Let m be the number of edges in G, and let m' be the number of modified edges. Then the visit rate is $x = m'/m$.

3.3 Edge Switch

In this section, we first determine the expected number of edge switch operations for a given visit rate, and then we present the sequential algorithm for switching edges.

3.3.1 Determining the Number of Edges to Switch for a Given Visit Rate

Let t be the total number of edge switch operations, and let $T = 2t$ be the number of edges switched to achieve a visit rate x. Since edge switching is a random process, performing the same number of edge switch operations in different executions of the same edge switch algorithm may exhibit different visit rates. Thus having an exact value of T in advance is not possible. However, we can calculate the *expected* value of T as described below. As we demonstrate later in this section, using this expected value of T leads to a very close approximation of the visit rate. Finding the expected value of T is similar to the coupon collector problem [3]. Our goal is to have $m' = mx$ modified edges in the graph by switching a sequence of edge pairs. The remainder $(m - m')$ of the edges remain unchanged.

At some point, there are already $(i-1)$ modified edges in the graph. From this point to have the i-th modified edge we need T_i number of edges switched. The probability of selecting the i-th original edge from the graph, given that there are $(i-1)$ modified edges, is $p_i = \frac{m-(i-1)}{m}$. Here, T and T_i are random variables, and T_i has a geometric distribution with expectation $1/p_i$. Using the linearity of expectation,

$$\begin{aligned} E[T] &= \sum_{i=1}^{mx} E[T_i] \\ &= \sum_{i=1}^{mx} \frac{1}{p_i} \\ &= \sum_{i=1}^{mx} \frac{m}{m-(i-1)} \\ &= m\left(\sum_{i=1}^{m} \frac{1}{i} - \sum_{i=1}^{m(1-x)} \frac{1}{i}\right) \\ &= m\left(H_m - H_{m(1-x)}\right) \end{aligned} \tag{3.1}$$

where H_m is the m-th harmonic number. For large m, $H_m \approx \ln m$, and consequently $E[T] \approx -m\ln(1-x)$ for $x < 1$, and $E[T] \approx m \ln m$ for $x = 1$. Note that every edge switch operation involves two edges. Now if we assign t to be $E[T]/2$, we obtain a visit rate extremely close to x as demonstrated below.

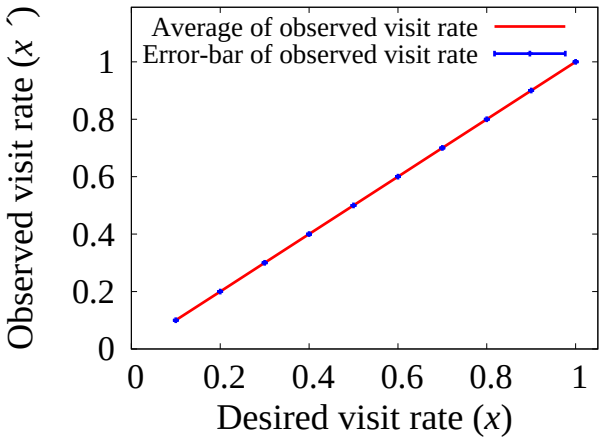


Figure 3.2: *Observed visit rate is almost equal to the desired visit rate for Miami network. The error is so small that the error-bar is almost invisible.*

Desired	Observed visit rate	
visit rate	Avg. error rate (%)	STDEV
0.1	0.00745	8.13E-6
0.2	0.00858	1.41E-5
0.3	0.00907	1.76E-5
0.4	0.00802	3.52E-5
0.5	0.00687	2.34E-5
0.6	0.00650	3.38E-5
0.7	0.00701	4.37E-5
0.8	0.01030	5.55E-5
0.9	0.00824	4.46E-5
1.0	2.4E-6	2.06E-8

Table 3.1: *Average error rate and standard deviation (STDEV) of observed visit rates for Miami network are near to 0. For each desired visit rate, ten experiments are performed.*

We perform experiments on a contact network of Miami city having $m = 52.7M$ edges (see Section 3.4.7 for details) to achieve a visit rate of $x = 1$, i.e., visit all of the $52.7M$ edges. The expected value of T is calculated using $E[T] \approx m \ln m$, and the edge switch algorithm performs $t = 468.5M$ edge switch operations. We repeat this experiment ten times and observe a visit rate of $x' = 1$ (visiting all edges) for 20% times, $x' = 0.99999998$ (visiting all but one edge) for 60% times and $x' = 0.99999994$ (visiting all but three edges) for 20% times. Thus the observed visit rates are extremely close to x. We perform additional experiments for desired visit rates $x = 0.1, 0.2, \ldots, 1$ on the Miami network. Each experiment is repeated ten times. Figure 3.2 demonstrates that the observed visit rates are almost equal to the desired visit rates. We plot the minimum and maximum of observed visit rates using error-bars. These values are so close to the desired visit rates that they almost overlap with each other and it is difficult to distinguish them in the figure. To better understand the differences between the desired and observed visit rates, we further compute the average error rate and standard deviation of the observed visit rates, which are shown in Table 3.1. The average error rate (%) is calculated as $\frac{\sum_i |x_i - x'_i|}{e x_i} \times 100\%$, where x_i and x'_i are the desired and observed visit rates, respectively, in the i-th experiment and e is the total number of experiments. The maximum, minimum and average error rates of the total 100 experiments are 0.027%, 0% and 0.007%, respectively, which are almost negligible. Therefore, for large m, we achieve a very close approximation of x, which is sufficient for almost all practical purposes.

Note that we can mark the modified edges and always select two original edges for the next edge switch operation. In such a case for a visit rate x to have mx modified edges, we simply need to perform $\frac{mx}{2}$ edge switch operations. For a specific application, one can do so. If we do not allow a modified edge to participate in any later edge switch operation, the process may not produce many networks with the same degree sequence. Unrestricted and independent random choice of the edges helps us obtain a random graph from the space of the graphs with the same degree sequence.

Furthermore, the visit rate can also be defined in other ways and converted to t. Our parallel algorithms can be used to perform t edge switch operations, irrespective of how t is obtained.

3.3.2 Keeping the Graph Simple

Because the edge switch problem deals with a simple graph, we need to ensure that none of the edge switch operations create self-loops or parallel edges. An edge switch between edges (u_1, v_1) and (u_2, v_2) may create a

- **Parallel edge:** if $u_1 \in N(v_2)$, $v_2 \in N(u_1)$, $u_2 \in N(v_1)$ or $v_1 \in N(u_2)$.
- **Self-loop:** if $u_1 = v_2$ or $u_2 = v_1$.

An edge switch operation does not make any change to the graph if the pair of edges remains the same after switching, and we say such an edge switch operation is *useless*. An edge switch between (u_1, v_1) and (u_2, v_2) is useless if $u_1 = u_2$ or $v_1 = v_2$. For an edge switch operation, two edges are selected and switched if the switch is not useless and does not create parallel edges or loops.

3.3.3 Sequential Edge Switch

We are given a simple graph $G = (V, E)$ and the number of edge switch operations t to be performed. The sequential algorithm is quite simple. Select a pair of edges uniformly at random and switch them if the resultant graph remains simple. This operation is repeated until t pairs of edges are switched. The graph, specifically the edge set, dynamically changes with the course of the edge switch process. Let $G' = (V, E')$ be such a graph where E' is the current set of edges at a given time. Algorithm 1 shows the pseudocode of switching edges sequentially. The adjacency list of a vertex can be stored using a balanced binary tree. Searching such an adjacency list of a vertex u to determine the possibility of parallel edge creation takes $O(\log d_u)$ time. If (u_1, v_1) and (u_2, v_2) are the edges participating in the i-th edge switch operation, then the time to switch t pairs of edges is $O\left(\sum_{i=1}^{t} \sum_{j \in \{u_1, v_1, u_2, v_2\}} \log d_j\right) \leq O(t \log d_{max})$, where d_{max} is the maximum degree of a vertex in the graph. Note that if an edge switch operation attempts to create a parallel edge or a loop, or is useless, the edge switch operation is restarted by selecting a new pair of edges. For a large and relatively sparse network, this probability is very small. As a result, the number of edge switch operations restarted is significantly smaller than t. Thus we have the run time $O(t \log d_{max})$.

3.4 Parallel Edge Switch

Although the sequential algorithm is very simple, parallelizing the simple edge switch operations turns out to be a non-trivial problem for the following reasons:

- Multiple pairs of edges are selected and switched simultaneously by different processors in the parallel process, whereas the sequential process selects and switches

Algorithm 1: Sequential Edge Switch

```
1 Procedure Switch-Edges(G, t)
2     i ← 1
3     E′ ← E
4     while i ≤ t do
5         (u1, v1), (u2, v2) ← two uniform random edges in E′
6         if u1 = v2, u2 = v1, u1 ∈ N(v2), or u2 ∈ N(v1) then      // self-loop or parallel edge
7             continue
8         Replace (u1, v1) and (u2, v2) in E′ by (u1, v2) and (u2, v1) respectively
9         i ← i + 1
```

a sequence of pairs of edges, one pair after another. Designing a parallel algorithm by maintaining a stochastic process equivalent to the sequential one leads to significant challenges.

- The requirement of keeping the graph simple requires complex synchronization and communication among the processors. To achieve a good speedup by parallelization, we need to design an efficient algorithm by minimizing such communication and computation costs.

In this section, we present an efficient parallel algorithm for switching edges in large graphs, accompanied by a rigorous comparative study of several partitioning schemes.

3.4.1 Overview of the Algorithm

The input graph G is partitioned into *blocks* or subsets of edges and distributed among the p processors. Each block contains a subset of the vertices and their adjacent edges, and is assigned to a processor. All the processors then perform t edge switch operations in parallel. We need to consider two cases for an edge switch operation:

- **Local Switch.** Both edges may be selected from the same block (or processor), and this is referred to as a *local switch*.
- **Global Switch.** The edges may be chosen from different blocks, and this is referred to as a *global switch*. The processors may need to communicate with each other to complete the edge switch operation.

3.4.2 Data Structures

A graph can be stored as adjacency lists or as an adjacency matrix. In an adjacency matrix, the existence of any edge can be determined in constant time; however, it takes $O(n^2)$ space. Our algorithms use adjacency lists, which require $O(m + n)$ space. Usually, $N(u)$ contains all neighbors of u. The graph can also be presented in many different representations such as Compressed Sparse Row (CSR) [52] format, which also requires $O(m + n)$ space.

Reduced Adjacency List. For an edge (u, v), if $N(u)$ and $N(v)$ belong to different blocks, the edge can be selected from two different blocks and participate in two different edge switch operations at the same time, leading to inconsistency. To ensure that any edge (u, v) can be selected only from one block, only neighbors with higher labels are kept in the adjacency list of a vertex u, i.e., $N(u) = \{v \in V | (u, v) \in E, u < v\}$, which is referred to as the *reduced adjacency list*. Although it is possible to deal with the above issue by storing all neighbors in the adjacency list, it will incur higher communication costs. Every edge switch operation involves updating four vertices' adjacency lists: one update for each end vertex of an edge. A reduced adjacency list minimizes the number of updates to only two or three vertices' adjacency lists; the details are discussed later in Section 3.4.4. Thus a reduced adjacency list reduces memory footprint, communication cost, and computation cost.

Straight and Cross Edge Switch. A difficulty arises from using the reduced adjacency list. If $N(u)$ contains all the neighbors of u, any edge (u_1, v_1) can be selected either as (u_1, v_1) from $N(u_1)$ (considering ordered pair), or as (v_1, u_1) from $N(v_1)$. The probability of being selected each way is $1/2m$. Let (u_1, v_1) and (u_2, v_2) (considering no ordering) be two edges selected for an edge switch operation. Depending on whether the edge (u_1, v_1) is selected from $N(u_1)$ or $N(v_1)$, and the other edge (u_2, v_2) is chosen from $N(u_2)$ or $N(v_2)$, the edges are replaced by either (u_1, v_2) and (u_2, v_1), or (u_1, u_2) and (v_1, v_2). Assuming $u_1 < v_1$ and $u_2 < v_2$, there is no possibility of selecting edges as (v_1, u_1) and (v_2, u_2) (considering ordered pair) due to the use of a reduced adjacency list. Therefore, an edge switch between (u_1, v_1) and (u_2, v_2) (considering unordered pair) misses the chance of generating the edges (u_1, u_2) and (v_1, v_2). This problem is solved by replacing the selected edges by either (u_1, u_2) and (v_1, v_2) with probability $1/2$, referred to as the *straight switch*, or (u_1, v_2) and (u_2, v_1) with probability $1/2$, referred to as the *cross switch*, as shown in Figure 3.3.

3.4.3 Partitioning the Network

Partitioning schemes usually have a significant impact on the performance of parallel graph algorithms in terms of both runtime and memory. A good partitioning scheme for the

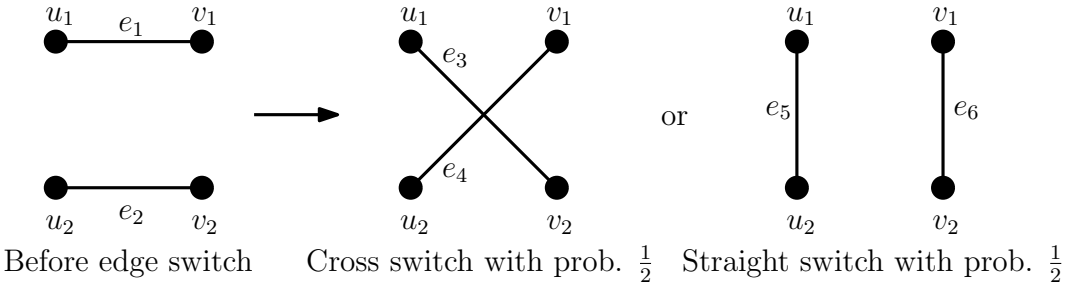


Figure 3.3: *Straight and cross edge switch.*

parallel algorithm should have the following properties:

- It can efficiently (in terms of runtime) partition a given network.
- Given a vertex v, the block where v belongs can be efficiently determined.
- The workload is uniformly distributed among the processors for different types of networks. The ***workload*** at a processor P_i is the number of edge switch operations P_i performs.

For a given simple graph $G = (V, E)$, we partition V into p disjoint subsets, $V_0, V_1, \ldots, V_{p-1}$, such that $\bigcup_i V_i = V$. Let V_i be the subset of vertices and E_i be the subset of edges in the block (or subgraph) $G_i = (V_i, E_i)$ belonging to P_i such that $E_i = \{(u, v) \in E | u \in V_i, u < v\}$. The reduced adjacency list of a vertex entirely belongs to one block. Note that the blocks are disjoint, i.e., $E_i \bigcap E_j = \emptyset$ for $i \neq j$, and $\bigcup_i E_i = E$. The subset of edges E_i at P_i dynamically changes with edge switch operations and the edges are selected from the current subset of edges at a given time.

We study four different partitioning schemes in conjunction with the algorithm and they are described below.

Consecutive Partitioning

The graph is partitioned such that a subset of consecutive (in terms of vertex labels) vertices are assigned to each block $G_i = (V_i, E_i)$ and each block contains roughly m/p edges. It is easy to determine which vertex belongs to which block. We refer to this partitioning scheme as ***Consecutive Partitioning (CP)***. Assigning a consecutive set of vertices to a block is also known as 1D or row-wise partitioning and is employed by the Parallel Boost Graph Library [52].

Hash-Based Partitioning

Another approach can be to use a ***Hash-based Partitioning (HP)*** scheme. A hash function can be a simple algebraic expression mapping vertex labels to blocks. Hash functions are deterministic in nature, and by using some simple hash functions, it can be very easy and efficient to determine which vertex belongs to which block, thus obeying the first two criteria of a good partitioning scheme. Hash functions may assign a different number of vertices and edges to blocks.

A good hash function for the partitioning schemes should have the following properties.

- It is simple and efficient to determine which vertex belongs to which block.
- Vertices are dispersed and well-distributed among the processors, i.e., all of the blocks are almost equal in size.

Division hash function (HP-D), ***multiplication hash function (HP-M)***, and ***universal hashing (HP-U)*** are a few such hash functions and they are described below.

1. **Division Hash Function.** A simple hash function can be a division function (HP-D) [35]. This scheme uses the following function:

$$h(v) = v \mod p \tag{3.2}$$

   where p is the number of processors.

2. **Multiplication Hash Function.** Another simple hash function is a multiplication function (HP-M) [35]. The hash function is:

$$h(v) = \lfloor p(va - \lfloor va \rfloor) \rfloor \tag{3.3}$$

   where $a \in (0, 1)$ is a constant. The fractional part of va is extracted by $va - \lfloor va \rfloor$ and is then multiplied by the number of processors p to determine the block where v belongs. Although this scheme works with any value of $a \in (0, 1)$, we use $a = (\sqrt{5} - 1)/2$ as suggested in [35] to obtain reasonably good performance.

3. **Universal Hashing.** The division and multiplication hash functions are quite simple. However, their workload distributions among the processors are dependent on the vertex labels of the input graph. If there is an adversary who knows the hash function being used in advance, the adversary can artificially manipulate the graph by assigning vertex labels in such a way that the workload distribution becomes

skewed. For example, many high degree vertices can be assigned to a block making the workload at the processor containing that block significantly higher compared to the other processors. To deal with such exploitation of hash functions by an adversary, universal hashing [35] can be a good choice. This scheme uses the following hash function:

$$h(v) = (((av + b) \mod c) \mod p) \tag{3.4}$$

where c is a large prime number such that all vertex labels are in the range $[0, c-1]$, $a \in [1, c-1]$ is a random integer, and $b \in [0, c-1]$ is another random integer. Since a and b are selected randomly, this method arbitrarily chooses a hash function from a large set of hash functions. As a result, there is no way for the adversary to know the exact hash function in advance or to exploit it to create a worse case scenario.

ParMETIS Partitioning

ParMETIS [66] is a well-known MPI-based parallel library for partitioning various types of unstructured graphs. It can efficiently compute high-quality partitioning of large graphs. We use the parallel multilevel k-way graph partitioning scheme and refer to this scheme as ***ParMETIS Partitioning (PP)***. Since the blocks may contain non-contiguous vertices, each processor requires $O(n)$ space to store the mapping of the vertices to blocks. To get rid of the $O(n)$ space requirement, the vertex labels can be reordered after the partitioning such that the vertices belonging to a processor are reassigned consecutive vertex labels, the lower ranked processors contain the lower vertex labels and each processor stores the starting vertex label in every processor.

Random Partitioning

Among other options, one simple way to partition a given network is assigning vertices to blocks uniformly at random. This approach may assign almost an equal number of vertices to the blocks although the number of edges may vary among them. To determine which vertex belongs to which block, each processor requires $O(n)$ space to store the mapping of the vertices to blocks. The vertex labels can be reordered to eliminate the $O(n)$ space requirement (as mentioned in the PP scheme). We refer to this scheme as ***Random Partitioning (RP)***.

3.4.4 Switching a Pair of Edges by a Single Processor

A simple approach to perform an edge switch operation is that processor P_i can select one pair of edges uniformly at random from the entire graph (i.e., selecting two processors from $[0, p-1]$ and request them for edges) and switch them by exchanging messages among the processors. However, this approach incurs significant synchronization and communication costs. Instead, P_i selects one edge (u_1, v_1), referred to as the *first edge*, uniformly at random from E_i, and another edge (u_2, v_2), referred to as the *second edge*, from the entire graph. To select the second edge, P_i selects a processor P_j with probability $|E_j|/|E|$ and requests P_j to select an edge (u_2, v_2) from E_j uniformly at random. If $P_i = P_j$, then it is a *local switch*, otherwise it is a *global switch*. Due to the use of reduced adjacency lists, one of the replacing edges (e_3, e_4, e_5 or e_6 in Figure 3.3) may belong to a different processor P_k ($P_i \neq P_k \neq P_j$); in this case, processor P_i, P_j and P_k work together to update the reduced adjacency lists of respective vertices by exchanging messages and thus complete the edge switch operation. A high-level overview of an edge switch operation is given in Algorithm 2. During the course of an edge switch operation, if any processor detects a possibility of creating loops or parallel edges, it notifies all other processors that are involved in the edge switch operation. Then the initiating processor (P_i in the above example) restarts the edge switch operation by selecting a new pair of edges.

Algorithm 2: Switching a Pair of Edges Initiated by a Processor P_i

1 $e_1 \leftarrow$ a uniform random edge in E_i
2 $P_j \leftarrow$ a random processor in $[0, p-1]$, where probability of choosing P_x is $\frac{|E_x|}{|E|}$
3 **if** $P_i = P_j$ **then** `// local switch`
4 Choose an edge e_2 from E_i to switch with edge e_1
5 Switch the edges e_1 and e_2 (P_i may communicate with a different processor P_k to complete the edge switch operation)
6 **else** `// global switch`
7 Send message $\langle e_1$**, *request* to select an edge from** $E_j \rangle$ to P_j
8 Upon receipt of the above message, P_j executes the following: `// global switch`
9 Choose an edge e_2 from E_j to switch with edge e_1
10 P_i and P_j work together to switch e_1 and e_2 (P_j may communicate with a different processor P_k to complete the edge switch operation)

Local Switch. P_i selects two edges (u_1, v_1) and (u_2, v_2) from E_i uniformly at random such that the edge switch does not create loops and is not useless. P_i decides between a *straight* and a *cross switch* with equal probability. If it is a *cross switch*, P_i checks whether (u_1, v_2)

and (u_2, v_1) create parallel edges. If no parallel edge is created, P_i removes (u_1, v_1) and (u_2, v_2), adds (u_1, v_2) and (u_2, v_1), thus completing the edge switch operation. If the edge switch is a *straight switch*, P_i determines P_k such that $\min(v_1, v_2) \in V_k$. If $P_i = P_k$, P_i determines whether (u_1, u_2) and (v_1, v_2) create parallel edges. If they do not create any parallel edge, P_i removes (u_1, v_1) and (u_2, v_2), adds (u_1, u_2) and (v_1, v_2) and completes the edge switch operation. If $P_i \neq P_k$, P_i checks whether (u_1, u_2) creates parallel edges. If the graph remains simple, P_i sends a message to P_k requesting to add (v_1, v_2). If (v_1, v_2) does not create parallel edges, P_k adds (v_1, v_2) and sends a message back to P_i informing it of the updates at P_k. Upon receiving this message, P_i removes (u_1, v_1), (u_2, v_2) and adds (u_1, u_2).

Global Switch. In a *global switch*, two edges are selected from two different processors, say P_i and P_j, $i < j$. Assuming P_i initiates the edge switch operation, P_i selects an edge $e_1 = (u_1, v_1)$ from E_i uniformly at random. P_i sends a message, containing the edge e_1 and a request to select an edge from E_j, to P_j. Upon receiving this message from P_i, processor P_j selects $e_2 = (u_2, v_2)$ from E_j uniformly at random, and decides between a *straight* and a *cross* switch with equal probability. At this point, P_j knows the new edges that will replace e_1 and e_2; we refer to these new edges as *potential edges* until the updates take place. Next we describe the *cross switch* in detail.

Processor P_j checks whether $u_2 = v_1$ and $v_1 = v_2$ to detect a loop and a useless edge switch respectively. If it does not create a loop and is not useless, P_j determines P_k such that $\min(u_2, v_1) \in V_k$. We need to consider the following three cases.

1. *Case $P_k = P_j$:*
   P_j checks whether (u_2, v_1) creates parallel edges. If a parallel edge is not created, then P_j sends v_2 to P_i. P_i checks whether (u_1, v_2) creates parallel edges. If the graph remains simple, P_i removes edge (u_1, v_1), adds edge (u_1, v_2), and sends a message back to P_j informing the updates at P_i. Upon receiving this message, P_j removes (u_2, v_2) and adds (u_2, v_1), thus completing the edge switch operation.

2. *Case $P_k = P_i$:*
   P_j sends a message, containing e_2 and a request to add both the new edges to P_i. Processor P_i checks whether (u_1, v_2) and (u_2, v_1) create parallel edges. If no parallel edge is created, P_i removes (u_1, v_1), adds edges (u_1, v_2) and (u_2, v_1), and sends a message back to P_j indicating the updates at P_i. Then P_j completes the edge switch operation by removing (u_2, v_2).

3. *Case $P_i \neq P_k \neq P_j$:*
   P_j sends (u_2, v_1) and v_2 to P_k. If (u_2, v_1) does not create any parallel edge, P_k sends v_2 to P_i. P_i checks whether (u_1, v_2) creates any parallel edge. If the graph remains

simple, P_i removes (u_1, v_1), adds (u_1, v_2), and sends messages to P_j and P_k notifying the updates taken place at P_i. Then P_j removes edge (u_2, v_2), and P_k adds edge (u_2, v_1), thus completing the edge switch operation.

A similar approach is followed for $i > j$ and for a straight switch as well. The use of reduced adjacency lists eliminates the following two constraints: (*i*) $u_1 = u_2$, and (*ii*) $u_1 = v_2$ if $i < j$, or $u_2 = v_1$ if $i > j$.

3.4.5 Simultaneous Edge Switches by All Processors

In a sequential algorithm, pairs of edges are selected randomly, one pair after another; as a result, the number of edges selected from each block E_i may not be equal. To have an equivalent parallel algorithm, we need to select the same number of edges from each block E_i as the sequential algorithm would do. Let X_i be the number of first edges selected from E_i by a sequential algorithm. A sequential algorithm does not need to know X_i in advance. However, for the parallel algorithm, for each i, X_i needs to be determined in advance so that processors can simultaneously perform edge switches in parallel. For any edge switch operation, the probability that the first edge is selected from E_i is $q_i = |E_i|/|E|$ for $i = 0, 1, \ldots, p-1$, and we have $\sum_{i=0}^{p-1} q_i = 1$. Then it is easy to see that the random variables X_i for $i = 0, 1, \ldots, p-1$ are multinomially distributed with parameters $(t, q_0, q_1, \ldots, q_{p-1})$; i.e.,

$$\langle X_0, X_1, \ldots, X_{p-1} \rangle \sim \mathcal{M}(t, q_0, q_1, \ldots, q_{p-1}). \tag{3.5}$$

The time complexity of the best known sequential algorithm, known as the *conditional distributed method* [37], for generating multinomial variables is $\Theta(t)$. Thus to have an efficient parallel algorithm for our edge switch problem, we need to use an efficient parallel algorithm for generating multinomial random variables. To the best of our knowledge, there is no published parallel algorithm for this problem. In Section 3.5, we present an efficient parallel algorithm for computing multinomial random variables that runs in $O\left(\frac{t}{p} + p \log p\right)$ time.

Each processor P_i simultaneously performs X_i edge switches and serves other processors' requests as well. After completing one edge switch, P_i proceeds to its next edge switch operation. Below we discuss two *issues* that arise from performing edge switch operations simultaneously.

- **Creating parallel edges in a new way.** Even after maintaining all the constraints to keep a graph simple, parallel edges can be created in a different way. As multiple

pairs of edges are switched by multiple processors simultaneously, the same new edge can be created by multiple processors at the same time. For example, more than one instance of an edge (u, v) is created simultaneously if more than one of the following four edge switches are performed simultaneously by different processors, where '$-$' denotes an end vertex of an edge. (*i*) Cross edge switch between $(u, -)$ and $(-, v)$. (*ii*) Cross edge switch between $(-, u)$ and $(v, -)$. (*iii*) Straight edge switch between $(u, -)$ and $(v, -)$. (*iv*) Straight edge switch between $(-, u)$ and $(-, v)$. Keeping track of *potential edges* at each processor ensures no parallel edges will be created in the above mentioned way.

- **Changing probability values with the course of an edge switch process.** As the edges are switched, the number of edges changes (i.e., increases or decreases) among the blocks due to the use of reduced adjacency lists. As a result, the probability values (q_i) of selecting edges from different blocks change, which need to be updated dynamically. However, updating the probability values after every edge switch operation incurs substantial communication costs, which in turn slows down the algorithm significantly. To deal with this difficulty, the processors perform a fixed number of edge switch operations (referred to as *step-size* and denoted by s) in a step, and then update the probability values that are used in the next step. Therefore, the algorithm performs edge switch operations in a number of steps. At the beginning of each step, s edge switch operations are distributed among p processors using the multinomial distribution. The program terminates when all of the t edge switch operations are performed in $\lceil \frac{t}{s} \rceil$ steps. With a reasonable step-size, a very close approximation of the sequential algorithm is achieved. The experimental results are shown later in Section 3.4.7.

Summary of the Parallel Algorithm. Let s be the *step-size*, and q be the probability vector $\langle q_0, q_1, \ldots, q_{p-1} \rangle$. All the processors perform s edge switch operations in one step, thus requiring a total of $\lceil \frac{t}{s} \rceil$ steps. If t is not a multiple of s, then $(t - s\lfloor \frac{t}{s} \rfloor)$ edge switch operations are performed in the last step. Below is a summary of the parallel algorithm.

1. **Generating multinomial random variables.** At the beginning of each step, s edge switch operations are distributed among p processors using the parallel algorithm for generating multinomial random variables with parameters $(s, q_0, q_1, \ldots, q_{p-1})$. This takes $O\left(\frac{s}{p} + p \log p\right)$ time. Let us denote S_i to be the number of edge switch operations that a processor P_i performs in the current step.

2. **Performing edge switch operations.** To perform an edge switch operation, a processor P_i selects one edge e_1 from E_i and the other edge e_2 from the entire graph and

completes the edge switch operation in conjunction with other processors (details in Section 3.4.4). Each processor P_i simultaneously performs S_i edge switch operations and serves other processors' requests as well. For an edge switch operation, a constant amount of message exchange is required; edges are updated in constant time and checking for parallel edges takes $O\left(\log d_{max}\right)$ time. Thus, performing S_i edge switch operations at P_i takes $O\left(S_i \log d_{max}\right)$ time.

3. **Updating probability vector and termination.** After completing S_i edge switch operations in the current step, P_i sends *end-of-step* signals (or messages) to each processor requiring $O\left(\log p\right)$ time. P_i continues to serve requests from other processors until receiving end-of-step signals from every processor, i.e., the end of the current step. At the end of each step, P_i receives $|E_j|$ from each P_j by exchanging messages and it takes $O\left(\log p\right)$ time due to the communication costs. P_i updates q with the received $|E_j|$ values in $O(p)$ time. Then, in the next step, s edge switch operations is again distributed among p processors using multinomial distribution with the updated q and edge switch operations are performed. This process continues until t edge switch operations are performed in $\lceil \frac{t}{s} \rceil$ steps.

3.4.6 Properties of Parallel Edge Switch

In this section, we examine some stochastic properties of the parallel edge switch process and study how stochastically similar it is to the sequential edge switch process.

Recall that in the sequential edge switch process, one pair of edges is selected uniformly at random, and the edges are switched before selecting the next pair of edges. After completing the i-th edge switch operation, one or both of the two new edges generated by the i-th switch can be selected for the $(i+1)$-th edge switch operation. In the parallel edge switch process, multiple pairs of edges are selected and switched simultaneously by different processors, and thus, the edges generated simultaneously by multiple processors cannot be selected for a simultaneous edge switch operation (restricting its choice). It raises the question of whether these two processes are stochastically equivalent or how close are they stochastically? We try to answer this question by studying the similarity of their effect, i.e., the resultant graphs generated by these two edge switch processes beginning with the same initial graph.

The *stochastic equivalence* of the sequential and parallel edge switch processes can be defined as follows. Let G_s^t and G_p^t be the resultant graphs after performing t edge switch operations by the sequential and parallel edge switch processes, respectively, where both processes begin with the same initial graph G. We say the two processes are stochastically equivalent

if $\Pr\{G_s^t = G'\} = \Pr\{G_p^t = G'\}$ for all graphs G' with the same degree sequence as G.

Theoretical analysis of the above stochastic equivalence seems to be difficult. Experimental analysis can also be prohibitively time-consuming. As the space of the graphs with a given degree sequence can be very large, estimating probabilities of generating G' by a reasonable number repetitions of the edge switch processes can be error prone.

Instead, we measure "similarity" of the two stochastic processes. We say the sequential and parallel processes are *similar* if they satisfy the following two conditions.

1. The distribution of the number of edges switched among different blocks (i.e., subsets of edges) is the same in both G_s^t and G_p^t, the resultant graphs of the sequential and parallel processes, respectively. This goal is achieved by the use of the multinomial distribution as described before in Section 3.4.5.

2. At the end of the edge switch processes, the distribution of the number of edges across different sets of vertices is the same for both sequential and parallel processes. Let $n_s(V_i, V_j)$ and $n_p(V_i, V_j)$ be the number of cross edges between the sets of vertices V_i and V_j in the resultant graphs G_s^t and G_p^t, respectively. For any positive integer t, after switching t pairs of edges, the distributions of $n_s(V_i, V_j)$ and $n_p(V_i, V_j)$, for all i, j, are the same.

The resultant graphs, G_s^t and G_p^t, are divided into r blocks (i.e., $0 \leq i, j \leq r-1$), with each block containing an equal number of vertices having consecutive vertex labels. Note that the i-th block V_i of G_s^t and G_p^t have the same set of vertices with vertex labels in $\left[\frac{i|V|}{r}, \frac{(i+1)|V|}{r} - 1\right]$ (assuming n is a multiple of r). The *edge difference* $ED(G_s^t, G_p^t)$ across different sets of vertices between G_s^t and G_p^t is computed using equation (3.6). We define the *error rate* $ER(G_s^t, G_p^t)$ between G_s^t and G_p^t as shown in equation (3.7), where the maximum value of $ED(G_s^t, G_p^t)$ can be $2m$. Due to randomness, some error rate can be observed even between two resultant graphs, G_{s1}^t and G_{s2}^t, generated by the sequential process in two different runs. If $ER(G_s^t, G_p^t)$ is roughly equal to $ER(G_{s1}^t, G_{s2}^t)$, then the sequential and parallel processes are said to be *similar*. For the same pair of resultant graphs G_s^t and G_p^t, the value of $ER(G_s^t, G_p^t)$ is different for different values of r. As a result, for a particular value of r, we are interested in how close $ER(G_s^t, G_p^t)$ and $ER(G_{s1}^t, G_{s2}^t)$ are to each other rather than the value of the error rate. The experimental results are explained in the next section.

$$ED\left(G_s^t, G_p^t\right) = \sum_{i,j \geq i} \left| n_s(V_i, V_j) - n_p(V_i, V_j) \right| \tag{3.6}$$

$$ER\left(G_s^t, G_p^t\right) = \frac{ED\left(G_s^t, G_p^t\right)}{2m} \times 100\% \tag{3.7}$$

3.4.7 Performance Analysis

In this section, we present strong and weak scaling of our parallel algorithm, demonstrate the *similarity* of the sequential and parallel edge switch processes, and analyze the trade-offs among step-size, error rate, and speedup. We also present a comparative study of the performance exhibited by the partitioning schemes along with the algorithms.

Experimental Setup. We use a high performance computing cluster of 64 Intel Sandy Bridge compute nodes (Dell C6220). Each computing node consists of a dual-socket Intel Sandy Bridge E5-2670 2.60GHz 8-core processor (16 cores per node) and 64GB of 1600MHz DDR3 RAM. The computing nodes are interconnected by Qlogic QDR Infiniband interconnects. To implement our algorithm, we use C++ and the MPICH2 implementation (version 1.9) of MPI. The CP, HP, and RP schemes are implemented as part of the algorithms, and we use ParMETIS [66] for the PP scheme.

Data Sets. We use both real-world and artificial networks for the experiments. A summary of the networks is provided in Table 3.2. New York, Los Angeles, and Miami are synthetic, yet realistic, social contact networks [12]. Each vertex represents a person in the corresponding city, and each edge represents any 'physical' contact between two individuals within a 24-hour time period. Flickr is an image-based online community network [71].

Table 3.2: *Data sets used in the experiments.* ***M*** *and* ***B*** *denote millions and billions, respectively.*

Network	Type of network	Vertices	Edges	Avg. degree
New York	Social Contact	20.38**M**	587.3**M**	57.63
Los Angeles	Social Contact	16.33**M**	479.4**M**	58.66
Miami	Social Contact	2.1**M**	52.7**M**	50.4
Flickr	Online Community	2.3**M**	22.8**M**	19.83
LiveJournal	Social	4.8**M**	42.8**M**	17.83
Small World	Random	4.8**M**	48**M**	20
Erdős-Rényi	Erdős-Rényi Random	4.8**M**	48**M**	20
PA-100M	Pref. Attachment	100**M**	1**B**	20

LiveJournal is a social network blogging site [71]. The small world graph is generated using the Watts-Strogatz small world graph model [111], Erdős-Rényi is generated using the Erdős-Rényi graph model [24], and PA is generated using the Preferential Attachment graph model [11].

Strong Scaling. Figures 3.4, 3.5, 3.6, and 3.7 show the strong scaling of the parallel algorithm of edge switch using the CP, PP, HP-U, and RP schemes, respectively. The algorithm performs t edge switch operations for a visit rate of $x = 1$ using a step-size of $t/100$. All the speedups are measured against the runtime of the sequential algorithm given in Table 3.3. We have experimented with eight different graphs and achieved a maximum speedup of 115 with 800 processors using the RP scheme on the Miami graph. Using the RP scheme, the harmonic mean speedup is 73.25 with 1024 processors. The absolute runtime of the parallel algorithm using the HP-U scheme is shown in Figure 3.8. The maximum, minimum, average and standard deviation of the speedup from 25 experiments of the parallel algorithm using the HP-U scheme for the Miami graph is shown in Table 3.4. Speedup varies for different graphs because of the types of graphs and difference in workload distribution among the processors. Speedup starts decreasing after some point with the increase of the number of processors indicating the domination of communication costs over computation costs.

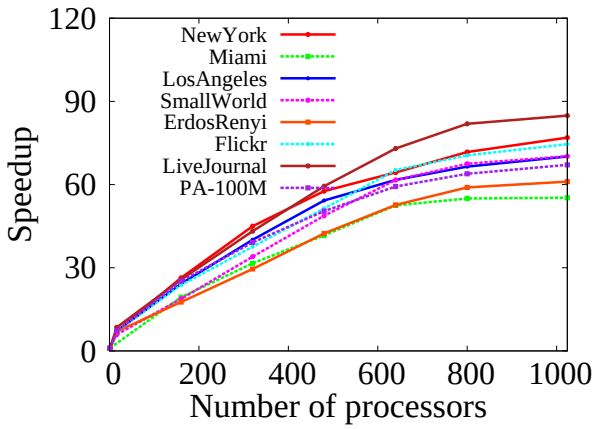


Figure 3.4: *Strong scaling of our algorithm on eight different graphs using the CP scheme.*

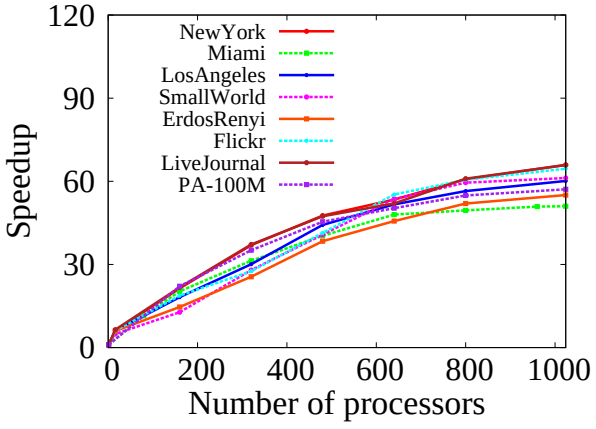


Figure 3.5: *Strong scaling of our algorithm on eight different graphs using the PP scheme.*

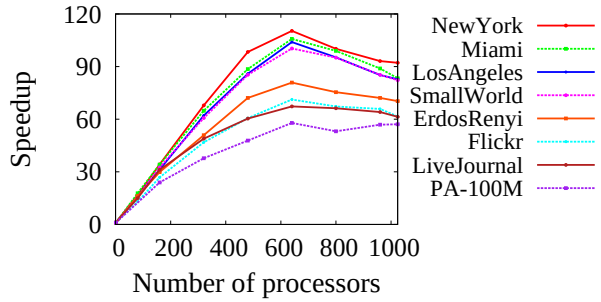


Figure 3.6: *Strong scaling of our algorithm on eight different graphs using the HP-U scheme.*

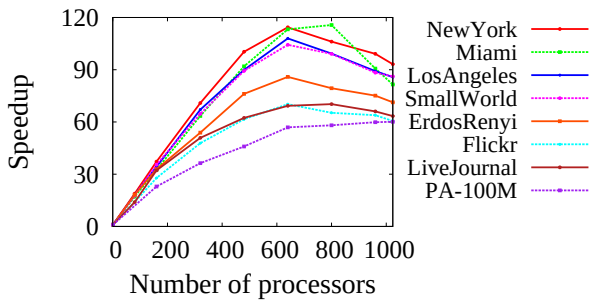


Figure 3.7: *Strong scaling of our algorithm on eight different graphs using the RP scheme.*

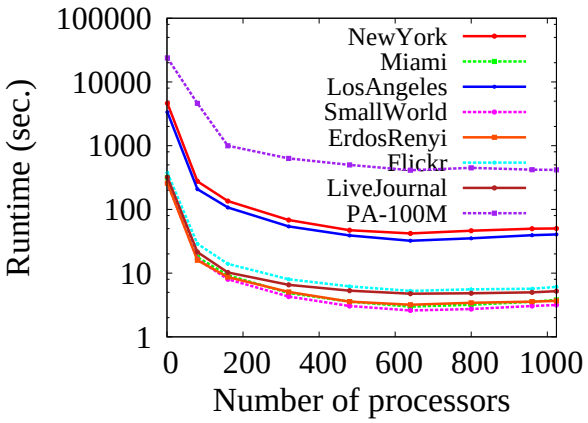


Figure 3.8: *Runtime of the parallel algorithm using the HP-U scheme for visit rate = 1.*

Network	**Time (sec.)**
New York	4634.6
Los Angeles	3386.5
Miami	316.3
Flickr	374.6
LiveJournal	320.0
Small World	260.1
Erdős-Rényi	258.9
PA-100M	23849.3

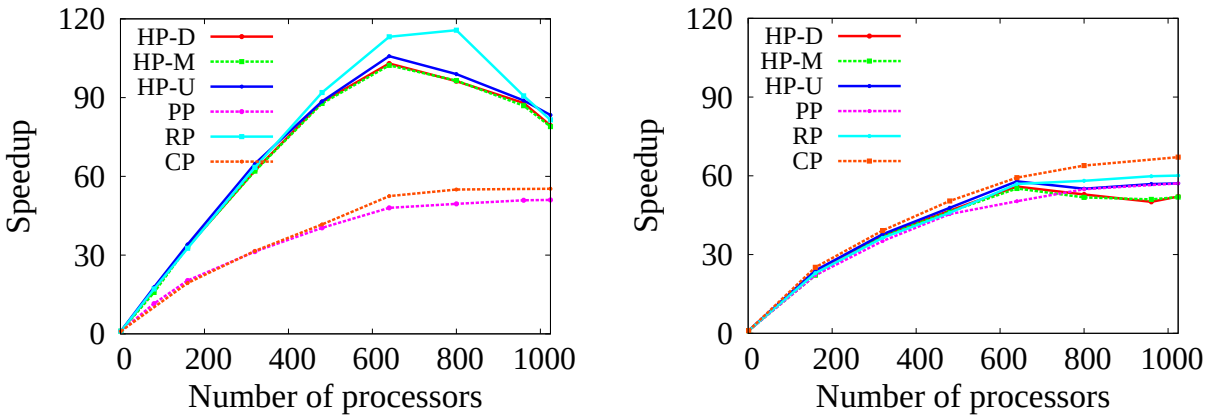
Table 3.3: *Runtime of the sequential algorithm for visit rate = 1.*

A comparison of the strong scaling performance of the parallel algorithms using different schemes on Miami and PA-100M graphs is demonstrated in Figure 3.9. The RP scheme shows better strong scaling for the Miami graph, whereas CP out-performs the other schemes for the PA-100M graph. To understand why speedup varies for different schemes and how well the schemes perform for various types of graphs, we further investigate workload distributions of different schemes for Miami and PA-100M graphs. We use $p = 1024$ processors for the remainder of the experiments in this section.

Load Balancing. Figures 3.10 and 3.11 show the distributions of vertices and edges (at the beginning of execution), respectively, among the processors in different schemes on the Miami graph. The HP, RP, and PP schemes assign roughly an equal number of vertices, whereas the CP scheme initially assigns almost an equal number of edges among the processors. Due to the use of reduced adjacency lists, the number of vertices assigned

Table 3.4: *Maximum, minimum, average and standard deviation (STDEV) of the speedup provided by the parallel algorithm using the HP-U scheme for the Miami graph. We use values of 25 experiments.*

Speedup	**Number of processors**							
	80	**160**	**320**	**480**	**640**	**800**	**960**	**1024**
Maximum	17.93	34.53	65.89	90.4	109.24	103.3	94.2	88.32
Minimum	17.61	33.34	62.88	85.56	102.63	94.4	82.96	77.49
Average	17.8	34.1	64.8	88.6	105.8	98.9	88.9	83.4
STDEV	0.08	0.41	0.81	1.43	2.13	2.89	3.56	3.67



(a) *Miami graph* ***(b)*** *PA-100M graph*

Figure 3.9: *A comparison of strong scaling of the parallel algorithms using different partitioning schemes for Miami and PA-100M graphs.*

to processors by the CP scheme gradually increases with the increase of processor ranks despite having an equal number of edges among the processors. The numbers of edges initially assigned to all the processors by the HP and RP schemes are very close, and the distributions can be considered as roughly load balanced although they are not as good as that of the CP scheme. On the other hand, the PP scheme considers two copies of each edge (as (u, v) and (v, u)) during the partitioning process, whereas our algorithm stores only one copy of edge ((u, v) such that $u < v$) to minimize the computation and communication costs and the memory footprint. Note that although the PP scheme may not assign contiguous vertices to blocks, in many cases, a block produced by the PP scheme contains a large portion of vertices (compared to all the vertices assigned to that block)

having consecutive vertex labels. As a result, the partitioning by the PP scheme incurs poor distribution of edges among the processors for our edge switch algorithm.

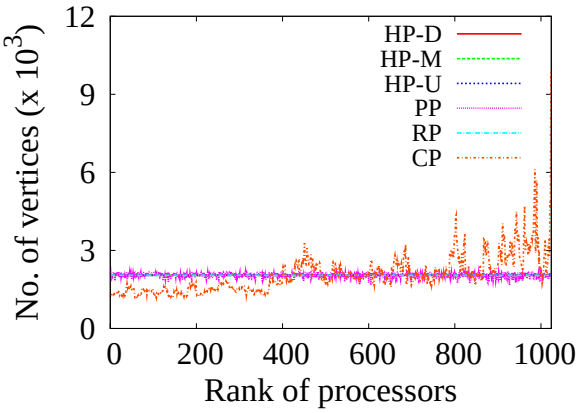


Figure 3.10: *Distribution of vertices among the processors in different partitioning schemes for the Miami graph.*

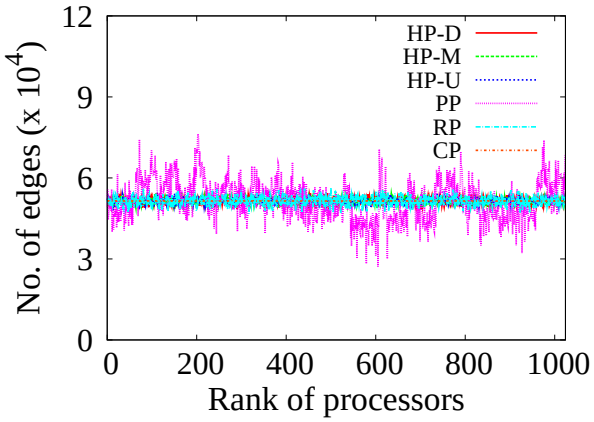


Figure 3.11: *Distribution of edges (at the beginning of execution) among the processors in different partitioning schemes for the Miami graph.*

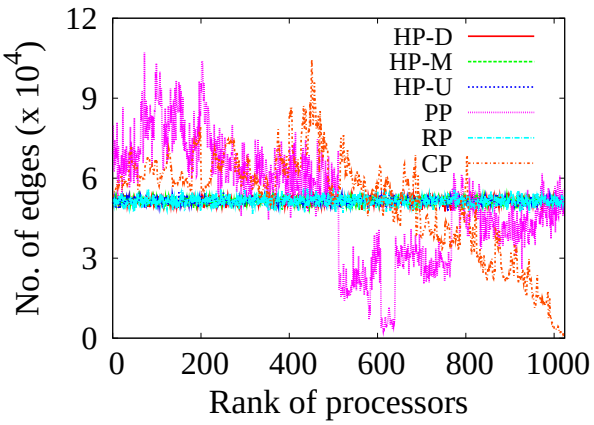


Figure 3.12: *Distribution of edges (after completing execution) among the processors in different partitioning schemes for the Miami graph.*

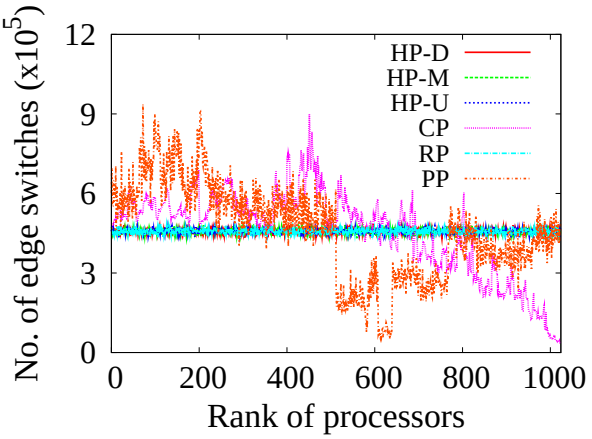


Figure 3.13: *Distribution of workload (number of edge switch operations) among the processors in different partitioning schemes for the Miami graph.*

Unlike the PP scheme, the parallel algorithms using the CP, RP, and HP schemes start the edge switch process with almost an equal number of edges at each processor as shown in

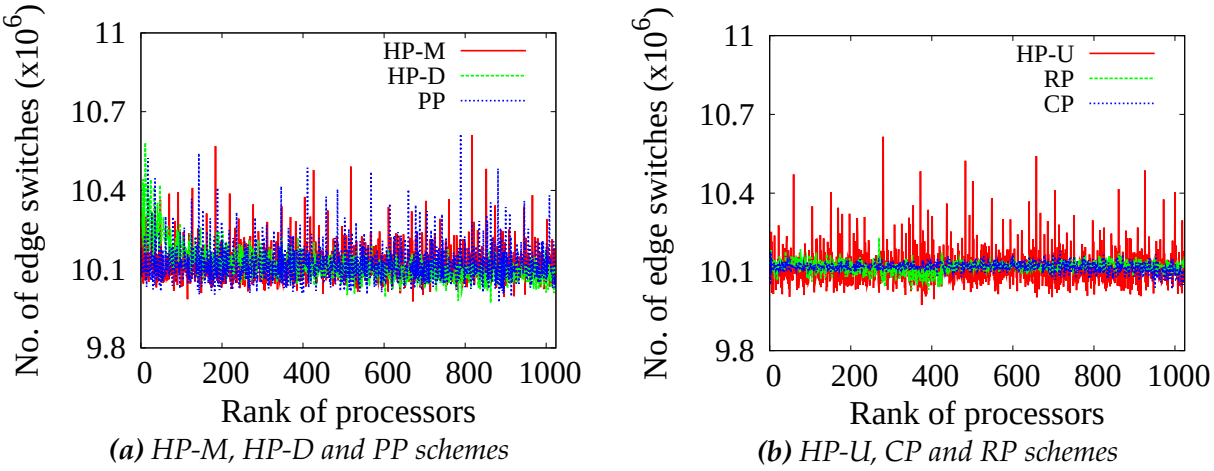
Figure 3.11. Recall that the number of edges gradually changes among the processors with the progress of the edge switch process. As a result, at the completion of the edge switch process, the processors may end up with a number of edges different than the numbers at the beginning of the process. Figure 3.12 shows the distribution of edges at the completion of an edge switch process using different schemes on the Miami graph. The CP and PP schemes show highly skewed distribution of edges compared to the HP and RP schemes. The skewness exhibited in the CP and PP schemes is a combined effect of the following reasons:

- A reduced adjacency list uses the ordering of vertex labels (from 0 to $n-1$) to store an edge (u, v): $N(u)$ stores v if and only if $u < v$.
- The same ordering of vertex labels is used to assign a consecutive subset of vertices to a block.

For example, let (u_1, v_1) be an edge belonging to the block in the highest ranked processor P_{p-1}, participating in an edge switch operation with another edge (u_2, v_2) belonging to the block in P_i ($i < p-1$). There is a probability that both replacing edges (edge e_3 and e_4, or e_5 and e_6 in Figure 3.3) can belong to $N(u_2)$ and $N(v_2)$, which reside in some processors other than P_{p-1}, thus decreasing one edge from the block in P_{p-1} and increasing one edge in the block in P_j ($j \neq p-1$). The occurrence of such a scenario increases for graphs having a high clustering coefficient. Note that Miami is a synthetic yet realistic contact network with maximum, minimum, and average degrees of 425, 1, and 50.4 respectively. It has good clustering among the vertices that is gradually destroyed with the progression of the edge switch process. For the Miami graph, most of the edges in the block belonging to the highest ranked processor are replaced by edges with one end vertex belonging to some other block, thus destroying the clustering among the vertices in the highest ranked processor as well as decreasing the number of edges in the block substantially. As a result, some processors contain a larger number of edges compared to other processors at the end of the edge switch process. Since the number of edge switch operations performed at a processor P_i depends on the number of edges at P_i, the skewness in the number of edges among the processors with the course of the edge switch process results in an imbalanced workload distribution as shown in Figure 3.13 for the Miami graph.

In contrast, the HP and RP schemes do not assign consecutive vertices to a block. Thus a subset of vertices having dispersed vertex labels along with their reduced adjacency lists belong to a block. As a result, the change in the number of edges among the blocks during the edge switch process is significantly less than that of the CP and PP schemes for the Miami graph, leading to a better workload distribution in the HP and RP schemes as shown in Figure 3.13. Hence, all of the HP and RP schemes out-perform the CP and PP

schemes for the Miami graph, which is illustrated in Figure 3.9. Among the RP and HP schemes, RP out-performs the HP schemes, and HP-U out-performs the other HP schemes by a slight margin for the Miami graph.



***(a)** HP-M, HP-D and PP schemes* ***(b)** HP-U, CP and RP schemes*

Figure 3.14: *Distribution of workload (number of edge switch operations) among the processors in different partitioning schemes for the PA-100M graph.*

On the other hand, Figure 3.14 illustrates that the CP scheme exhibits better workload distribution for a Preferential Attachment graph having 100M vertices and 1B edges. The PA graph has a very highly skewed degree distribution, i.e., it has a few very high degree and many low-degree vertices. The maximum, minimum, and average degrees of the PA-100M graph are 55225, 10, and 20, respectively. The CP scheme assigns a consecutive subset of vertices to blocks and uses the degrees of vertices to ensure that all the blocks have an equal number of edges; whereas the HP and RP schemes assign vertices to blocks using only vertex labels; they neither use the degree of vertices nor consider the number of edges already assigned to a block. As a result, the HP and RP schemes assign several high degree vertices to some processors for the PA graph, thus making the initial edge distribution slightly more skewed compared to the CP scheme. Since PA is a random graph having a very low clustering coefficient, the numbers of edges initially assigned to processors vary negligibly with the course of the edge switch process in the CP scheme. As a result, the CP scheme has an advantage of a better initial edge distribution and thus demonstrates a better workload distribution and speedup compared to the other schemes as shown in Figures 3.14 and 3.9 respectively.

A Worse Case Scenario for the HP-D Scheme. Unlike the CP, RP, and PP schemes, one potential disadvantage of the HP schemes is that if there is an adversary aware of the exact hash function being used as the partitioning scheme, the adversary may generate a

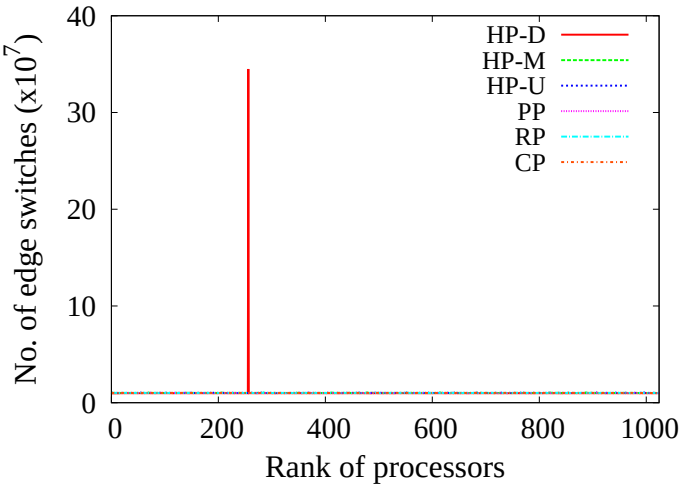


Figure 3.15: *A worse case scenario of distribution of workload (number of edge switch operations) among the processors for the HP-D scheme on the PA-100M graph.*

worse case scenario by artificially manipulating vertex labels of a graph. We simulate such a scenario for the HP-D scheme using 1024 processors. We intentionally reassign vertex labels of the PA-100M graph in such a way that all of the n/p highest degree vertices are assigned to a single processor, say P_k. Thus P_k has a very high number of edges compared to other processors despite having an equal number of vertices among the processors. As a result, P_k performs a substantially higher number of edge switch operations compared to other processors as shown in Figure 3.15 (in this example, P_k is the processor with rank 256), whereas other schemes show good performance by executing faster on the same graph as shown in Figure 3.16. An adversary can generate a similar worse case scenario for the HP-M scheme as well.

The Advantage of the HP-U and RP Schemes. Universal hashing randomly selects a hash function from a large set of hash functions. As a consequence, there is no way for an adversary to know in advance exactly which hash function will be used. Therefore, the HP-U scheme overcomes the drawbacks of the HP-D and HP-M schemes. The RP scheme also has the same advantage of randomly assigning vertices to blocks. In addition, HP-U and RP demonstrate good speedups for all types of graphs and out-perform the other schemes in many cases. The HP-U does not require reordering the vertex labels after partitioning to eliminate the $O(n)$ space requirement for storing the mapping information of vertices like the PP and RP schemes. Although the CP scheme exhibits the best performance for the PA-100M graph, speedups achieved by the HP-U and RP schemes are very close to that of CP, justifying HP-U and RP as good choices in general. The PP scheme exhibits the poorest performance among all the schemes because of poor load distribution among processors.

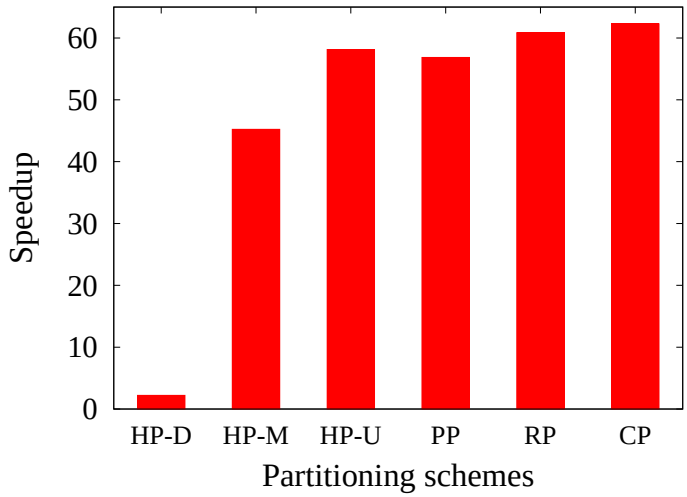


Figure 3.16: *A comparison of speedup of a worse case scenario for the HP-D scheme with other schemes on a PA-100M graph with 1024 processors.*

The Similarity of the Outcomes of the Parallel and Sequential Algorithms and Determining Suitable Step-size. For convenience, we first present the effect of step-size on the parallel algorithm using the CP scheme, and then we discuss the effect of step-size for other schemes. We use visit rate $x = 1$, current calendar time as random seed, $r = 20$ blocks, $p = 1024$ processors, and an average value of ten experiments.

Effect of step-size on the CP scheme. Figure 3.17 shows that better strong scaling is achieved for a larger step-size on the Miami graph. For a particular step-size, error rate remains roughly constant with the increase of processors on the Miami graph, as shown in Figure 3.18. The effects of step-size on speedup and error rate for the Miami graph are shown in Figures 3.19 and 3.20, respectively. Both the speedup and error rate increase with the increase of step-size.

While keeping the error rate to a minimum, we want to achieve as much speedup as possible. From Figure 3.20, we observe that with up to a step-size of $2M$, the error rate between the resultant graphs generated by the sequential and parallel algorithms is roughly the same as the error rate between the resultant graphs generated by two different execution of the sequential algorithm. Hence, $2M$ can be a *suitable* step-size for the Miami graph, since the error rate is minimal, and a good speedup factor of 50 using 1024 processors is achieved at the same time. If we further increase the step-size, both the speedup and error rate increase. For example, using a step-size of $9.4M$, the error rate is a negligible 0.4%; however a higher speedup factor of 62 is achieved using 1024 processors. Figures 3.21 and 3.22 illustrate the effect of step-size on speedup and error rate, respectively, for different graphs. Suitable step-size may vary from graph to graph, depending on the graph size and type.

For example, the error rate is roughly constant for different step-sizes on Erdős-Rényi and LiveJournal graphs, though it varies for the Flickr and Miami graphs as shown in Figure 3.22. A *suitable* step-size for the Flickr, Miami, LiveJournal and Erdős-Rényi graphs can be $1.5M$, $2M$, $4M$, and $8M$ respectively. In general, if we use a lower step-size, say $2M$, for any medium-sized graph (having more than $20M$ edges), we expect to have a very small error rate along with a good speedup. The above experiments show that the sequential

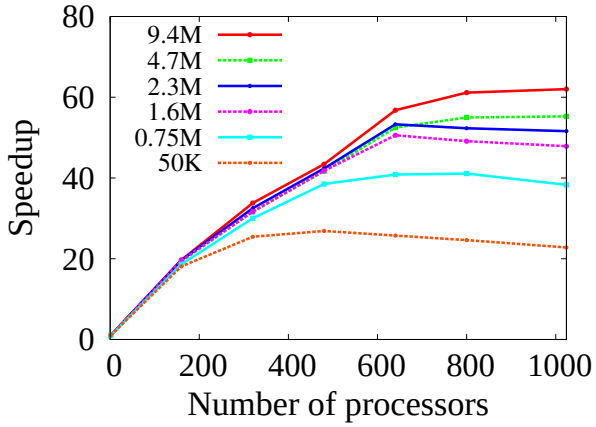


Figure 3.17: *A comparison of strong scaling performance on the Miami graph for different step-sizes: 9.4M, 4.7M, 2.3M, 1.6M, 0.75M, and 50K.*

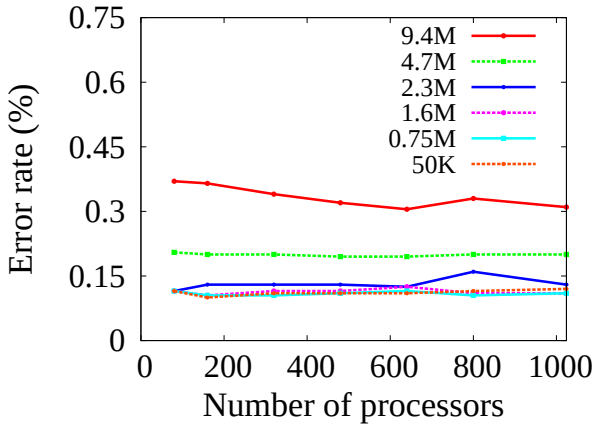


Figure 3.18: *Error rate with increasing number of processors on the Miami graph using different step-sizes: 9.4M, 4.7M, 2.3M, 1.6M, 0.75M, and 50K.*

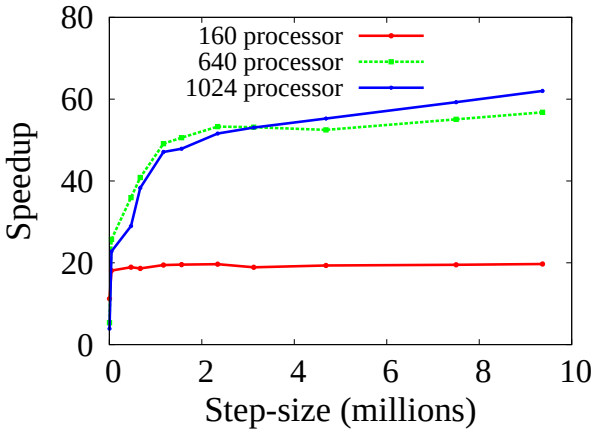


Figure 3.19: *Speedup with increasing step-size on the Miami graph using 160, 640 and 1024 processors.*

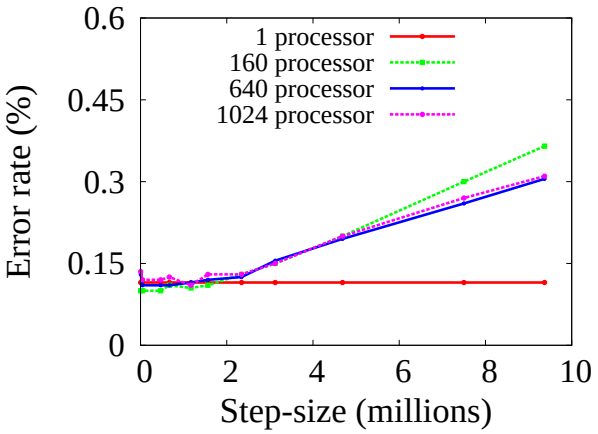


Figure 3.20: *Error rate with increasing step-size on the Miami graph using 1, 160, 640 and 1024 processors.*

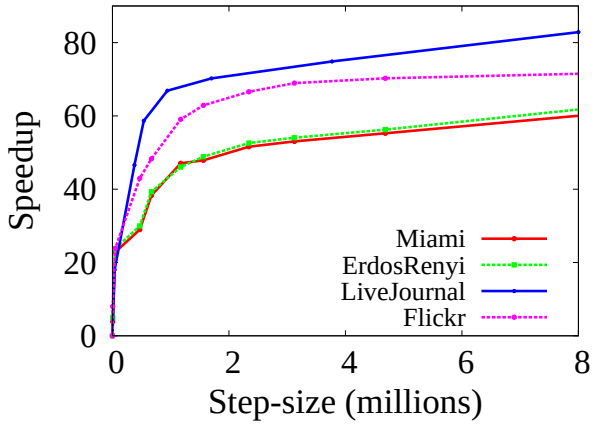


Figure 3.21: *Speedup with increasing step-size for different graphs using 1024 processors.*

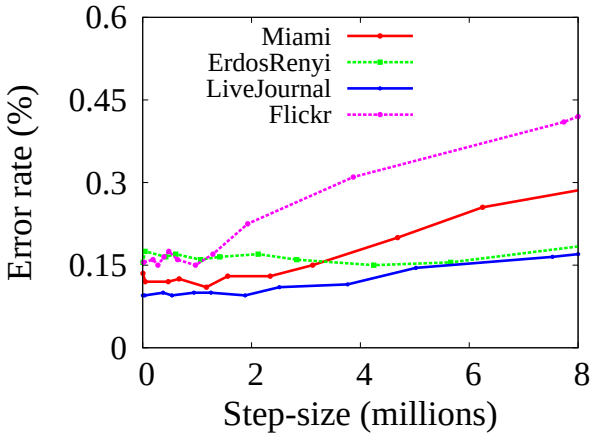


Figure 3.22: *Error rate with increasing step-size for different graphs using 1024 processors.*

and the parallel edge switch processes are similar with a suitable step-size.

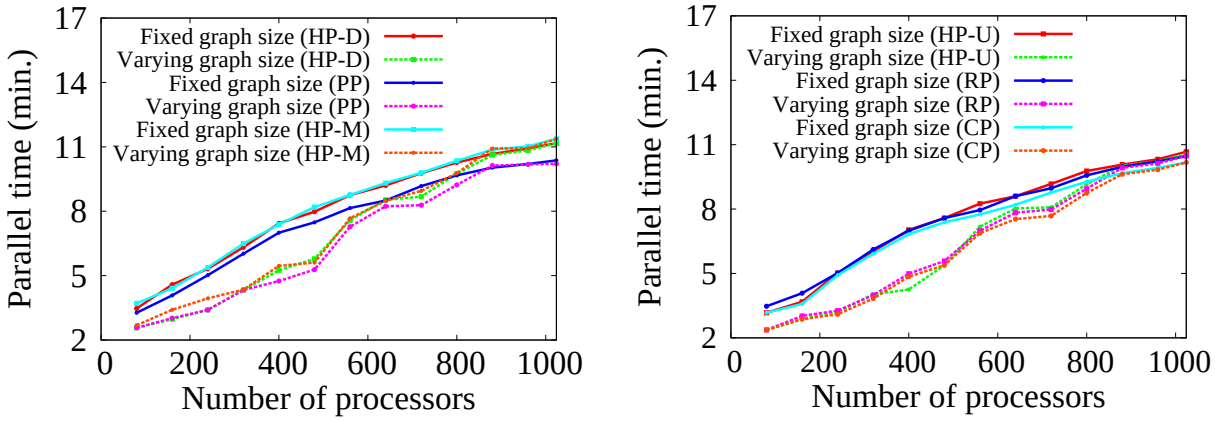
Effect of step-size on other schemes. Table 3.5 shows the error rate comparison of the outcomes of the parallel algorithms using different schemes, with that of the sequential one suggesting that even for performing edge switch operations in one step, the outcomes of the parallel algorithms using the HP and RP schemes are similar to that of the sequential algorithm with a negligible error rate deviation. Since the HP and RP schemes assign vertices dispersedly among the blocks, the number of edges initially belonging to the blocks change negligibly with edge switch operations compared to that of the CP and PP schemes. Hence the HP and RP schemes can perform edge switch operations in only one step with reasonable accuracy, which consequently makes computation faster. As a result, the parallel algorithms using the HP and RP schemes no longer need a suitable step-size. In contrast, finding a suitable step-size is important for the CP and PP schemes to obtain a close approximation of the outcome of the sequential algorithm.

Weak Scaling. Figure 3.23 shows weak scaling comparison of different schemes on PA graphs. In one experiment, we increase the graph size with the increase of processors and use Preferential Attachment graphs with $(p \times 0.1M)$ vertices and an average degree of 20. In another experiment, we use a fixed Preferential Attachment graph with $102.4M$ vertices and $1.024B$ edges. In both experiments, we use $t = p \times 10M$ and step-size $= t/1000$. Ideally, the parallel runtime should remain constant. However, in practice the communication increases with the increase in number of processors, leading to a higher runtime. Our algorithm shows good weak scaling as the runtime increases linearly in both the cases.

How Do Network Properties Change with Switching Edges? We also analyze how some

Table 3.5: *Error rate comparison of the outcomes of the parallel algorithms using different partitioning schemes with that of the sequential algorithm for different graphs. We use the average of ten experiments.*

	Error rates (%) in sequential and parallel algorithms										
		Parallel									
		Using 1 step				Using 100 steps					
Networks	**Sequential**	**HP-D**	**HP-M**	**HP-U**	**RP**	**HP-D**	**HP-M**	**HP-U**	**RP**	**CP**	**PP**
Miami	0.117	0.118	0.123	0.117	0.117	0.111	0.127	0.123	0.120	0.164	0.175
SmallWorld	0.112	0.100	0.112	0.119	0.116	0.106	0.118	0.109	0.115	0.115	0.121
LiveJournal	0.116	0.117	0.118	0.117	0.117	0.116	0.116	0.116	0.1177	0.115	0.126



(a) *HP-D, PP and HP-M schemes* ***(b)*** *HP-U, RP and CP schemes*

Figure 3.23: *Weak scaling comparison of the parallel algorithms using various partitioning schemes with fixed and varying size PA graphs. In one experiment, we use a fixed graph having* $102.4M$ *vertices and* $1.024B$ *edges while in the other experiment, we increase (or vary) graph size with the increase of processors. The varying graphs have* $(p \times 0.1M)$ *vertices and an average degree of 20, where* p *is the number of processors. For both experiments, we use* $t = p \times 10M$ *and step-size* $= t/1000$.

network properties change with edge switch operations by the sequential and parallel algorithms. We use the Miami, LiveJournal, and Flickr graphs, and vary the visit rate from 0.1 to 1. Figures 3.24 and 3.25 show that the average clustering coefficient and average shortest path distance of a graph change in the same way with edge switches by the sequential and parallel algorithms. Average clustering coefficient measures the tendency of the vertices to cluster together or create tightly knit groups, which is a pervasive phenomena in many social networks where people with common friends tend to be friends themselves.

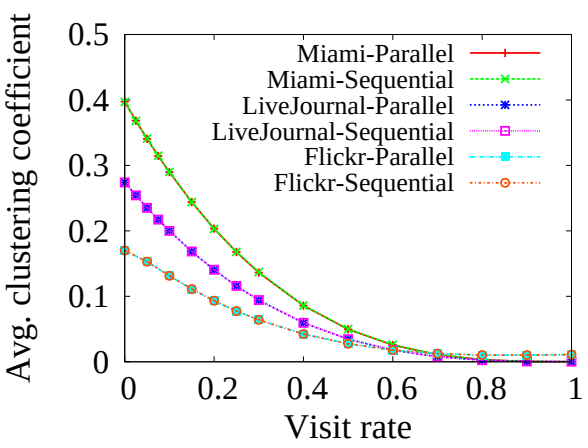


Figure 3.24: *Average clustering coefficient changes similarly with edge switch operations by the sequential and parallel algorithms.*

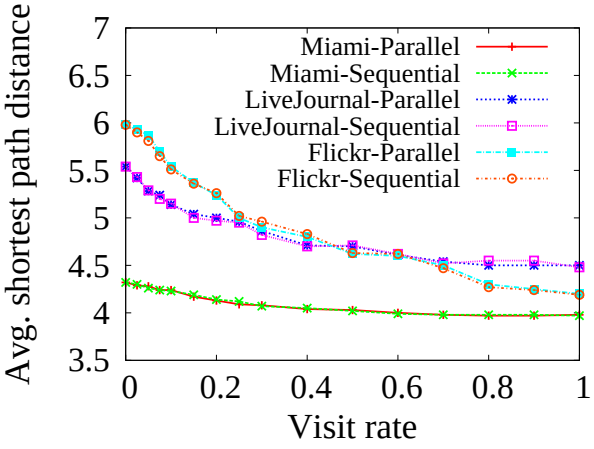


Figure 3.25: *Average shortest path distance changes similarly with edge switch operations by the sequential and parallel algorithms.*

With the progress of an edge switch process, the edges are replaced by random edges. Therefore, the clustering among the vertices gets destroyed rapidly, which eventually reaches very close to 0 for a visit rate of 1.0. As the edges are switched, the vertices further get connected by shorter paths, thus reducing the average shortest path distance as well. For both properties, the changes by the sequential and parallel algorithms are very similar; in fact, they overlap with each other, and it is difficult to distinguish them in the figures.

Generating Assortative Networks by Switching Edges. We demonstrate how edge switching can be used to generate assortative networks. In a labeled network, each vertex u has an associated attribute $L(u)$. Such attributes can be, for example, the age of people in a contact network or the degree of the vertices. Adding vertex attribute constraints with edge switch leads to many interesting problems. Xulvi-Brunet et al. [115] proposed one such algorithm to produce assortative mixing to a desired degree by imposing constraints on vertex attributes during an edge switch process. *Assortative mixing* is an important network feature measuring the tendency of vertices to associate with similar or dissimilar vertices and is quantified by a metric named the assortative coefficient (r) [84, 85]. In other words, assortativity measures the correlation of vertices based on vertex attributes. A network is called an *assortative network* if $r > 0$.

In this demonstration, we use the degree of a vertex as its attribute, i.e., $L(u) = d_u$. The level or extent of assortative mixing is controlled by a parameter p ($0 \leq p \leq 1$). Then an edge switch operation selects two edges randomly with the four end vertices having different degrees in general. The four vertices are ordered based on their degrees. Then

with probability p, an edge switch operation connects the two higher degree vertices with an edge and the two lower degree vertices with another edge. With probability $(1 - p)$, the edges are switched randomly. This algorithm generates a random network with parameter $p = 0$. With the increase of p, the assortativity increases and it reaches a maximum value for $p = 1$ (see [115] for a good discussion). We apply the principle of [115] with our parallel algorithm of edge switch to generate assortative networks and present the results below.

Figure 3.26 shows how the assortative coefficient changes with the edge switch process for different values of p on the Miami graph. For $p = 1$, we obtain a maximum assortative coefficient value of 0.999992, beyond which the assortativity does not increase due to the restriction imposed by the degree distribution. Figure 3.27 shows the speedup obtained by the parallel algorithm using the HP-U scheme for performing $300M$ edge switch operations on the Miami graph.

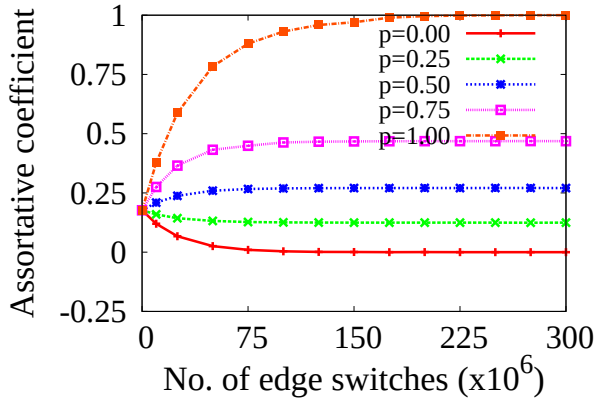


Figure 3.26: *Change of assortative coefficient (considering the degree of a vertex as its attribute) with the edge switch process on the Miami graph. The parameter p is varied from 0 to 1.*

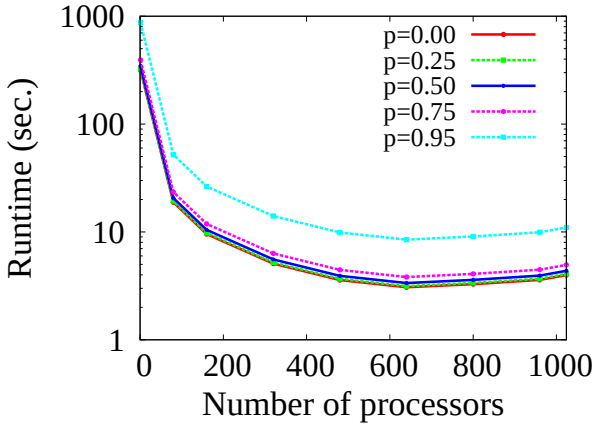


Figure 3.27: *Speedup gained by the parallel algorithm of edge switch for different values of p. The algorithm performs $300M$ edge switch operations on the Miami graph.*

Synopsis of the Experimental Results. All of the partitioning schemes demonstrate reasonably good performance. Below is a summary of the results.

- The hash-based and random partitioning schemes exhibit better performance for many graphs because of a well-balanced workload distribution.
- The HP and RP schemes can perform edge switch operations in only one step with reasonable accuracy, thus eliminating the need for performing edge switch operations

in a number of steps.

- There is a possibility of a worse case scenario arising from the HP-D and HP-M schemes that may slow down the algorithm significantly. The HP-U scheme overcomes this drawback by randomly choosing a hash function from a large set of hash functions. Like the HP-U scheme, the CP, PP, and RP schemes are also not vulnerable to adversaries for generating worse case scenarios.
- The CP scheme shows good performance with some computation overhead by performing edge switch operations with a suitable step-size for all types of graphs, and in some cases (e.g., PA-100M) out-performs the other schemes.
- The PP scheme exhibits the poorest performance among all the schemes due to poor workload distribution. It also requires performing edge switch operations with a suitable step-size.
- Unlike the HP and CP schemes, the PP and RP schemes need to reassign the vertex labels after partitioning to get rid of the $O(n)$ space requirement at each processor to store the mapping of vertices to blocks.

3.5 Parallel Algorithm for Computing Binomial and Multinomial Distribution

In this section, we present a parallel algorithm for computing a multinomial distribution. First, we briefly review the current state-of-the-art sequential algorithm.

3.5.1 Sequential Algorithm for Computing Multinomial Distribution

One simple approach for computing multinomial random variables is to perform N independent trials, where the outcome of each trial can be $0, 1, \ldots, \ell - 1$ with probability $q_0, q_1, \ldots, q_{\ell-1}$, respectively. This algorithm takes at least $\Omega\left(N \log \ell\right)$ time. An efficient state-of-the-art algorithm is the *conditional distributed method* [37], which runs in $O\left(N\right)$ time. This method generates multinomial random variables $\langle X_0, X_1, \ldots, X_{\ell-1}\rangle$ by iteratively

generating ℓ binomial random variables:

$$X_i \sim \mathcal{B}\left(N - \sum_{j=0}^{i-1} X_j, \frac{q_i}{1 - \sum_{j=0}^{i-1} q_j}\right) \tag{3.8}$$

The inverse transformation method (BINV) [62] is the best-known algorithm for computing binomial random variables. To generate a binomial random variable Y with parameters N and q, it takes $O(Y)$ time. Note that the expected value of Y is Nq.

The algorithms for the inverse transformation method (BINV) [62] to generate binomial random variables and for the conditional distributed method [37] to generate multinomial random variables are shown in Algorithms 3 and 4, respectively. For additional details, see [62] and [37].

Algorithm 3: Sequential Binomial

```
1  Procedure BINOMIAL(N, q)
2      if q = 1 then
3          return N
4      i ← 0                                   // i is the binomial random variable
5      Generate u ~ U(0, 1) uniformly at random
6      Q ← (1 − q)^N
7      S ← Q
8      while S < u do
9          i ← i + 1
10         Q ← Q ((N − i + 1)/i) (q/(1 − q))
11         S ← S + Q
12     return i
```

The conditional distributed method shown in Algorithm 4 runs in $\sum_{i=0}^{\ell-1} O(X_i) = O(N)$ time. In the next section, we present an efficient parallelization of Algorithm 4.

3.5.2 Parallel Algorithm for Computing Multinomial Distribution

Based on the conditional distributed method shown in Algorithm 4, we propose a parallel algorithm for computing multinomial distribution $X \sim \mathcal{M}(N, q)$, where q denotes

Algorithm 4: Sequential Multinomial

```
1  Procedure SEQUENTIAL-MULTINOMIAL(N, q_0, q_1, ..., q_{ℓ-1})
2      X_s ← 0
3      Q_s ← 0
4      for i = 0 to ℓ − 1 do
5          if Q_s < 1 then
               /* X_i is the multinomial random variable for outcome i   */
6              X_i ← BINOMIAL(N − X_s, q_i / (1 − Q_s))
7              X_s ← X_s + X_i
8              Q_s ← Q_s + q_i
9          else
10             X_i ← 0
11     return ⟨X_0, X_1, ..., X_{ℓ-1}⟩
```

probability vector $\langle q_0, q_1, \ldots, q_{\ell-1}\rangle$. One tempting approach to parallelize the conditional distributed method is to distribute the generation of X_i, $0 \leq i < \ell$ (Line 6 of Algorithm 4) among the processors. However, a difficulty arises from the sequential nature of computing X_i due to the dependency of X_i on X_j $(0 \leq j \leq i-1)$ for all i, j. We overcome this difficulty by exploiting some properties of binomial and multinomial random variables, as described below.

Let N_i, for $0 \leq i < k$, be some integers such that $N = \sum_{i=0}^{k-1} N_i$. If $X_i \sim \mathcal{B}(N_i, q)$, then

$$X = \sum_{i=0}^{k-1} X_i \sim \mathcal{B}\left(\sum_{i=0}^{k-1} N_i, q\right) = \mathcal{B}(N, q) \tag{3.9}$$

The above property of the binomial random variables leads to the following property of the multinomial random variables. If

$$\left\langle X_{0,i}, X_{1,i}, \ldots, X_{\ell-1,i}\right\rangle \sim \mathcal{M}\left(N_i, q_0, q_1, \ldots, q_{\ell-1}\right)$$

for $0 \leq i < k$, then

$$\langle X_0, X_1, \ldots, X_{\ell-1}\rangle \sim \mathcal{M}\left(N, q_0, q_1, \ldots, q_{\ell-1}\right) \tag{3.10}$$

where $X_j = \sum_{i=0}^{k-1} X_{j,i}$ for $0 \leq j < \ell$ and $N = \sum_{i=0}^{k-1} N_i$.

Now we describe the parallel algorithm for computing multinomial distribution, which uses the above property. First, we explain the case of $p = \ell$. Our algorithm divides the number of trials N into p almost equal small number of trials N_i, and assigns N_i to P_i. Then each processor P_i computes the multinomial distribution of N_i using the same probability vector q. At the end, the results of all the processors are aggregated. The pseudocode is given in Algorithm 5, where processor P_i holds the multinomial random variable X_i at the end of the computation.

Algorithm 5: Parallel Multinomial with $p = \ell$

1 **Procedure** PARALLEL-MULTINOMIAL$(N, q_0, q_1, \ldots, q_{\ell-1})$
  /* Each processor P_i executes the following in parallel: */
2 **if** $i < (N \mod p)$ **then**
3 $N_i \leftarrow \left\lfloor \frac{N}{p} \right\rfloor + 1$
4 **else**
5 $N_i \leftarrow \left\lfloor \frac{N}{p} \right\rfloor$
6 $\langle X_{0,i}, X_{1,i}, \ldots, X_{\ell-1,i} \rangle \sim$ SEQUENTIAL-MULTINOMIAL$(N_i, q_0, q_1, \ldots, q_{\ell-1})$
7 Send $X_{j,i}$ to processor P_j // $0 \le j \le p-1$
8 Upon receiving $X_{i,k}$ from every processor P_k: // $0 \le k \le p-1$
9 $X_i \leftarrow \sum_{k=0}^{p-1} X_{i,k}$ // Processor P_i contains the multinomial random variable X_i

For $p \neq \ell$, the algorithm is the same up to the multinomial distribution computation of N_i at P_i, i.e., Lines 2–6 of Algorithm 5. The only difference is how the generated multinomial random variables will be stored among the processors. The variables can be stored in many ways, e.g., all the X_is can be gathered to the root processor P_0, or they (X_is) can be distributed among the processors in a round-robin fashion, i.e., assigning X_i to processor $P_{(i \mod p)}$, etc. X_i is always computed by summing up all the $X_{i,k}$s ($0 \le k < p$), after receiving them from all processors.

The parallel computation is almost perfectly load balanced among the processors since each processor computes a multinomial distribution of $\frac{N}{p}$ independently, taking $O\left(\frac{N}{p}\right)$ time. The communication cost at the end takes $O\left(\ell \log p\right)$ time. Hence, the time complexity of this algorithm is $O\left(\frac{N}{p} + \ell \log p\right)$. The algorithm is almost perfectly parallelized because the number of processors, p (which is in the range of hundreds or at most thousands), and the number of outcomes ℓ, are significantly smaller than the number of trials N (which is in the range of billions), in a general case. Algorithm 5 computes a binomial distribution

for $\ell = 2$.

During binomial random variable generation, the computation of $(1-q)^N$ (Line 6 of Algorithm 3) results in numerical underflow for large values of N, e.g., billions. Using the *long double* data type cannot solve this underflow problem for large N. In addition, some round-off errors may appear. We deal with these difficulties by using the property of the binomial distribution again, i.e., we divide N into small N_is such that $\sum_i N_i = N$, compute X using equation (3.9). The upper threshold value of N_i is set such that no underflow occurs, that is,

$$(1-q)^{N_i} \geq z \tag{3.11}$$

$$N_i \leq \frac{-\log z}{\log(1-q)} \leq \frac{-\log z}{2q} \tag{3.12}$$

where z is the smallest positive real number that can be represented by the data type (e.g., float, double) used and $q < 1$.

3.5.3 Performance Analysis of the Parallel Algorithm

In this section, the performance of the parallel algorithm for multinomial distribution is demonstrated by strong scaling and weak scaling.

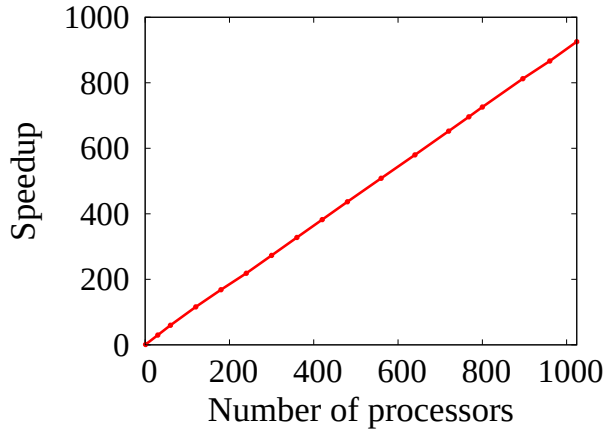


Figure 3.28: *Strong scaling of the parallel algorithm of multinomial distribution using $N = 10000B$, $\ell = 20$ and $q_i = \frac{1}{\ell}$.*

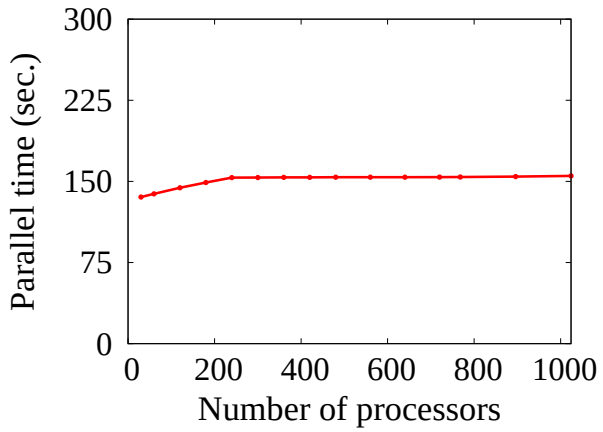


Figure 3.29: *Weak scaling of the parallel algorithm of multinomial distribution using $N = p \times 20B$, $\ell = p$ and $q_i = \frac{1}{\ell}$.*

Strong Scaling. The strong scaling of the parallel algorithm is illustrated in Figure 3.28. We keep the problem size fixed ($N = 10000B$, $\ell = 20$ and $q_i = \frac{1}{\ell}$), and achieve a speedup of

925 using 1024 processors. The speedup increases almost linearly with the increase in the number of processors. The parallel algorithm can compute a multinomial distribution of $10000B$ in 71 seconds using 1024 processors.

Weak Scaling. Figure 3.29 shows the weak scaling of our parallel algorithm. We use $\ell = p$ (i.e., the number of processors), $N = p \times 20B$ (i.e., $20B$ per processor), and equal probability values, $q_i = \frac{1}{\ell}$. The parallel run time is almost constant indicating a very good weak scaling.

3.6 Parallel Algorithm for 1-Flipper

In this section, we present a shared memory parallel algorithm for 1-Flipper, which maintains the connectivity of the graph while switching edges. First, we briefly review the sequential algorithm.

3.6.1 Sequential Algorithm

The algorithms for switching edges presented earlier in this chapter do not maintain the connectivity of a given graph, i.e., a connected graph may become disconnected after switching edges and vice versa. However, preserving the connectivity is desired in many dynamic graphs such as the communication and peer-to-peer networks. To do so, Mahlmann et al. [78] introduced k-Flipper operation in a graph, which selects a path of $k + 2$ edges and interchanges the end vertices of the path. This operation preserves the connectivity of the graph in addition to the degree of the vertices. They used these operations in modeling peer-to-peer networks to provide high robustness by maintaining the connectivity of the networks. It has important applications in generating random regular graphs and transforming them into expander graphs [78, 117, 118]. It has also been shown in [78] that every connected regular graph can be reached by a series of these operations for all $k \geq 1$.

A 1-Flipper operation selects a path of 3 edges and swaps the end vertices of the path, as shown in Figure 3.30. First, a hub edge (v_1, v_2) is selected randomly from the given graph, and then two flipping edges (u_1, v_1) and (u_2, v_2) are selected randomly from the adjacency lists of v_1 and v_2, respectively, and the end vertices of the flipping edges are swapped such that they do not produce loops or parallel edges. This operation is repeated as many times as required. The pseudocode of the algorithm to perform t 1-Flipper operations in a given graph G is given in Algorithm 6. In case of a disconnected graph, a 3-length path is always selected from the same connected component, and the distribution of the

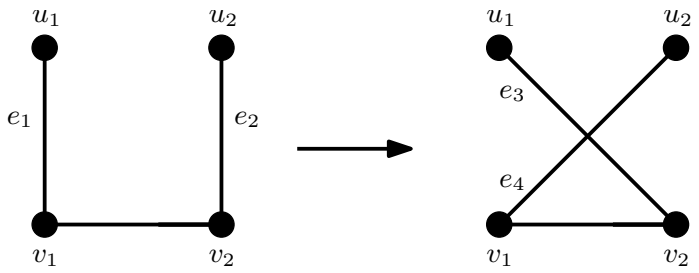


Figure 3.30: *A 1-Flipper operation replaces two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ by new edges $e_3 = (u_1, v_2)$ and $e_4 = (u_2, v_1)$ by maintaining the connectivity of the graph through the hub edge (v_1, v_2).*

Algorithm 6: Sequential 1-Flipper

```
1  Procedure Flip-Edges(G, t)
2      i ← 1
3      E′ ← E
4      while i ≤ t do
5          (v1, v2) ← a uniform random edge in E′          // (v1, v2) is a hub edge
6          u1 ← a uniform random vertex in N(v1)\{v2}      // (u1, v1) is a flipping edge
7          u2 ← a uniform random vertex in N(v2)\{v1}      // (u2, v2) is a flipping edge
8          if u1 ∈ N(v2) or u2 ∈ N(v1) then                // parallel edge
9              continue
10         Replace (u1, v1) and (u2, v2) in E′ by (u1, v2) and (u2, v1) respectively
11         i ← i + 1
```

connected components remains the same after performing 1-Flipper operations.

3.6.2 Parallel Algorithm

In a shared memory parallel algorithm, the graph is typically stored in the shared address space and all the cores can access to the shared global address space. The parallel algorithm assigns an equal number of 1-Flipper operations to each core and the cores simultaneously perform the 1-Flipper operations. Due to the simultaneous nature of the parallel algorithm, the algorithm needs to deal with the following challenges: (***i***) Parallel edges can be created in a new way that does not arise in the sequential algorithm as we described before. We deal with this issue by keeping track of the potential new edges. (***ii***) Any edge of a 3-length path participating in a 1-Flipper operation can not simultaneously participate in another 1-Flip operation at the same time, i.e., they can not be selected by another core

Algorithm 7: Parallel 1-Flipper

Procedure PARALLEL-FLIP-EDGES(G, t)
  /* Each core P_i executes the following in parallel: */
  if $i < (t \mod p)$ **then**
    $t_i \leftarrow \left\lfloor \frac{t}{p} \right\rfloor + 1$
  else
    $t_i \leftarrow \left\lfloor \frac{t}{p} \right\rfloor$
  $j \leftarrow 1$
  $E' \leftarrow E$
  $B \leftarrow \phi$ // set of busy edges participating in 1-Flipper operations
  $P \leftarrow \phi$ // set of potential new edges
  while $j \leq t_i$ **do**
    $(v_1, v_2) \leftarrow$ a uniform random edge in E' // (v_1, v_2) is a hub edge
    $u_1 \leftarrow$ a uniform random vertex in $N(v_1) \backslash \{v_2\}$ // (u_1, v_1) is a flipping edge
    $u_2 \leftarrow$ a uniform random vertex in $N(v_2) \backslash \{v_1\}$ // (u_2, v_2) is a flipping edge
    if $(v_1, v_2) \in B$, $(u_1, v_1) \in B$, $(u_2, v_2) \in B$, $(u_1, v_2) \in P$, $(u_2, v_1) \in P$, $u_1 \in N(v_2)$, *or* $u_2 \in N(v_1)$ **then**
      continue
    $B \leftarrow B \bigcup \{(u_1, v_1), (u_2, v_2), (v_1, v_2)\}$
    $P \leftarrow P \bigcup \{(u_1, v_2), (u_2, v_1)\}$
    Replace (u_1, v_1) and (u_2, v_2) in E' by (u_1, v_2) and (u_2, v_1) respectively
    $B \leftarrow B \backslash \{(u_1, v_1), (u_2, v_2), (v_1, v_2)\}$
    $P \leftarrow P \backslash \{(u_1, v_2), (u_2, v_1)\}$
    $j \leftarrow j + 1$

when they are participating in a 1-Flipper operation. We deal with this issue by keeping track of which edges are participating in 1-Flipper operations at a given time and always selecting edges which are not participating in any other simultaneous 1-Flipper operations. (***iii***) Concurrent writing by multiple cores can lead to race conditions, and we deal with this issue by atomic update operations. The pseudocode of the parallel algorithm is shown in Algorithm 7.

Performance Analysis

In this section, we present the experimental setup and strong scaling performance of the parallel algorithm for 1-Flipper.

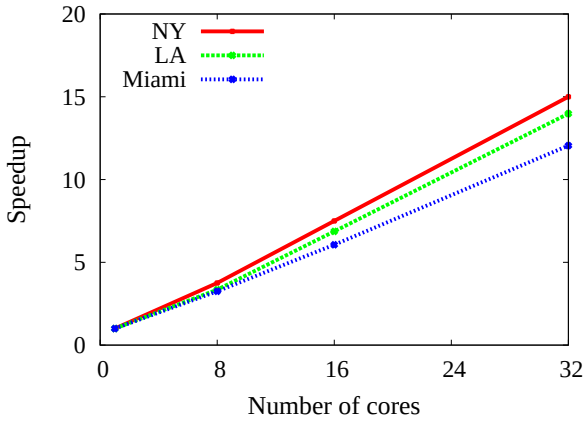


Figure 3.31: *Strong scaling of the parallel algorithm for 1-Flipper.*

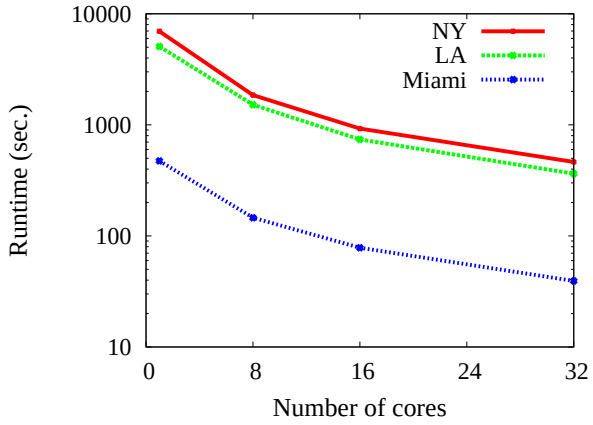


Figure 3.32: *Runtime of the parallel algorithm for 1-Flipper.*

Experimental Setup. We use a 32-core Haswell-EP E5-2698 v3 2.30GHz (3.60GHz Turbo) dual processor node with 128GB of memory, 1TB internal hard drive, and QLogic QDR InfiniBand adapter. We use OpenMP version 3.1 and GCC version 4.7.2 for implementation.

Strong Scaling. We use the NewYork (NY), LosAngeles (LA), and Miami network in the experiments. The strong scaling and runtime of the parallel algorithm are shown in Figures 3.31 and 3.32, respectively. The speedups increase almost linearly with the increase in the number of cores, and we achieve a maximum speedup of 15 with 32 cores on the NY network.

3.7 Conclusion

We presented distributed memory parallel algorithms for switching edges in large networks. They can be used in studying various properties of large dynamic networks as well as in generating massive scale random graphs. The algorithms scale well to a large number of processors and exhibit good speedup. We also presented the trade-offs of several partitioning schemes. We demonstrated an application of our parallel algorithms to generate assortative networks. In addition, we developed a parallel algorithm for generating multinomial random variables that is almost perfectly parallelized. This algorithm can be of independent interest and prove useful in parallelizing many other stochastic processes. We also presented a shared memory parallel algorithm for 1-Flipper, which can generate random graphs by switching edges and maintaining the connectivity of the given graph. We believe that the parallel algorithms will contribute significantly when dealing with big

data, one of the most challenging problems in today's research world.

Acknowledgment

We thank our external collaborators, members of the Network Dynamics and Simulation Science Laboratory (NDSSL), and anonymous reviewers for their suggestions and comments. We are grateful to Anil Vullikanti for interesting discussions and helpful comments on a draft of this work. We also sincerely thank Maureen Lawrence-Kuether and Jim Walke for proof-reading this chapter. This work has been partially supported by DTRA CNIMS Contract HDTRA1-11-D-0016-0001, DTRA Grant HDTRA1-11-1-0016, NSF NetSE Grant CNS-1011769, NSF SDCI Grant OCI-1032677, NIH MIDAS Grant 5U01GM070694-11, NSF DIBBs Grant ACI-1443054 and NSF Big Data Grant IIS-1633028.

Chapter 4

Efficient Algorithms for Assortative Edge Switch in Large Labeled Networks

4.1 Introduction

Networks are simple representations of many complex real-world systems. Assortative edge switch is an important problem in the analysis of such networks and has many real-world applications. Assuming each vertex u has an associated label $L(u)$, an assortative edge switch operation selects two edges (a, b) and (c, d) randomly, and replaces them with edges (a, d) and (c, b), respectively, if $L(a) = L(c)$ and $L(b) = L(d)$. For each vertex u, the degree as well as the distribution of the labels of adjacent vertices of u remain invariant with an assortative edge switch process. Such labels can be discrete characteristics (e.g., language, race, and gender in social networks) or scalar properties (e.g., age and degree) of the vertices.

An assortative edge switch operation preserves the *assortative mixing* of a given network, which is a fundamental and important network feature measuring the tendency of vertices to associate with similar or dissimilar vertices and is quantified by a metric named the *assortative coefficient* [27, 84, 85, 88]. In other words, assortativity measures the correlation of vertices based on vertex labels. Newman [84] showed that assortative mixing is a pervasive phenomenon found in many real-world networks and has a profound impact on the structural properties and functionalities of the networks. For example, social (e.g., co-authorship) networks exhibit positive assortativity—more association among similar types of vertices—whereas technological (e.g., Internet and WWW) and biological (e.g., food-web) networks show negative assortativity—more association among dissimilar types of vertices [84]. It helps us to answer interesting questions such as is the friendship, sexual,

or marriage pattern in a social network affected by the language, race, age, gender, or other characteristics [85]? If they are correlated, how strong is the correlation? Some other major applications include finding insights about communities and clustering [85], epidemic behavior [80], resilience [95], and evolution [99] of networks.

Assortative edge switch can be used to assess and analyze the sensitivity of mixing patterns and network structural properties such as clustering coefficient, community structure, and average shortest path distance on dynamics over a network, such as disease dynamics over a social contact network [44, 82]. Random network models often do not capture many structural properties (e.g., assortative mixing) of real-world networks; as a result, to be more realistic, modeling random networks with a prescribed assortative coefficient has gained popularity in the research community [27, 81], and many of these random networks are generated from a joint degree matrix [13]. Assortative edge switch can be combined with regular edge switch process to generate random networks with a prescribed assortative coefficient from a given network [115].

NetworkX [56] has a sequential implementation of regular edge switch; however, it does not have an implementation of assortative edge switch. A parallel algorithm for regular edge switch has been presented in Chapter 3. However, no effort was given to design a parallel algorithm for assortative edge switch in a network. Although the algorithm in Chapter 3 can be applied to perform assortative edge switch operations in a network, it can lead to an extremely slow and inefficient algorithm. To perform an edge switch operation using the algorithm in Section 3.4, the edges are selected randomly from the entire network irrespective of the vertex labels, which can result in many failed attempts for an assortative edge switch operation due to dissatisfying the constraint on the vertex labels. As a result, we need an entirely new algorithmic approach.

In this chapter, we first present a novel sequential algorithm for assortative edge switch; then we present a parallel algorithm based on our sequential algorithm. The dependencies among successive assortative edge switch operations and the requirement of keeping the network simple as well as maintaining the same assortativity during the assortative edge switch process pose significant challenges in designing a parallel algorithm, which in turn requires complex synchronization and communication among the processors. Moreover, achieving a good speedup through a well-balanced load distribution among the processors seems to be a non-trivial challenge for this problem. The performance of the algorithm significantly depends on the size of the network as well as on the distribution of the labels of the vertices. For various network size and diverse distribution of labels, the parallel algorithm provides speedup ranging between 68 and 772 with 1024 processors. To the best of our knowledge, these are the first non-trivial efficient sequential and parallel algorithms for the problem.

The remainder of the chapter is organized as follows. The preliminaries and data sets used in the chapter are briefly described in Section 4.2. The problem of assortative edge switch and the sequential algorithm are discussed in Section 4.3. We present the parallel algorithm along with the performance analysis in Section 4.4. Finally, we conclude in Section 4.5.

4.2 Preliminaries

Below are the notations, definitions, and data sets used in this chapter.

4.2.1 Notations and Definitions

We are given a simple, labeled network $G = (V, E, L)$, where V is the set of vertices, E is the set of edges, and $L : V \rightarrow \mathbb{N}_0$ is the label function. There are a total of $n = |V|$ vertices with vertex ids $0, 1, 2, \ldots, n-1$ and $m = |E|$ edges in G. Each vertex $u \in V$ has an associated label $L(u)$. There are a total of ℓ distinct vertex labels with label-ids $0, 1, 2, \ldots, \ell - 1$, i.e., for each $u \in V$, $L(u) \in [0, \ell - 1]$. The adjacency list and degree of a vertex $u \in V$ are denoted as $N(u)$ and d_u, respectively. We use K, M and B to denote thousands, millions, and billions, respectively; e.g., $1B$ stands for one billion. For the parallel algorithm, let there be P processors with ranks $0, 1, 2, \ldots, P-1$, where P_i denotes the processor with rank i.

Assortative Edge Switch. An assortative edge switch operation imposes an extra constraint on the labels of the end vertices of the two selected edges in addition to the regular edge switch constraints. That is, it randomly selects two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ from E, and replaces them by new edges $e_3 = (u_1, v_2)$ and $e_4 = (u_2, v_1)$ (see Figure 4.1), if the following constraints are satisfied.

- ***Constraint 1:*** It does not create self-loops or parallel edges, i.e., $u_1 \neq v_2$, $u_2 \neq v_1$, $u_1 \notin N(v_2)$, $v_2 \notin N(u_1)$, $u_2 \notin N(v_1)$ and $v_1 \notin N(u_2)$.
- ***Constraint 2:*** The two edges have the same end vertex labels, i.e., $L(u_1) = L(u_2)$, and $L(v_1) = L(v_2)$.

Visit Rate. Some edges of the given network G are changed (or visited) due to assortative edge switch operations, and some edges do not participate in any such operations and remain unchanged (or unvisited). We define the *visit rate* (x) as the fraction of edges of G that have been changed by a sequence of assortative edge switch operations, i.e.,

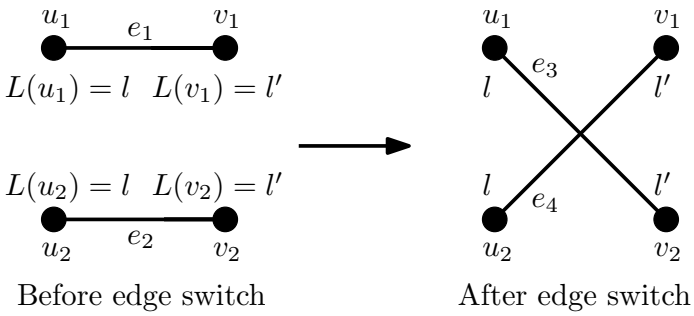


Figure 4.1: *An assortative edge switch operation replaces two randomly selected edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ by new edges $e_3 = (u_1, v_2)$ and $e_4 = (u_2, v_1)$, if $L(u_1) = L(u_2)$ and $L(v_1) = L(v_2)$.*

$x = \frac{m'}{m}$, where m' is the number of edges changed due to assortative edge switches. To achieve a given visit rate x, the expected number of assortative edge switch operations τ to be performed in G is $\tau = \frac{1}{2} m \ln m$ for $x = 1$, and $\tau = -\frac{1}{2} m \ln(1 - x)$ for $0 < x < 1$ (see Section 3.3.1 for detail).

4.2.2 Data Sets

We use both artificial and real-world networks for the experiments. A summary of the networks is given in Table 4.1. New York (NY), Los Angeles (LA), Miami, and Sweden are synthetic, yet realistic, social contact networks [12]. Each vertex represents a person in the corresponding city or country and has an associated label denoting the age of the individual, and each edge represents any 'physical' contact between two persons within a 24-hour time period. Orkut [1] is an online social network, Facebook [1] is an anonymized Facebook friendship network, and LiveJournal [1] is a social network blogging site. The SmallWorld, ErdosRenyi, and PA networks are random networks generated using the Watts-Strogatz small world network [111], Erdős-Rényi network [24], and Preferential Attachment network [11] models, respectively. For all the networks, we perform *degree-assortative* edge switch operations, where the degree of each vertex is considered as its label. In addition, we perform *age-assortative* edge switch operations on the social contact networks of Miami, NY, and LA, where each person's age is considered as its label. The age information for the other networks are either inappropriate or unavailable.

Table 4.1: *Data sets used in the experiments.* ***M*** *and* ***B*** *denote millions and billions, respectively.*

Network	Type of network	Vertices	Edges	Avg. degree
New York (NY)	Social Contact	17.88**M**	480.1**M**	53.70
Los Angeles (LA)	Social Contact	16.23**M**	459.3**M**	56.59
Miami	Social Contact	2.09**M**	51.50**M**	49.28
Sweden	Social Contact	9.46**M**	406.2**M**	85.88
Orkut	Social	3.07**M**	117.2**M**	76.28
Facebook	Social	3.10**M**	23.67**M**	15.28
LiveJournal	Social	4.80**M**	42.85**M**	17.68
ErdosRenyi	Erdős-Rényi	1.00**M**	500.00**M**	1000.00
SmallWorld	Small World	4.80**M**	48.00**M**	20.00
PA	Pref. Attachment	100.0**M**	1.00**B**	20.00

4.3 A Sequential Algorithm

We are given a network $G = (V, E, L)$ and the number of assortative edge switch operations τ to be performed. A naïve approach to perform an assortative edge switch operation is selecting two edges (u_1, v_1) and (u_2, v_2) uniformly at random from E and swapping the end vertices of the edges if the constraints are satisfied. If any constraint is not satisfied, the selected pair of edges is discarded, and a new pair is selected. For a large and relatively sparse network, the number of such discarded attempts (or pairs of edges) due to dissatisfying the first constraint is almost negligible; however, for the second constraint, the number of discarded attempts can be very large, as shown in Table 4.2. For example, to perform 10K degree-assortative edge switch operations on the LA network, the number of

Table 4.2: *Number of discarded attempts (due to dissatisfying the two constraints) to perform* 10*K age- and degree-assortative edge switch operations on different networks.*

Constraints	Miami-Age	LA-Deg	Facebook	LiveJournal	ErdosRenyi
Constraint 1	2	1	148	34	23
Constraint 2	30.3**M**	179.4**M**	617.2**M**	165.2**M**	125.4**M**

discarded attempts due to the first and second constraints are 1 and 179.4M, respectively. Assume that there are $\ell = 100$ different labels uniformly distributed among the vertices of a given network, i.e., for any label i, the number of vertices with label i is $\frac{n}{100}$. Then the probability of randomly selecting two edges (u_1, v_1) and (u_2, v_2) satisfying the second constraint is $(\frac{1}{100})^2$, which is very small. As a result, the number of discarded attempts can be very large, and it can make the algorithm extremely slow. To deal with this difficulty, we present an efficient sequential algorithm using a novel algorithmic approach.

4.3.1 An Efficient Sequential Algorithm

Let us denote a *bin* Z_{ij} to be the set of all edges having the same end vertex labels (i, j), where $0 \leq i, j \leq \ell - 1$, i.e., for an edge $(u, v) \in E$, if $L(u) = i$ and $L(v) = j$, then $(u, v) \in Z_{ij}$. For an undirected network, $Z_{ij} = Z_{ji}$, and we use Z_{ij} such that $j \leq i$. Note that the bins are disjoint and $\bigcup_{j \leq i} Z_{ij} = E$. For convenience, we relabel the bins from two indices (i, j) to a single bin number k. Let b_k be the new label of the bin Z_{ij}, where $k = \frac{i(i+1)}{2} + j$. Assume that there are Y such bins and let us denote them as $b_0, b_1, \ldots, b_{Y-1}$; there can be at most $\binom{\ell}{2} + \ell = O(\ell^2)$ such bins. The *size* of a bin b_i is the number of edges in b_i and is denoted by m_i, i.e., $m_i = |b_i|$. Then the number of possible pairs of edges in b_i is $a_i = {m_i}^2$. Note that the set of edges in a bin b_i changes dynamically during the course of an assortative edge switch process, although m_i remains invariant throughout the process.

Algorithm 8: Sequential Assortative Edge Switch

```
1 Procedure Assort-Edge-Switch(G, τ)
2     Partition E into minimum number of disjoint bins b_0, b_1, ..., b_{Y-1}, where for any i and for any
      pair of edges (u_1, v_1), (u_2, v_2) ∈ b_i, L(u_1) = L(u_2) and L(v_1) = L(v_2).
3     for k = 1 to τ do
4         b_i ← a random bin in [b_0, b_{Y-1}] with a probability of q_i = a_i / Σ_{j=0}^{Y-1} a_j
5         (u_1, v_1), (u_2, v_2) ← two uniform random edges in b_i
6         if u_1 = v_2, u_2 = v_1, u_1 ∈ N(v_2), or u_2 ∈ N(v_1) then      // self-loop or parallel edge
7             go to Line 4
8         Replace (u_1, v_1) and (u_2, v_2) by (u_1, v_2) and (u_2, v_1)
```

First, the algorithm constructs the bin b_i from G. Then an assortative edge switch operation is performed as follows: (*i*) a bin b_i is chosen randomly, where the probability of selecting b_i is $q_i = \frac{a_i}{\sum_{j=0}^{Y-1} a_j}$, (*ii*) a pair of edges is selected uniformly at random from b_i, and (*iii*) the end vertices of the edges are swapped with each other. This operation is repeated until τ

pairs of edges are switched. Note that this algorithm guarantees that both of the edges for an assortative edge switch operation are always selected from the same bin irrespective of how many bins there are, thus overcoming the drawback of the naïve approach. The pseudocode of the algorithm is given in Algorithm 8.

Time Complexity. Partitioning the set of edges E into the Y bins (Line 2 in Algorithm 8) takes $O(m + Y)$ time as initializing the bins takes $O(Y)$ time, and the bins are constructed from E in $O(m)$ time. The adjacency list of a vertex u can be maintained using a balanced binary tree, and searching (for parallel edges) and updating such a tree takes $O(\log d_u)$ time. Hence, an assortative edge switch operation (Lines 4–8) can be performed in $O(\log d_{max})$ time, where $d_{max} = \max_u d_u$ and τ such operations (Lines 3–8) take $O(\tau \log d_{max})$ time. Therefore, the time complexity of the algorithm is $O(m + Y + \tau \log d_{max})$.

Space Complexity. Storing all the m edges into the Y bins takes $O(m + Y)$ space.

4.3.2 Performance Evaluation of the Sequential Algorithm

Table 4.3 demonstrates the runtime comparison of our algorithm with the naïve approach. We use current calendar time as a random seed and a visit rate of 0.01 for the experiments since the naïve approach takes a large amount of time with a visit rate of 1.0. Our algorithm shows very good overall performance, e.g., for degree-assortative edge switch on the LA network, our algorithm is 494 times faster than the naïve approach.

Table 4.3: *Runtime comparison of our sequential algorithm and the naïve approach on various networks. Experiments performed with a visit rate of* 0.01.

Network	**Runtime (min.)**		**Faster by a**
	Naïve approach	**Our algo.**	**factor of**
Miami-Age	8.8	0.14	62.86
LA-Deg	929	1.88	494.1
Facebook	69.1	0.08	863.75
LiveJournal	48.87	0.15	325.8
ErdosRenyi	342.5	2.22	154.3

Although all of our experiments perform degree- and age-assortative edge switch operations, in the end, our algorithm can be used to perform τ assortative edge switch operations

on any labeled network, irrespective of what the label denotes and how the value of τ is obtained.

4.4 The Parallel Algorithms

A parallel algorithm for regular edge switch is presented in Chapter 3; however, this algorithm can be extremely slow for assortative edge switch for the same reasons as explained in Section 4.3. In this section, we present a novel parallel algorithm for assortative edge switch, which is based on our sequential algorithm presented in Algorithm 8.

Recall that the sequential algorithm selects both of the edges for an assortative edge switch operation from the same bin. Hence, assortative edge switch operations in a bin are independent of those in other bins. The parallel algorithm exploits this property to perform simultaneous assortative edge switch operations in parallel in different bins. The bins are distributed among the processors such that the computation load distribution is well-balanced. If a bin is very large compared to the other bins, the algorithm might need to partition and distribute the bin among multiple processors, which is discussed later in Section 4.4.3. For now, assume that each bin is entirely assigned to a single processor.

4.4.1 The Parallel Algorithm with Each Bin Assigned to a Single Processor

The parallel algorithm should distribute the bins to processors such that the computation cost is equally divided among the processors. Therefore, we need to estimate the computation cost associated with each bin, which is the number of assortative edge switch operations performed in a bin b_i. It raises the question of how many assortative edge switch operations among the total τ operations are performed in b_i? Let us denote X_i as the number of assortative edge switch operations performed in b_i by the sequential algorithm. Recall that, to perform an assortative edge switch operation, the sequential algorithm randomly selects a bin b_i and then chooses two edges randomly from b_i. Hence, a sequential algorithm does not need to know X_i in advance. However, in the parallel algorithm, all of the processors perform simultaneous assortative edge switch operations in parallel and different processors may need to work on different bins at the same time. As a result, for each i, X_i needs to be determined in advance by the parallel algorithm. It is easy to see that the random variables X_i are the multinomial random variables generated

by a multinomial distribution with parameters $(\tau, q_0, q_1, \ldots, q_{Y-1})$, that is,

$$\langle X_0, X_1, \ldots, X_{Y-1} \rangle \sim M(\tau, q_0, q_1, \ldots, q_{Y-1}). \tag{4.1}$$

where τ is the number of assortative edge switch operations and $q_i = \frac{a_i}{\sum_{j=0}^{Y-1} a_j}$ is the probability of selecting a bin b_i. The time complexity of the best known sequential algorithm, referred to as the *conditional distributed method* [37], for generating multinomial variables is $\Theta(\tau)$. To have an efficient parallel algorithm for assortative edge switch, we need a parallel algorithm for generating multinomial random variables. We use the algorithm presented in Section 3.5.2, which is the only known parallel algorithm for the problem and has a runtime of $O\left(\frac{\tau}{P} + Y \log P\right)$.

An overview of the parallel algorithm is as follows:

- ***Step 1:*** Generate multinomial random variables X_i in parallel with parameter $(\tau, q_0, q_1, \ldots, q_{Y-1})$ to estimate the computational cost associated with each bin.
- ***Step 2:*** Using the estimated costs X_i, partition the bins $b_0, b_1, \ldots, b_{Y-1}$ among the P processors such that the computation load distribution is well-balanced.
- ***Step 3:*** Each processor P_i simultaneously performs assortative edge switch operations in parallel in the bins assigned to it.

Now we describe the last two steps of the algorithm.

Partitioning (Step 2). Let B_i be the set of bins, $Y_i = |B_i|$ be the number of bins and $M_i = \sum_{b_j \in B_i} m_j$ be the number of edges assigned to a processor P_i. Then the *workload* (or *load*) W_i in P_i is the summation of the computation costs associated with the bins in B_i, i.e., $W_i = \sum_{b_j \in B_i} X_j$. Let W be the *maximum load* among all of the processors, i.e., $W = \max_i W_i$. Now the goal is to distribute the Y bins among the P processors such that the maximum load W is minimized. Finding an assignment of the bins for the optimum solution is, unfortunately, an NP-hard problem [68]. The best-known approximation algorithm for this problem is presented in [68], which has an approximation ratio of 1.5. This greedy algorithm sorts the bins in non-increasing order of the loads. To do so, we use a parallel version of quick sort [50]. Then the algorithm makes one pass over the sorted bins to assign each bin to a processor having the minimum load at the time of the assignment, as shown in Algorithm 9.

Switching Edges (Step 3). Each processor P_i constructs the bins in B_i and then simultaneously performs W_i assortative edge switch operations in parallel, as shown in Algorithm 10. The program terminates when all of the processors complete their assortative edge switch operations.

Algorithm 9: Partitioning Algorithm

1 **Procedure** PARTITIONING$\left(\{b_i\}_{i\in[0,Y)}, \{X_i\}_{i\in[0,Y)}\right)$
2 **for** $i = 0$ *to* $P-1$ **do**
3 $W_i \leftarrow 0$ `// computation cost at processor` P_i
4 $B_i \leftarrow \emptyset$ `// set of bins assigned to processor` P_i
5 **Sort** the bins in non-increasing order of X_j `//` $0 \le j \le Y-1$
6 Assume that $X_0 \ge X_1 \ge \ldots \ge X_{Y-1}$
7 **for** $j = 0$ *to* $Y-1$ **do**
8 Let P_i be the processor with rank $i = \arg\min_k W_k$ `//` $0 \le k \le P-1$
9 $B_i \leftarrow B_i \bigcup \{b_j\}$
10 $W_i \leftarrow W_i + X_j$

Algorithm 10: Parallel Assortative Edge Switch

`/* Each processor` P_i `performs the following in parallel: */`
1 **Procedure** PARALLEL-ASSORT-EDGE-SWITCH$\left(B_i, \{X_j\}_{b_j\in B_i}\right)$
2 **foreach** *bin* $b_j \in B_i$ **do**
3 **for** $k = 1$ *to* X_j **do**
4 $(u_1, v_1), (u_2, v_2) \leftarrow$ two uniform random edges in b_j
5 **if** $u_1 = v_2$, $u_2 = v_1$, $u_1 \in N(v_2)$, *or* $u_2 \in N(v_1)$ **then** `// self-loop or parallel edge`
6 go to Line 4
7 **Replace** (u_1, v_1) and (u_2, v_2) by (u_1, v_2) and (u_2, v_1)

Time Complexity. In step 1, multinomial random variables are generated in parallel in $O\left(\frac{\tau}{P} + Y \log P\right)$ time. In step 2, the parallel version of quick sort and the assignment of the bins take $O\left(\frac{Y}{P}\log Y + \log^2 P\right)$ and $O\left(P + Y \log P\right)$ time, respectively. In step 3, each processor P_i constructs the Y_i bins in $O\left(Y_i + M_i\right)$ time and then performs the assortative edge switch operations in $O\left(W_i \log d_{max}\right)$ time. Hence, the time complexity in each processor P_i is $O\left(\frac{\tau}{P} + Y\log P + \frac{Y}{P}\log Y + \log^2 P + P + Y_i + M_i + W_i \log d_{max}\right)$.

Space Complexity. Each processor P_i stores all the M_i edges in Y_i bins using $O\left(Y_i + M_i\right)$ space.

4.4.2 Performance Analysis of the Parallel Algorithm with Each Bin Assigned to a Single Processor

In this section, we analyze the performance of the parallel algorithm. Table 4.4 provides a summary of the number of bins and assortative edge switch operations performed on the ten networks with a visit rate of 1.0.

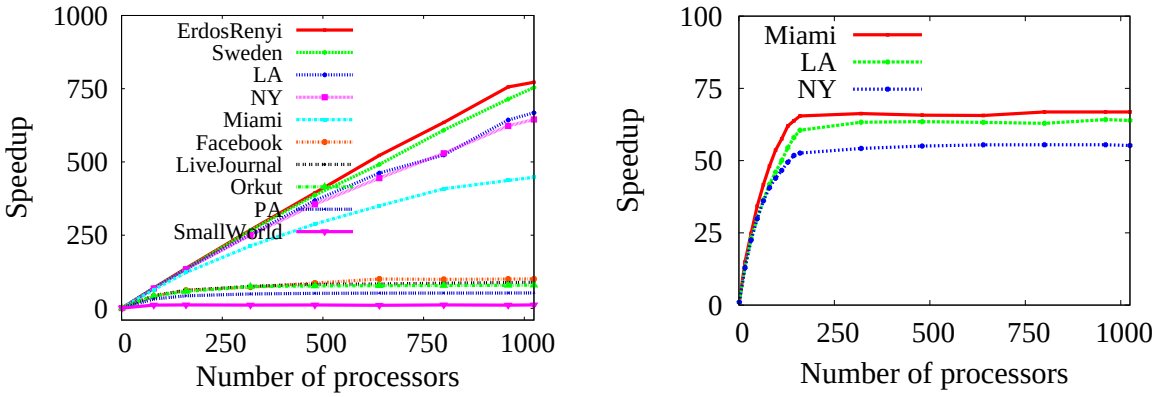
Table 4.4: *Number of bins and assortative edge switch operations performed on the ten networks with a visit rate of* 1.0.

Network	**Bins**		**Assort. edge switch**
	Degree assort.	**Age assort.**	**(Visit rate = 1.0)**
New York (NY)	85.7**K**	4186	4.80**B**
Los Angeles (LA)	83.1**K**	4186	4.58**B**
Miami	66.1**K**	4186	457.2**M**
Sweden	83.2**K**	-	4.03**B**
Orkut	2.46**M**	-	1.09**B**
Facebook	378**K**	-	200.9**M**
LiveJournal	802**K**	-	376.5**M**
ErdosRenyi	36.62**K**	-	5.0**B**
SmallWorld	161	-	424.5**M**
PA	1.58**M**	-	10.36**B**

Experimental Setup. We use a HPC cluster consisting of 64 compute nodes. Each node has a dual-socket Intel Sandy Bridge E5-2670 2.60GHz 8-core processor (16 cores per node) with 64GB memory. The algorithms are developed with MPICH2 (v1.9), optimized for Qlogic QDR Infiniband cards.

Strong Scaling

Figures 4.2(a) and 4.2(b) illustrate the strong scaling performance for degree- and age-assortative edge switch, respectively, on various networks. The algorithm achieves a speedup between 12 and 772 with 1024 processors. For some networks, the speedup is



(a) *Degree-assortative* ***(b)*** *Age-assortative*

Figure 4.2: *Strong scaling of the parallel algorithm.*

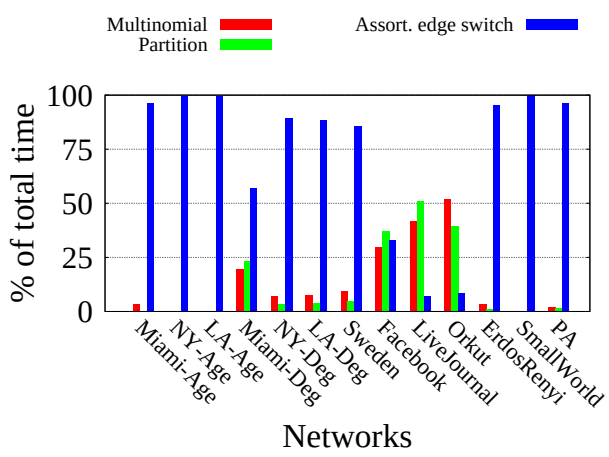


Figure 4.3: *Execution time of the individual steps of the parallel algorithm for age- and degree-assortative edge switch on various networks.*

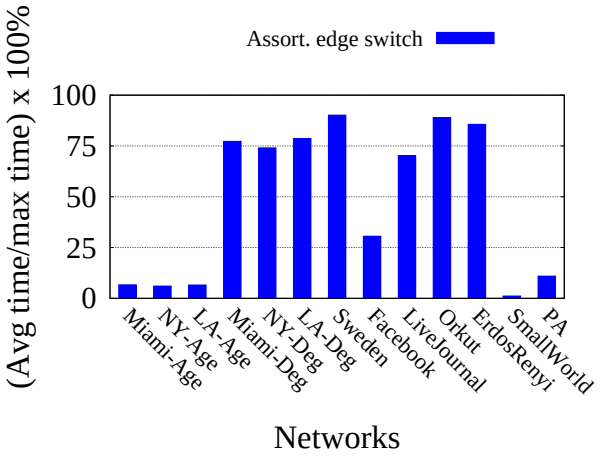
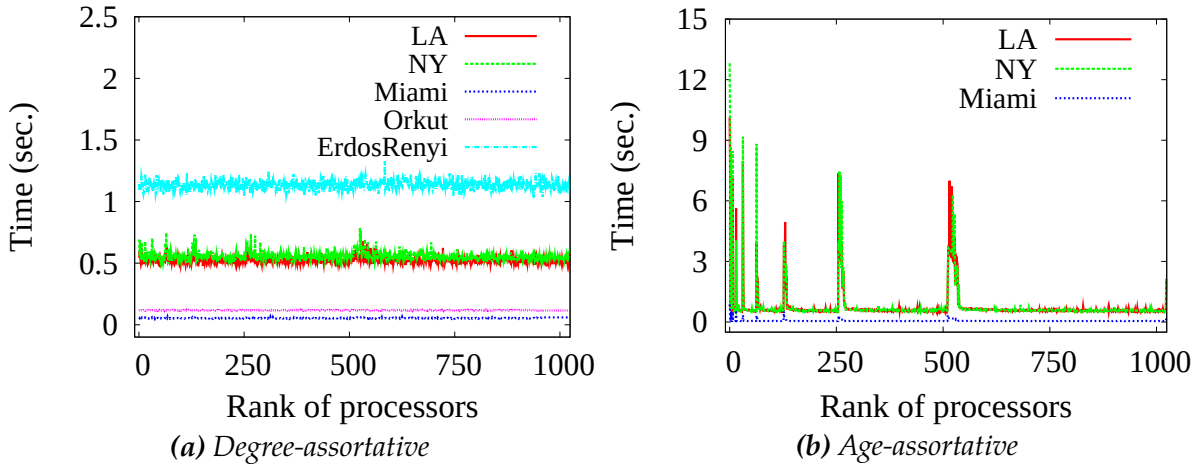


Figure 4.4: *Ratio of the average execution time of all processors to the maximum execution time taken by a processor in the third step of the algorithm.*

poor compared to the other networks. Next, we investigate the load distribution among the processors to understand the reason for this poor performance.

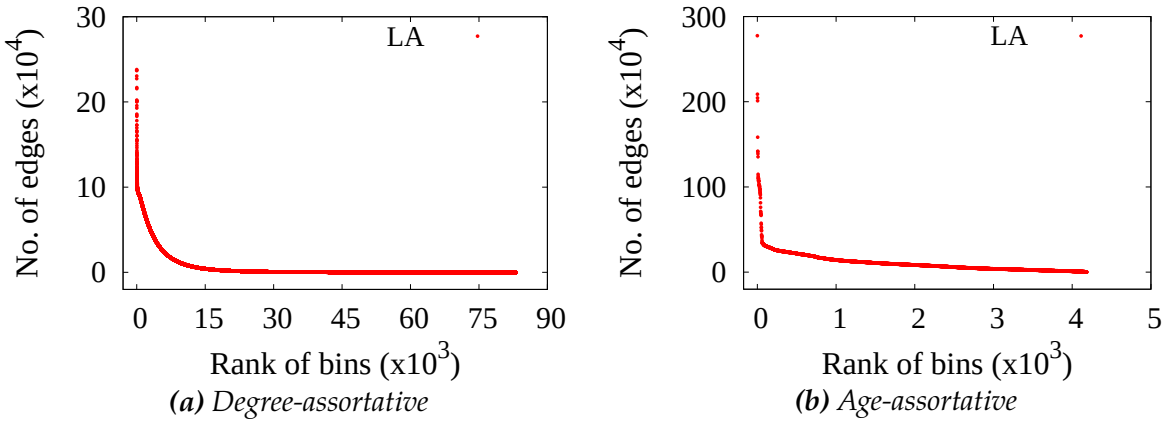
Load Distribution

First, we measure the time taken by each of the three steps (step 1: multinomial, step 2: partitioning, step 3: assortative edge switch) of the algorithm with 1024 processors, as shown in Figure 4.3. The assortative edge switch step takes the largest amount of time



(a) *Degree-assortative* ***(b)*** *Age-assortative*

Figure 4.5: *Execution time of the individual processors in the third step of the algorithm.*



(a) *Degree-assortative* ***(b)*** *Age-assortative*

Figure 4.6: *Distribution of edges among the bins for the LA network.*

among the three steps for all the networks, except for the Facebook, LiveJournal, and Orkut networks, in which the number of bins Y is significantly higher than that of the other networks (see Table 4.4). Therefore, generating the multinomial random variables and partitioning the bins take more time for these three networks. Unlike the first two steps, where every processor takes almost an equal amount of time, the execution time of the individual processors in the third step can vary significantly because of a poor load distribution. Figure 4.4 demonstrates the ratio of T_{avg} and T_{max}, where T_{avg} is the average execution time of the processors in the assortative edge switch step and T_{max} is the maximum time among them. The maximum value of the ratio is 100% for a perfectly-balanced load distribution in the ideal case. A higher value implies a well-balanced load distribution, whereas a lower value indicates a poor load distribution among the processors. We observe a well-balanced load distribution for degree-assortative edge switch on the

Miami, NY, LA, Sweden, LiveJournal, Orkut, and ErdosRenyi networks. In contrast, we observe a poor load distribution for age-assortative edge switch on the Miami, NY, and LA networks and degree-assortative edge switch on the PA and SmallWorld networks. For the PA network, the poor load distribution in the third step causes the step to take the largest amount of time despite having a large number of bins, which is in contrast to the scenario for the Facebook, LiveJournal, and Orkut networks. For the Facebook network, a few processors contain many small-size bins (with a few edges), and the number of discarded attempts in these bins are very high, yielding a larger execution time in these processors (hence, T_{max} is large). As a result, we observe a low ratio despite each processor performing roughly an equal number of assortative edge switch operations and the low ratio is not a consequence of load balancing. The observations are further supported by Figures 4.5(a) and 4.5(b), which show the individual execution time of the processors in the third step of the algorithm. For a better understanding, we analyze the load distribution in detail for the LA network.

Figures 4.6(a) and 4.6(b) illustrate the distribution of the sizes of the bins for degree- and age-assortative edge switch, respectively, on the LA network. The distribution for age-assortative edge switch is highly skewed, having a few very large bins and many small bins, which in turn makes a poor load distribution among the processors, as shown in Figure 4.5(b). A few processors containing the larger bins are taking significantly more time than the processors containing the smaller bins, which results in a low speedup. On the other hand, for degree-assortative edge switch, there is a good number of both larger and smaller bins among the total 83.1K bins and the differences between the larger and smaller bins are substantially smaller than that of the age-assortative counterpart. The algorithm assigns a few larger bins along with many smaller bins to each processor and consequently exhibits a well-balanced load distribution, as shown in Figure 4.5(a), thus resulting in a good speedup.

Note that if the number of bins is less than the number of processors, i.e., $Y < P$, then some processors remain idle in the third step of the computation; we observe this phenomenon for the SmallWorld network. To deal with the poor load distribution, we present a parallel algorithm with an improved load balancing scheme in the next section.

4.4.3 The Parallel Algorithm with Improved Load Balancing

As we discussed earlier, some bins can have higher computational costs, i.e., the number of assortative edge switch operations X_i performed in a bin b_i can be significantly larger than $\frac{\tau}{P}$, which can cause a poor load distribution. For a better load balancing, such a bin may need to be partitioned and distributed among multiple processors. Let Δ be some *threshold*

such that Δ is larger than $\frac{\tau}{P}$. We call a bin *large* if $X_i \geq \Delta$, and *small*, otherwise. Let Q be the total number of processors assigned to the large bins. First, we explain how to perform assortative edge switch operations in a single large bin with multiple processors.

Assume that a large bin b_i is partitioned into disjoint *blocks* (or subsets of edges), and the blocks are distributed among the processors $P_x, P_{x+1}, \ldots, P_y$, where j, k, l are processor ranks such that $x \leq j, k, l \leq y$. The bin b_i is partitioned into blocks such that a subset of vertices, having consecutive vertex-ids, along with their incident edges in b_i are assigned to a block and each such block contains almost an equal number of edges. A vertex u's *partial* adjacency list in b_i, i.e., $N_i(u) = \{v \in V | (u, v) \in b_i\}$, entirely belongs to a unique block. Then each processor P_j performs simultaneous assortative edge switch operations in parallel. Assortative edge switch in a bin is similar to regular edge switch because the end vertices of the edges in a bin have the same labels. A parallel algorithm for regular edge switch has been presented in Chapter 3. We use this algorithm to switch the edges in a large bin. A summary of an assortative edge switch operation performed by a processor P_j is as follows. P_j selects an edge (u_1, v_1) randomly from its own block. The another edge (u_2, v_2) is chosen in two steps: (*i*) P_j selects a processor P_k with a probability proportional to the number of edges belonging to P_k, and (*ii*) P_j requests P_k to select (u_2, v_2) randomly from its block. Then P_j and P_k work together to check whether switching the edges (u_1, v_1) and (u_2, v_2) creates any loop or parallel edge. If it creates any loop or parallel edge, the selected pair of edges is discarded, and a new pair is selected by P_j. Otherwise, P_j and P_k work together to update (u_1, v_1) and (u_2, v_2) by (u_1, v_2) and (u_2, v_1), respectively. In fact, P_j and P_k may require updating an edge in another processor P_l ($P_j \neq P_l \neq P_k$). The details can be found in Chapter 3.

Now, we discuss how to determine Δ and Q. Apparently, it seems that any bin b_i with $X_i > \frac{\tau}{P}$ needs to be partitioned and distributed among multiple processors. However, partitioning a bin incurs communication and synchronization overhead due to the need for exchanging messages among multiple processors even for a single edge switch operation. Thus, we partition a bin among multiple processors only when X_i is significantly larger than $\frac{\tau}{P}$, i.e., when the gain achieved by partitioning a bin is larger than the communication and synchronization cost incurred by partitioning the bin. We assume $\Delta = \alpha \times \frac{\tau}{P}$ for some constant $\alpha > 1$. Similarly, to perform X_i assortative edge switch operations in a large bin b_i, we should ideally assign $\lceil \frac{X_i}{\tau/P} \rceil$ processors for b_i. However, due to communication overhead, we need to assign a larger number of processors. Hence, we assign $Q_i = \lceil \beta \times X_i \times \frac{P}{\tau} \rceil$ processors for a large bin b_i for some constant $\beta > 1$. Then $Q = \sum_i Q_i$ and the number of processors assigned to the small bins is $S = P - Q$. The performance of the algorithm greatly depends on Δ and Q, thusly on α and β. We experimented with many different types of networks and find that for $\alpha \in [2.4, 2.75]$ and $\beta \in [12, 15]$, the algorithm exhibits

good performance, which is very close to the optimal performance, as shown in the next section.

4.4.4 Performance Analysis of the Parallel Algorithm with Improved Load Balancing

In this section, we analyze the performance of the algorithm with the same experimental setup as before.

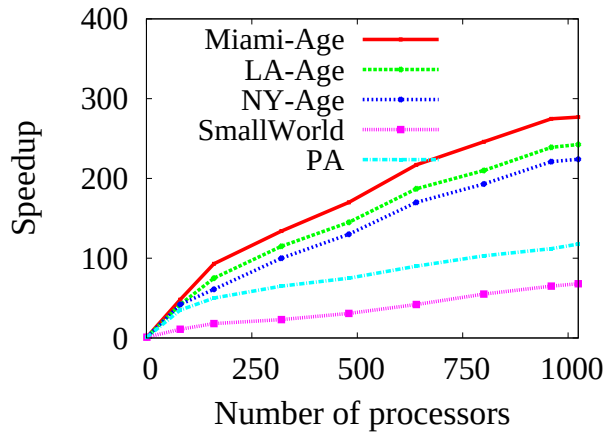


Figure 4.7: *Strong scaling of the parallel algorithm with improved load balancing.*

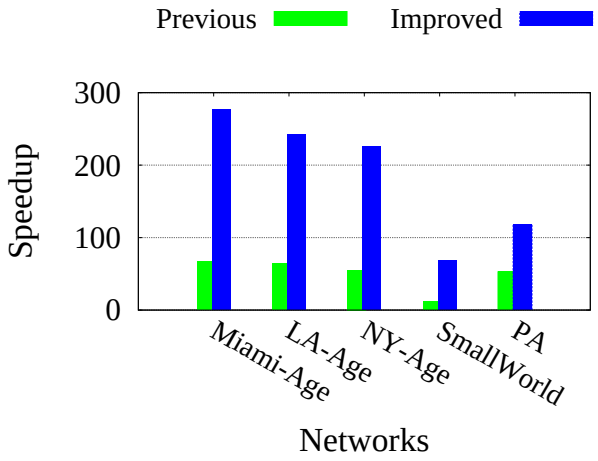


Figure 4.8: *A comparison of speedup improvement with 1024 processors.*

Figure 4.7 shows the strong scaling performance of the parallel algorithm and Figure 4.8 demonstrates a comparison of the speedup improvement. The algorithm achieves a maximum speedup of 277 for age-assortative edge switch on the Miami network with 1024 processors, which is a four-fold improvement obtained by a well-balanced load distribution among the processors, as shown in Figure 4.9. Next, we analyze the effect of α and β (thus Δ and Q) on the speedup.

Figure 4.10 shows how the speedup varies with different values of α and β for age-assortative edge switch on the Miami network with 1024 processors. For a fixed value of β (say $\beta = 14$) and with the increase of α, more large-size bins are partitioned among the same number of processors, yielding a speedup increase up to some value of α, referred to as *optimal* α (optimal $\alpha = 2.5$ for $\beta = 14$), beyond which the speedup starts decreasing because of the increase of communication and synchronization overhead incurred by the increasing number of bins getting partitioned. We observe a similar pattern for a fixed value of α

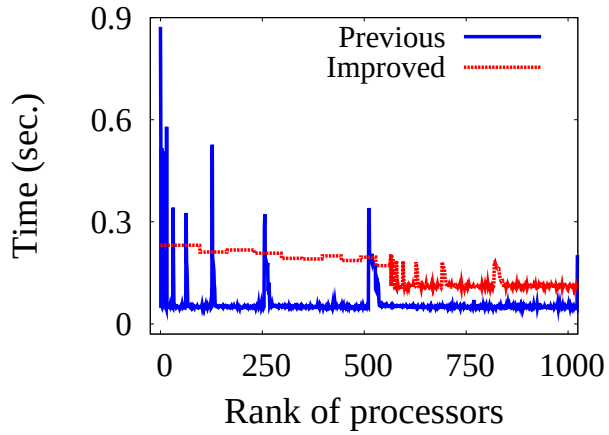


Figure 4.9: *A comparison of runtime of the individual processors for Miami-Age.*

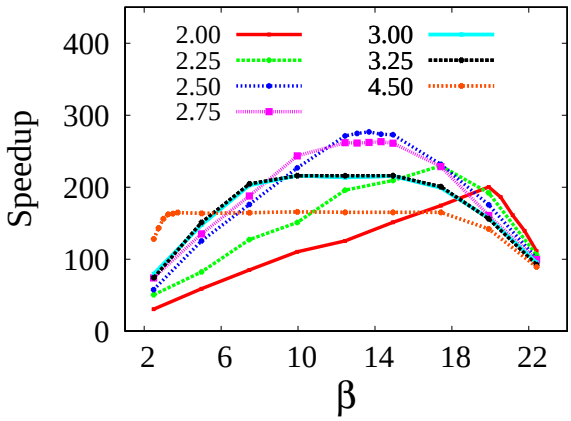


Figure 4.10: *Speedup with increasing β for Miami-Age using different values of α.*

and with the increase of β as well. For lower values of β, fewer processors are working on the large bins. As a result, the execution times of the Q processors are higher than that of the S processors working on the small bins. This is further illustrated in Figure 4.11, which shows the execution time of the individual processors with varying β and a fixed $\alpha = 2.5$. We observe a pattern of many flat segments, where each flat horizontal segment is the time taken by the processors working on the same large bin. With the increase of β, thusly Q, the amount of work for each of the Q processors decreases, whereas the time taken by the S processors increase as fewer processors are dealing with the small bins. The highest speedup is achieved when the maximum execution time among the Q processors is somewhat balanced with that of the S processors. We observe similar phenomena for the other networks as well and recommend using $\alpha \in [2.4, 2.75]$ and $\beta \in [12, 15]$, for which the algorithm achieves a good speedup.

In the parallel algorithm, multiple processors perform simultaneous assortative edge switch operations in parallel, whereas in the sequential algorithm, one pair of edges is selected and the edges are switched before selecting the next pair of edges. It raises the question of whether the parallel algorithm has the same effect as that of the sequential algorithm? In principle, the parallel algorithm is designed such that it stochastically produces the same effect (with the help of the multinomial distribution) on a network as the sequential algorithm would do. We also experimentally verify this by showing that the average clustering coefficient and the average shortest path distance of a network change similarly with assortative edge switches by the sequential and parallel algorithms (see Figures 4.12 and 4.13). We use the NY, Orkut, and Facebook networks and vary the visit rate from 0.025 to 1. For both properties, the changes by the sequential and

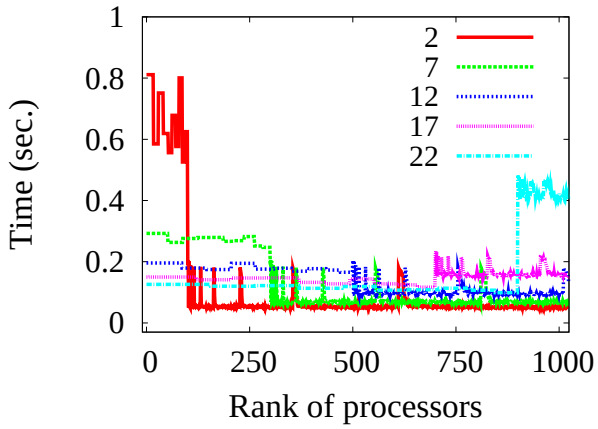


Figure 4.11: *Runtime of the individual processors with different values of β for Miami-Age using a fixed $\alpha = 2.5$.*

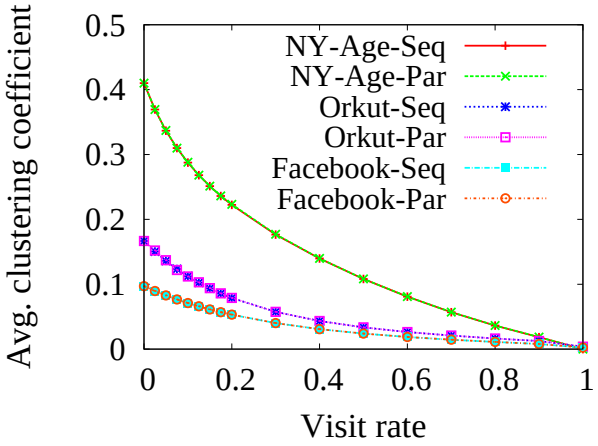


Figure 4.12: *Average clustering coefficient changes similarly with assortative edge switches by the sequential and parallel algorithms.*

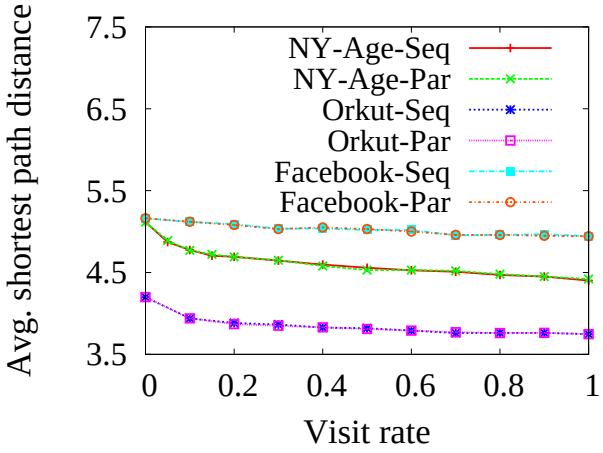


Figure 4.13: *Average shortest path distance changes similarly with assortative edge switches by the sequential and parallel algorithms.*

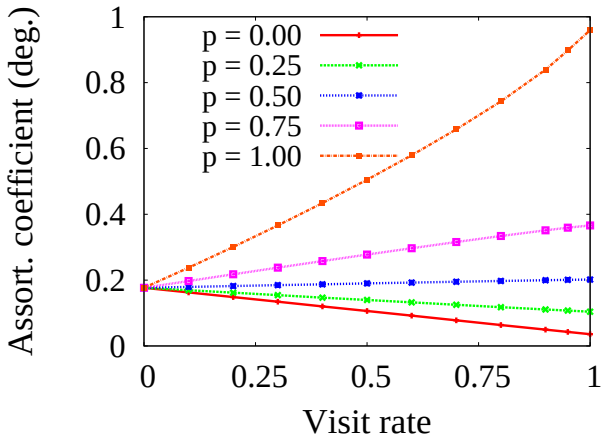


Figure 4.14: *Varying degree-assort. coefficient by age-assort. edge switch on the Miami network through a parameter $p \in [0, 1]$.*

parallel algorithms are very similar; in fact, they overlap with each other, and it is difficult to distinguish them in the figures. These experiments also show how some network properties change with assortative edge switches while keeping the degree sequence and assortativity invariant. We also demonstrate how assortative edge switch can be used to generate random networks by keeping one assortative coefficient A_1 invariant and varying another assortative coefficient A_2 to a desired level through a parameter p

($0 \leq p \leq 1$). Xulvi-Brunet et al. [115] proposed one such algorithm, where with probability p, an edge switch operation connects the two higher degree vertices with an edge and the two lower degree vertices with another edge. With probability $(1 - p)$, the edges are switched randomly. Figure 4.14 shows how the degree assortative coefficient changes for different value of p with the age-assortative edge switch process on the Miami network.

4.5 Conclusion

We have developed efficient sequential and parallel algorithms for assortative edge switch in large networks. They can be used to study the sensitivity of network topology on the dynamics over a network as well as to generate network perturbations of a given network by maintaining the same degree sequence and assortative coefficient. Our sequential algorithm employs a novel algorithmic approach, which further led to the development of an efficient parallel algorithm for the problem. We rigorously analyzed the computation load distribution of the parallel algorithm, which resulted in the design of a good partitioning scheme exhibiting a well-balanced load distribution among the processors, thus leading to an efficient algorithm. Our parallel algorithm provides good speedup and scales to a large number of processors. We believe that the presented algorithms will prove useful for the efficient analysis and mining of emerging massive networks.

Acknowledgment

We thank our external collaborators, members of the Network Dynamics and Simulation Science Laboratory (NDSSL), and 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, NSF NetSE Grant CNS-1011769, NSF SDCI Grant OCI-1032677, NIH MIDAS Grant 5U01GM070694-11, NSF DIBBs Grant ACI-1443054 and NSF Big Data Grant IIS-1633028.

Chapter 5

Parallel Algorithms for Generating Random Networks with Prescribed Degree Sequences

5.1 Introduction

Random graphs are widely used for modeling many complex real-world systems such as the Internet [103], biological [47], social [116], and infrastructure [70] networks to understand how the systems work through obtaining rigorous mathematical and simulation results. Many random graph models such as the Erdős-Rényi [24], the Preferential Attachment [11], the small-world [111], and the Chung-Lu [32] models have been proposed to capture various characteristics of real-world systems. Degree sequence is one of the most important aspects of these systems and has been extensively studied in graph theory [15, 58, 75]. It has significant applications in a wide range of areas including structural reliability and communication networks because of the strong ties between the degrees of vertices and the structural properties of and dynamics over a network [21].

Random graphs with given degree sequences are widely used in uniform sampling of random graphs as well as in counting the number of graphs having the same degree sequence [14, 20, 33, 39]. For example, in an epidemiology study of sexually transmitted diseases [73], anonymous surveys collect data about the number of sexual partners of an individual within a given period of time, and then the problem reduces to generating a network obeying the degree sequence collected from the survey, and studying the disease dynamics over the network. Other examples include determining the total number of structural isomers of chemical compounds such as alkanes, where the valence of an atom is

the degree. Moreover, the random graphs with given degree sequences can capture many characteristics such as dependent edges and non-binomial degree distribution that are absent in many classical models such as the Erdős-Rényi [24] graph model. They also have important applications in string theory, random matrix theory, and matching theory [75].

The problem of generating a random graph with a given degree sequence becomes considerably easier if self-loops and parallel edges are allowed. Throughout this chapter, we consider simple graphs with no self-loops or parallel edges. Most prior work on generating random graphs involves sequential algorithms, and they can be broadly categorized in two classes: (***i***) edge swapping and (***ii***) stub-matching. Edge swapping [18, 34, 45, 48] uses the Markov chain Monte Carlo (MCMC) scheme on a given graph having the degree sequence, as discussed in Chapters 3 and 4.

On the other hand, among the swap-free stub-matching methods, the *configuration* or *pairing* method [86] is very popular and uses a direct graph construction method. For each vertex, it creates as many stubs or "dangling half-edges" as of its degree. Then edges are created by choosing pairs of vertices randomly and connecting them. This approach creates parallel edges, which are dealt with by restarting the process. Unfortunately, the probability of restarting the process approaches 1 for larger degree sequences. Many variants [67, 108, 114] of the configuration models have been studied to avoid parallel edges for the regular graphs. By using the Havel-Hakimi method [57], a deterministic graph can be generated following a given degree sequence. Bayati et al. [14] presented an algorithm for counting and generating random simple graphs with given degree sequences. However, this algorithm does not guarantee to always generate a graph, and it is shown that the probability of not generating a graph is small for a certain bound on the maximum degree, which restricts many degree sequences. Genio et al. [39] presented an algorithm to generate a random graph from a given degree sequence, which can be used in sampling graphs from the graphical realizations of a degree sequence. Blitzstein et al. [20] also proposed a sequential importance sampling [74] algorithm to generate random graphs with a given degree sequence, which can generate every possible graph with the given degree sequence with a non-zero probability. Moreover, the distribution of the generated graphs can be estimated, which is a much-desired result used in sampling random graphs.

A deterministic parallel algorithm for generating a simple graph with a given degree sequence has been presented by Arikati et al. [10], which runs in $O(\log n)$ time using $O(n+m)$ CRCW PRAM [65] processors, where n and m denote the number of vertices and edges in the graph, respectively. From a given degree sequence, the algorithm first computes an appropriate bipartite sequence (degree sequence of a bipartite graph), generates a deterministic bipartite graph obeying the bipartite sequence, applies some edge swap techniques to generate a symmetric bipartite graph, and then reduces the symmetric

bipartite graph to a simple graph having the given degree sequence. Another parallel algorithm, with a time complexity of $O\left(\log^4 n\right)$ using $O\left(n^{10}\right)$ EREW PRAM processors, has been presented in [40], where the maximum degree is bounded by the square-root of the sum of the degrees, which restricts many degree sequences. A parallel algorithm for generating a random graph with a given *expected* degree sequence has been presented in [5]. However, there is no existing parallel algorithm for generating random graphs following an exact degree sequence, which can provably generate each possible graph, having the given degree sequence, with a positive probability. In this chapter, we present an efficient parallel algorithm for generating a random graph with an exact given degree sequence. We choose to parallelize the sequential algorithm by Blitzstein et al. [20] because of its rigorous mathematical and theoretical results, and the algorithm supports all of the important and much-desired properties below, whereas the other algorithms are either heuristic-based or lack some of the following properties:

- It can construct a random simple graph with a prescribed degree sequence.
- It can provably generate each possible graph, obeying the given degree sequence, with a positive probability.
- It can be used in importance sampling by explicitly measuring the weights associated with the generated graphs.
- It is guaranteed to terminate with a graph having the prescribed degree sequence.
- Given a degree sequence of a tree, a small tweak while assigning the edges allows the same algorithm to generate trees uniformly at random.
- It can be used in estimating the number of possible graphs with the given degree sequence.

In this chapter, we present an efficient shared-memory parallel algorithm for generating random graphs with given exact degree sequences. The dependencies among assigning edges to vertices in a particular order to ensure the algorithm always successfully terminates with a graph, the requirement of keeping the graph simple, maintaining an exact stochastic process as that of the sequential algorithm, and concurrent writing by multiple cores in the global address space lead to significant challenges in designing a parallel algorithm. Dealing with these requires complex synchronization among the processing cores. Our parallel algorithm achieves a maximum speedup of 20.4 with 32 cores. We also present a comparative study of some structural properties of the random graphs generated by our parallel algorithm with that of the real-world graphs and random graphs generated

by swapping edges. One of the steps in our parallel algorithm requires checking the graphicality of a given degree sequence, i.e., whether there exists a graph with the degree sequence, using the Erdős-Gallai characterization [43] in parallel. We present here a novel parallel algorithm for checking the Erdős-Gallai characterization, which achieves a speedup of 23 using 32 cores.

The rest of the chapter is organized as follows. Section 5.2 describes the preliminaries and notations used in this chapter. Our main parallel algorithm for generating random networks along with the performance results are presented in Section 5.3. In Section 5.4, we present a parallel algorithm for checking the Erdős-Gallai characterization of a given degree sequence accompanied by the performance evaluation of the algorithm. Finally, we conclude in Section 5.5.

5.2 Preliminaries

Below are the notations, definitions, and computation model used in this chapter.

Notations. We use $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ to denote a simple graph, where $\mathbb{V}$ is the set of vertices and $\mathbb{E}$ is the set of edges. We are given a *degree sequence* $\mathbb{D} = (d_1, d_2, \ldots, d_n)$. There are a total of $n = |\mathbb{V}|$ vertices labeled as $1, 2, \ldots, n$, and d_i is the degree of vertex i, where $0 \leq d_i \leq n-1$. For a degree sequence $\mathbb{D}$ and distinct $u, v \in \{1, 2, \ldots, n\}$, we define $\ominus_{u,v}^{\mathbb{D}}$ to be the degree sequence obtained from $\mathbb{D}$ by subtracting 1 from each of d_u and d_v. Let d'_j be the degree of vertex j in the degree sequence $\ominus_{u,v}^{\mathbb{D}}$, then

$$d'_j = \begin{cases} d_j - 1 & \text{if } j \in \{u, v\}, \\ d_j & \text{otherwise.} \end{cases} \tag{5.1}$$

If there is a simple graph $\mathbb{G}$ having the degree sequence $\mathbb{D}$, then there are $m = |\mathbb{E}|$ edges in $\mathbb{G}$, where $2m = \sum_i d_i$. The terms graph and network are used interchangeably throughout the chapter. We use `K`, `M`, and `B` to denote thousands, millions, and billions, respectively; e.g., `1M` stands for one million. For the parallel algorithms, let $\mathcal{P}$ be the number of processing cores, and $\mathbb{P}_k$ the core with rank k, where $0 \leq k < \mathcal{P}$. A summary of the frequently used notations (some of them are introduced later for convenience) is provided in Table 5.1.

Residual Degree. During the course of a graph generation process, the *residual degree* of a vertex u is the remaining number of edges incident on u, which have not been created yet. From hereon, we refer to the degree d_u of a vertex u as the residual degree of u at any given time, unless otherwise specified.

Graphical Sequence. A degree sequence $\mathbb{D}$ of non-negative integers is called *graphical*

Table 5.1: *Notations used frequently in the chapter.*

Symbol	Description	Symbol	Description
$\mathbb{D}$	Degree sequence	d_i	Degree of vertex i
$\mathbb{V}$	Set of vertices	n	Number of vertices
$\mathbb{E}$	Set of edges	m	Number of edges
$\mathcal{P}$	Number of cores	$\mathbb{P}_k$	Core with rank k
$\mathbb{C}$	Candidate set	C	Corrected Durfee number
$\mathbb{G}$	Graph	K	Thousands
M	Millions	B	Billions

if there exists a labeled simple graph with vertex set $\{1, 2, \ldots, n\}$, where vertex i has degree d_i. Such a graph is called a *realization* of the degree sequence $\mathbb{D}$. Note that there can be many graphs realizing the same degree sequence. Eight equivalent necessary and sufficient conditions for testing the graphicality of a degree sequence are listed in [76]. Among them, the Erdős-Gallai characterization [43] is the most famous and frequently used criterion. Another popular recursive test for checking a graphical sequence is the Havel-Hakimi method [57].

Erdős-Gallai Characterization [43]. Assuming a given degree sequence $\mathbb{D}$ is sorted in non-increasing order, i.e., $d_1 \geq d_2 \geq \ldots \geq d_n$, the sequence $\mathbb{D}$ is graphical if and only if $\sum_{i=1}^{n} d_i$ is even and

$$\text{for each } k \in \{1, 2, \ldots, n\}, \sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(k, d_i). \tag{5.2}$$

For example, $\mathbb{D}_1 = (3, 3, 2, 2, 2)$ is a graphical sequence and there is a realization of $\mathbb{D}_1$ as it satisfies the Erdős-Gallai characterization, whereas $\mathbb{D}_2 = (4, 3, 2, 1)$ is not a graphical sequence and there is no simple graph realizing $\mathbb{D}_2$, as shown in Figures 5.1 and 5.2.

Computation Model. We develop algorithms for shared-memory parallel systems. All the cores can read from and write to the global address space. In addition, each core can have its own local variables and data structures.

	Degree seq. D_1					Degree seq. D_2			
i	1	2	3	4	5	1	2	3	4
d_i	3	3	2	2	2	4	3	2	1
LHS	3	6	8	10	12	4	7	9	10
RHS	4	8	10	14	20	3	5	7	12

Figure 5.1: *Graphicality check for the degree sequences* $\mathbb{D}_1 = (3, 3, 2, 2, 2)$ *and* $\mathbb{D}_2 = (4, 3, 2, 1)$ *using the Erdős-Gallai characterization, where LHS and RHS denote the left hand side and right hand side values of Eq. (5.2), respectively.*

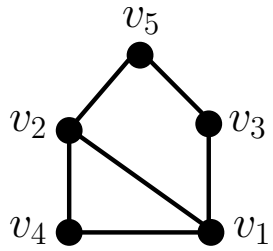


Figure 5.2: *A simple graph realizing the degree sequence* $\mathbb{D}_1 = (3, 3, 2, 2, 2)$.

5.3 Generating Random Graphs with Prescribed Degree Sequences

We briefly discuss the sequential algorithm in Section 5.3.1. Then we present our parallel algorithm in Section 5.3.2 and the experimental results in Section 5.3.3.

5.3.1 Sequential Algorithm

Blitzstein et al. [20] presented a sequential importance sampling [74] algorithm for generating random graphs with exact prescribed degree sequences. This approach first creates all edges incident on the vertex having the minimum degree in the sequence, then moves to the next vertex having the minimum degree to create its incident edges and so on. To create an edge incident on a vertex u, a candidate list $\mathbb{C}$ is computed using the Erdős-Gallai characterization such that, after adding an edge by connecting u to any candidate vertex v from the list $\mathbb{C}$, the residual degree sequence remains graphical and the graph remains simple. Then an edge (u, v) is assigned by choosing v from the candidate list $\mathbb{C}$ with a probability proportional to the degree of v. This process is repeated until all edges incident on vertex u are assigned.

For example, for a given degree sequence $\mathbb{D} = (3, 3, 2, 2, 2)$, the algorithm starts by assigning edges incident on vertex v_3. It computes the candidate list $\mathbb{C} = \{v_1, v_2, v_4, v_5\}$. Say it chooses the vertex v_5 from $\mathbb{C}$ and assigns the edge (v_3, v_5). Then the new degree sequence is $\mathbb{D} = (3, 3, 1, 2, 1)$, and the new candidate list for assigning the remaining edge incident on vertex v_3 is $\mathbb{C} = \{v_1, v_2\}$. Say the algorithm selects v_1 from $\mathbb{C}$ and assigns the edge (v_3, v_1). Now the new degree sequence is $\mathbb{D} = (2, 3, 0, 2, 1)$, and the algorithm will proceed to assign

Algorithm 11: Sequential Random Graph Generation

```
1  Procedure GENERATE-RANDOM-GRAPH(𝔻)
2      𝔼 ← ∅                                              // initially empty set of edges
3      while 𝔻 ≠ 0 do
4          Select the least u such that d_u is a minimal positive degree in 𝔻
5          while d_u ≠ 0 do
6              ℂ ← {v ≠ u : (u, v) ∉ 𝔼 ⋀ ⊖^𝔻_{u,v} is graphical}        // candidate list
7              v ← a random candidate in ℂ where probability of selecting v is proportional to d_v
8              𝔼 ← 𝔼 ∪ {(u, v)}
9              𝔻 ← ⊖^𝔻_{u,v}
10     Output 𝔼
```

edges incident on vertex v_5 and so on. One possible sequence of degree sequences is

$$(3,3,2,2,2) \to (3,3,1,2,1) \to (2,3,0,2,1) \to (2,2,0,2,0) \\ \to (1,2,0,1,0) \to (0,1,0,1,0) \to (0,0,0,0,0),$$

with the corresponding edge set

$$\mathbb{E} = \{(v_3, v_5), (v_3, v_1), (v_5, v_2), (v_1, v_4), (v_1, v_2), (v_2, v_4)\}.$$

The corresponding graph is shown in Figure 5.2. Note that during the assignment of incident edges on a vertex u, a candidate at a later stage is also a candidate at an earlier stage. The pseudocode of the algorithm is shown in Algorithm 11. Since a total of m edges are generated for the graph $\mathbb{G}$ and computing the candidate list (Line 6) for each edge takes $O(n^2)$ time, the time complexity of the algorithm is $O(mn^2)$.

Unlike many other graph generation algorithms, this method never gets stuck, i.e., it always terminates with a graph realizing the given degree sequence (proof provided in Theorem 3 in [20]) or creates loops or parallel edges through the computation of the candidate list using the Erdős-Gallai characterization. The algorithm can generate every possible graph with a positive probability (proof given in Corollary 1 in [20]). For additional details about the importance sampling and estimating the number of graphs for a given degree sequence, interested readers are referred to Sections 8 and 9 in [20].

5.3.2 Parallel Algorithm

To design an exact parallel version by maintaining the same stochastic process (in order to retain the same theoretical and mathematical results) as that of the sequential algorithm,

Algorithm 12: Parallel Random Graph Generation

Procedure PARALLEL-GENERATE-RANDOM-GRAPH($\mathbb{D}$)
  $\mathbb{E} \leftarrow \emptyset$ // initially empty set of edges
  while $\mathbb{D} \neq 0$ **do** // Assign edges until the degree of each vertex reduces to 0
    Select the least u such that d_u is a minimal positive degree in $\mathbb{D}$
    $\mathbb{C} \leftarrow \emptyset$ // candidate list
    while $d_u \neq 0$ **do** // Assign all d_u edges incident on u
      if $\mathbb{C} = \emptyset$ **then**
        $\mathbb{F} \leftarrow \{v \neq u : (u,v) \notin \mathbb{E} \ \wedge \ d_v > 0\}$
      else
        $\mathbb{F} \leftarrow \mathbb{C}$
        $\mathbb{C} \leftarrow \emptyset$
      for ***each*** $v \in \mathbb{F}$ ***in parallel*** **do** // Compute the candidate list
        $flag \leftarrow$ PARALLEL-ERDŐS-GALLAI($\ominus^{\mathbb{D}}_{u,v}$)
        if $flag = TRUE$ **then** // $\ominus^{\mathbb{D}}_{u,v}$ is a graphical degree sequence
          $\mathbb{C} \leftarrow \mathbb{C} \cup \{v\}$ // v is a candidate
      if $d_u = |\mathbb{C}|$ **then**
        for ***each*** $v \in \mathbb{C}$ ***in parallel*** **do**
          $\mathbb{E} \leftarrow \mathbb{E} \cup \{(u,v)\}$
          $\mathbb{D} \leftarrow \ominus^{\mathbb{D}}_{u,v}$
        break
      /* Assign an edge (u,v) from $\mathbb{C}$ */
      $v \leftarrow$ a random candidate in $\mathbb{C}$ where probability of selecting v is proportional to d_v
      $\mathbb{E} \leftarrow \mathbb{E} \cup \{(u,v)\}$
      $\mathbb{D} \leftarrow \ominus^{\mathbb{D}}_{u,v}$
      $\mathbb{C} \leftarrow \mathbb{C} - \{v\}$
  Output E // final set of edges

the vertices are considered (to assign their incident edges) in the same order in the parallel algorithm, i.e., in ascending order of their degree. Hence, we emphasize parallelizing the computation of the candidate list $\mathbb{C}$ of the sequential algorithm, i.e., Line 6 in Algorithm 11. For computing the candidate list to assign edges incident on a vertex u, we need to consider all other vertices v with non-zero degree d_v as potential candidates; and we parallelize this step. While considering a particular vertex v as a candidate, we need to check whether $\ominus^{\mathbb{D}}_{u,v}$ is a graphical sequence using the Erdős-Gallai characterization. If $\ominus^{\mathbb{D}}_{u,v}$ is graphical, then v is added to the candidate list $\mathbb{C}$. The time complexity of the best known sequential

algorithm for testing the Erdős-Gallai characterization is $O(n)$ [20, 59]. Thus to have an efficient parallel algorithm for generating random graphs, we need to use an efficient parallel algorithm for checking the Erdős-Gallai characterization. In Section 5.4, we present an efficient parallel algorithm for checking the Erdős-Gallai characterization that runs in $O\left(\frac{n}{\mathcal{P}} + \log \mathcal{P}\right)$ time. The parallel algorithm for the Erdős-Gallai characterization returns *TRUE* if the given degree sequence is graphical and *FALSE* otherwise.

Once the candidate list is computed, if the degree of u is equal to the cardinality $|\mathbb{C}|$ of the candidate list, then new edges are assigned between u and all candidate vertices v in the candidate list $\mathbb{C}$ in parallel. Otherwise, like the sequential algorithm, a candidate vertex v is chosen randomly from $\mathbb{C}$, a new edge (u, v) is assigned, the degree sequence $\mathbb{D}$ is updated by reducing the degree of each of u and v by 1, and the process is repeated until the degree d_u of vertex u is reduced to 0. After assigning all edges incident on vertex u, the algorithm proceeds with assigning edges incident on the next vertex having the minimum positive degree in $\mathbb{D}$ and so on. We present the pseudocode of our parallel algorithm for generating random graphs in Algorithm 12.

Theorem 1. *The parallel algorithm for generating random graphs maintains an exact stochastic process as that of the sequential algorithm and preserves all mathematical and theoretical results of the sequential algorithm.*

Proof. The parallel algorithm always selects the vertex u with the minimum degree in the sequence (Line 4), assigns d_u edges incident on u (Lines 6-24), and then proceeds with the next vertex in the sequence as the sequential algorithm would do. While assigning the first edge incident on a vertex u, all vertices in the sequence that do not create self-loops or parallel edges are considered as potential candidates $\mathbb{F}$ (Line 8), whereas for assigning the subsequent edges incident on u, the candidates $\mathbb{C}$ in an earlier stage are considered as the potential candidates $\mathbb{F}$ (Line 10) in a later stage. The candidate list is then computed in parallel by checking whether an edge can be assigned between u and a potential candidate v in $\mathbb{F}$ by checking whether the residual degree sequence $\ominus^{\mathbb{D}}_{u,v}$, if an edge (u, v) is assigned, is a graphical sequence by using the parallel algorithm for the Erdős-Gallai characterization (Lines 12-15). If the cardinality of the candidate list is equal to the degree d_u of vertex u, then edges are assigned between u and all vertices v in the candidate list $\mathbb{C}$ in parallel (Lines 16-20). Although this step is not explicitly mentioned in the sequential algorithm, this is obvious since the sequential algorithm would assign all d_u edges incident on u and there are no additional candidates other than the d_u candidates in $\mathbb{C}$. We parallelize this step to improve the performance of the algorithm. If the candidate list $\mathbb{C}$ has more than d_u candidates, then a vertex v is selected randomly from $\mathbb{C}$ with probability proportional to d_v, and an edge (u, v) is assigned, as the sequential algorithm would do. Hence, the parallel algorithm maintains an exact stochastic process as that of the sequential algorithm.

As a consequence, all mathematical and theoretical results (except the time complexity) of the sequential algorithm are applicable to our parallel algorithm as well. □

Theorem 2. *The time complexity of each of the core $\mathbb{P}_k$ in the parallel algorithm for generating random graphs is $O\left(mn\left(\frac{n}{\mathcal{P}} + \log \mathcal{P}\right)\right)$.*

Proof. The parallel algorithm assigns m edges one by one. To assign an edge incident on a vertex u, it computes the candidate list in parallel (Line 12). Whether a vertex v is a candidate is computed using the parallel algorithm for the Erdős-Gallai characterization (Line 13), which has a time complexity of $O\left(\frac{n}{\mathcal{P}} + \log \mathcal{P}\right)$. Hence, the time complexity of the parallel algorithm for generating random graphs is $O\left(mn\left(\frac{n}{\mathcal{P}} + \log \mathcal{P}\right)\right)$. □

Theorem 3. *The space complexity of the parallel algorithm for generating random graphs is $O(m + n)$.*

Proof. Storing the degree sequence and the edges take $O(n)$ and $O(m)$ space, respectively, making a space requirement of $O(m + n)$. □

5.3.3 Experimental Results

In this section, we present the data sets used in the experiments and the strong scaling and runtime of our parallel algorithm for generating random graphs. Moreover, we present a comparative study of several structural properties of the random graphs generated by our parallel algorithm with that of the real-world graphs and random graphs generated by the edge swapping method.

Experimental Setup. We use a 32-core Haswell-EP E5-2698 v3 2.30GHz (3.60GHz Turbo) dual processor node with 128GB of memory, 1TB internal hard drive, and QLogic QDR InfiniBand adapter. We use OpenMP version 3.1 and GCC version 4.7.2 for implementation.

Data Sets. We use degree sequences of six real-world networks for the experiments. A summary of the networks is given in Table 5.2. Facebook [100] is an anonymized Facebook friendship network of the students of CMU. The vertices are Facebook users and there is an edge between two users if they are Facebook friends. OpenFlights [93] is an air transportation network, where vertices are non-US-based airports and there is an edge between two vertices if there is a direct flight between them. PPI [104] is a protein-protein interaction network of *Drosophila melanogaster*, which is the common fruit fly. The vertices are proteins and there is an edge between two proteins if they interact. Internet [2] represents a snapshot of the structure of the Internet at the level of Autonomous Systems [77]. The vertices are Autonomous Systems (parts of the Internet infrastructure)

Table 5.2: *Data sets used in the experiments, where n, m, $\frac{2m}{n}$, Δ, and δ denote the no. of vertices, no. of edges, average degree, maximum degree, and minimum degree of the networks, respectively.* ***K*** *denotes thousands.*

Network	**Type**	n	m	$\frac{2m}{n}$	Δ	δ
Facebook [100]	Social	6.6**K**	250**K**	75.50	840	1
OpenFlights [93]	Transportation	2.94**K**	15.7**K**	10.67	242	1
PPI [104]	Biological	7.4**K**	25.6**K**	6.92	190	1
PowerGrid [2]	Power	4.94**K**	6.6**K**	2.67	19	1
Internet [2]	Internet	22.9**K**	48.4**K**	4.22	2.39**K**	1
CondMaterial [87]	Collaboration	16.3**K**	47.6**K**	5.85	107	1

and there is an edge between two Autonomous Systems if they exchange Internet traffic. The PowerGrid [2] network represents a high-voltage power grid in the Western states of the USA. The vertices are transformers, substations, and generators, and the edges are high-voltage transmission lines. CondMaterial [87] is a scientific collaboration network. The vertices are authors of scientific manuscripts on the condensed matter topic and there is an edge between two authors if they co-appear in the same scholarly article.

Strong Scaling. The strong scaling and runtime of the parallel algorithm are shown in Figures 5.3 and 5.4, respectively. The speedups increase almost linearly with the increase

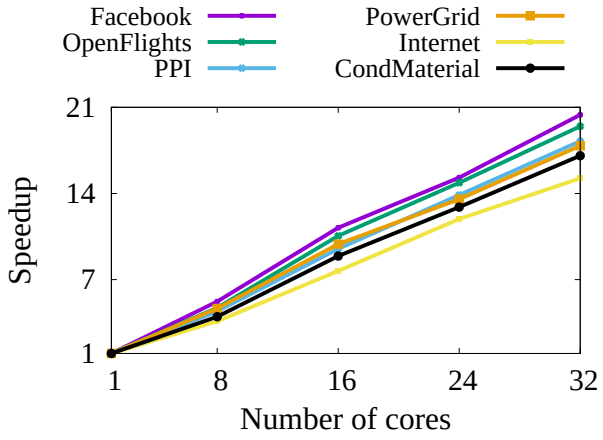


Figure 5.3: *Strong scaling of the parallel algorithm for generating random networks.*

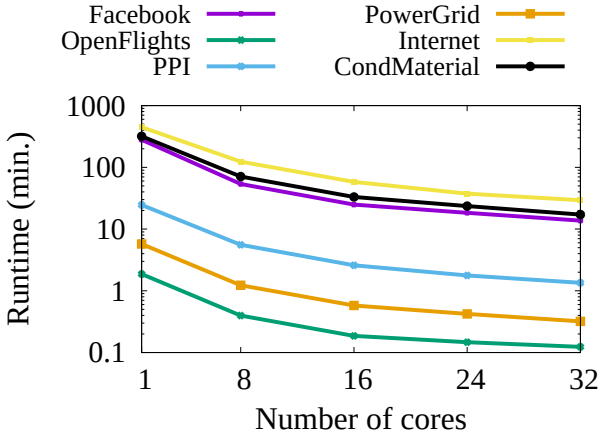


Figure 5.4: *Runtime of the parallel algorithm for generating random networks.*

in the number of cores, and we achieve a maximum speedup of 20.4 with 32 cores on the Facebook network.

Structural Properties of the Random Graphs. A comparison of some structural properties of the random graphs generated by our parallel algorithm with that of the real-world graphs and random graphs generated by swapping edges is given in Table 5.3. To generate random graphs by swapping edges, we perform $\frac{m}{2} \ln m$ edge swap operations to swap 100% edges [19] of the real-world graphs, as discussed in Chapter 3. We study the degree distributions, average shortest path lengths, and average clustering coefficients of the networks. Below we describe them in details.

Degree Distribution. The degree distributions of the real-world networks are illustrated in Figure 5.5. We observe a high fraction of vertices have low degree and a low fraction of vertices have high degree in all networks except for the PowerGrid network, which has a very low maximum degree of 19.

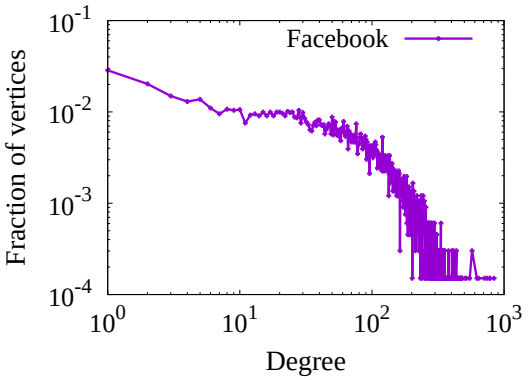
Clustering Coefficient. Figures 5.6 and 5.7 show the local and degree-dependent clustering coefficient distributions of the real-world and random networks. We observe that the real-world Facebook network has a good average clustering coefficient of 0.28, supporting the common fact that friends of friends are more likely to be friends with each other. By contrast, the random graphs have a low average clustering coefficient of 0.04 since they do not consider assigning more edges among the neighbors of the vertices. We observe a similar phenomenon in the OpenFlights and CondMaterial networks as well. With the progress of an edge swap process, the edges are replaced by random edges, which gradually destroys the triangles associated with the vertices, thus reducing the clustering coefficient. On the other hand, our parallel algorithm assigns edges randomly from the candidate set without considering assigning more edges among the neighbors of the vertices. The real-world PPI and PowerGrid networks have low clustering coefficients, and the corresponding random graphs have clustering coefficients close to 0. We observe some interesting results for the Internet network, where the real-world network has a clustering coefficient of 0.23. The Internet network has a very skewed degree distribution, as shown in Figure 5.5(e). It has many vertices with very low degree and a few vertices with very high degree. Almost 90% of the vertices have degree less than five, whereas six vertices have degree greater than 1000 and fifteen vertices have degree larger than 500. Note that it is easier for a low degree vertex to have a high local clustering coefficient than a high degree vertex because a fixed number of triangles associated with a low degree vertex contribute more to the local clustering coefficient compared to a high degree vertex associated with the same number of triangles. Since the real-world Internet network has more low degree vertices having high local clustering coefficient, as shown in Figure 5.7(e), it has a higher average clustering coefficient of 0.23 compared to that (0.11 – 0.12) of the corresponding

Table 5.3: *A comparison of some structural properties of the random networks generated, from the degree sequences of the real-world networks, by our parallel algorithm with that of the real-world networks and random networks generated by swapping 100% edges of the real-world networks. We use average values of 25 experiments with their standard deviations given in parentheses.*

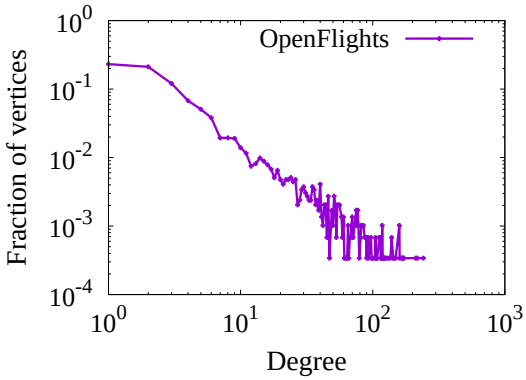
		Network structural properties	
Network	**Network model**	**Avg. shortest path length**	**Avg. clustering coefficient**
Facebook	Real-world	2.74	0.28
	Our algo.	2.5(0.001)	0.04(0.0006)
	Edge swap	2.49(0.001)	0.04(0.0003)
OpenFlights	Real-world	4.1	0.45
	Our algo.	3.3(0.009)	0.07(0.002)
	Edge swap	3.31(0.01)	0.08(0.004)
PPI	Real-world	4.34	0.01
	Our algo.	4.13(0.007)	0.007(0.0006)
	Edge swap	4.13(0.006)	0.007(0.0007)
PowerGrid	Real-world	18.99	0.08
	Our algo.	8.48(0.03)	0.0004(0.0003)
	Edge swap	8.5(0.03)	0.0004(0.00008)
Internet	Real-world	3.84	0.23
	Our algo.	3.8(0.01)	0.11(0.001)
	Edge swap	3.62(0.01)	0.12(0.002)
CondMaterial	Real-world	6.63	0.64
	Our algo.	4.91(0.004)	0.0012(0.0002)
	Edge swap	4.91(0.004)	0.001(0.0001)

random networks.

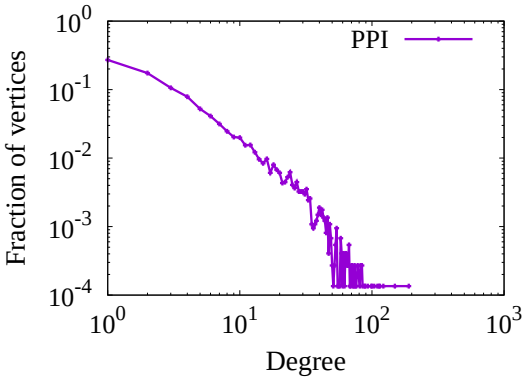
Average Shortest Path Length. A comparison of the average shortest path length distribu-



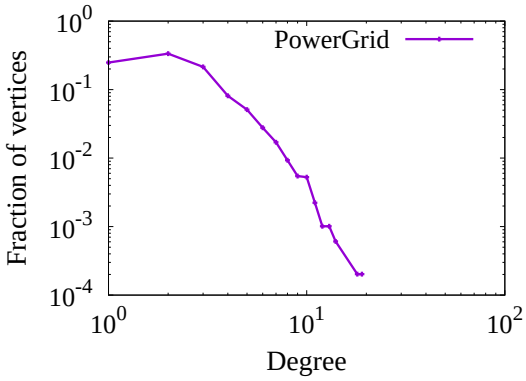
***(a)** Facebook*



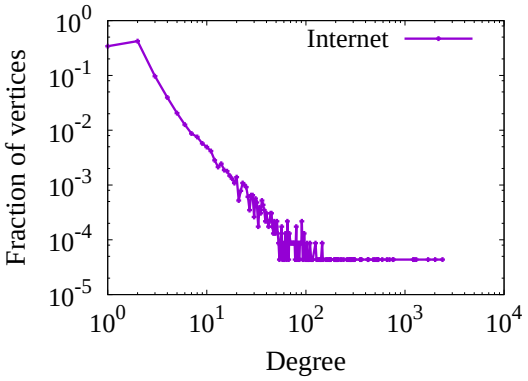
***(b)** OpenFlights*



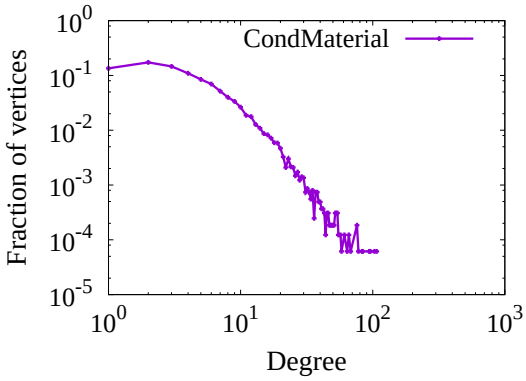
***(c)** PPI*



***(d)** PowerGrid*



***(e)** Internet*



***(f)** CondMaterial*

***Figure 5.5:** Degree distributions in the real-world graphs.*

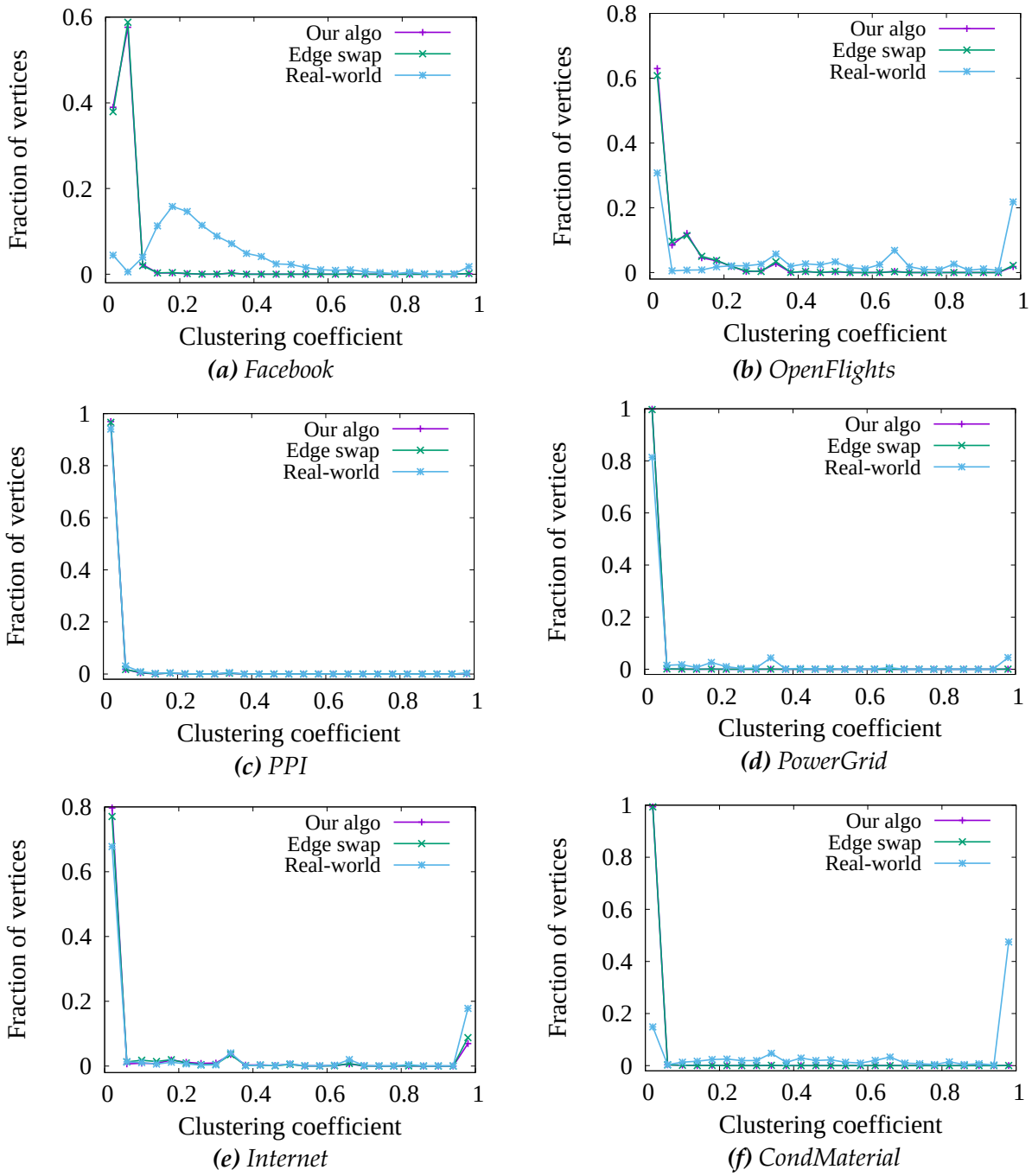


Figure 5.6: *A comparison of local clustering coefficient distributions in the real-world graphs with that in the random graphs generated by our parallel algorithm and by swapping edges of the real-world graphs. The distributions in the random graphs almost completely overlap with each other.*

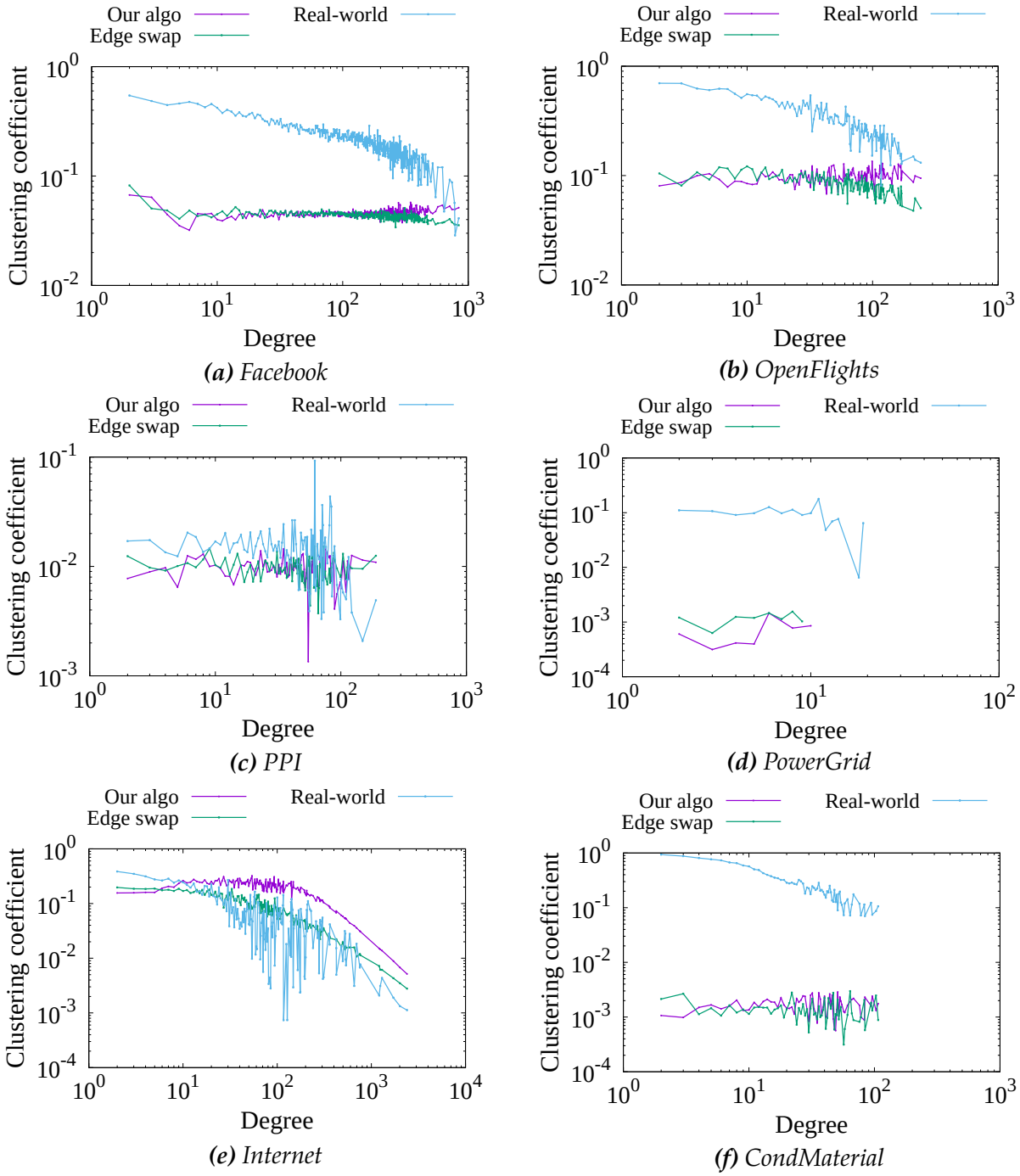


Figure 5.7: *A comparison of degree-dependent clustering coefficient of vertices in the real-world graphs with that in the random graphs generated by our parallel algorithm and by swapping edges of the real-world graphs.*

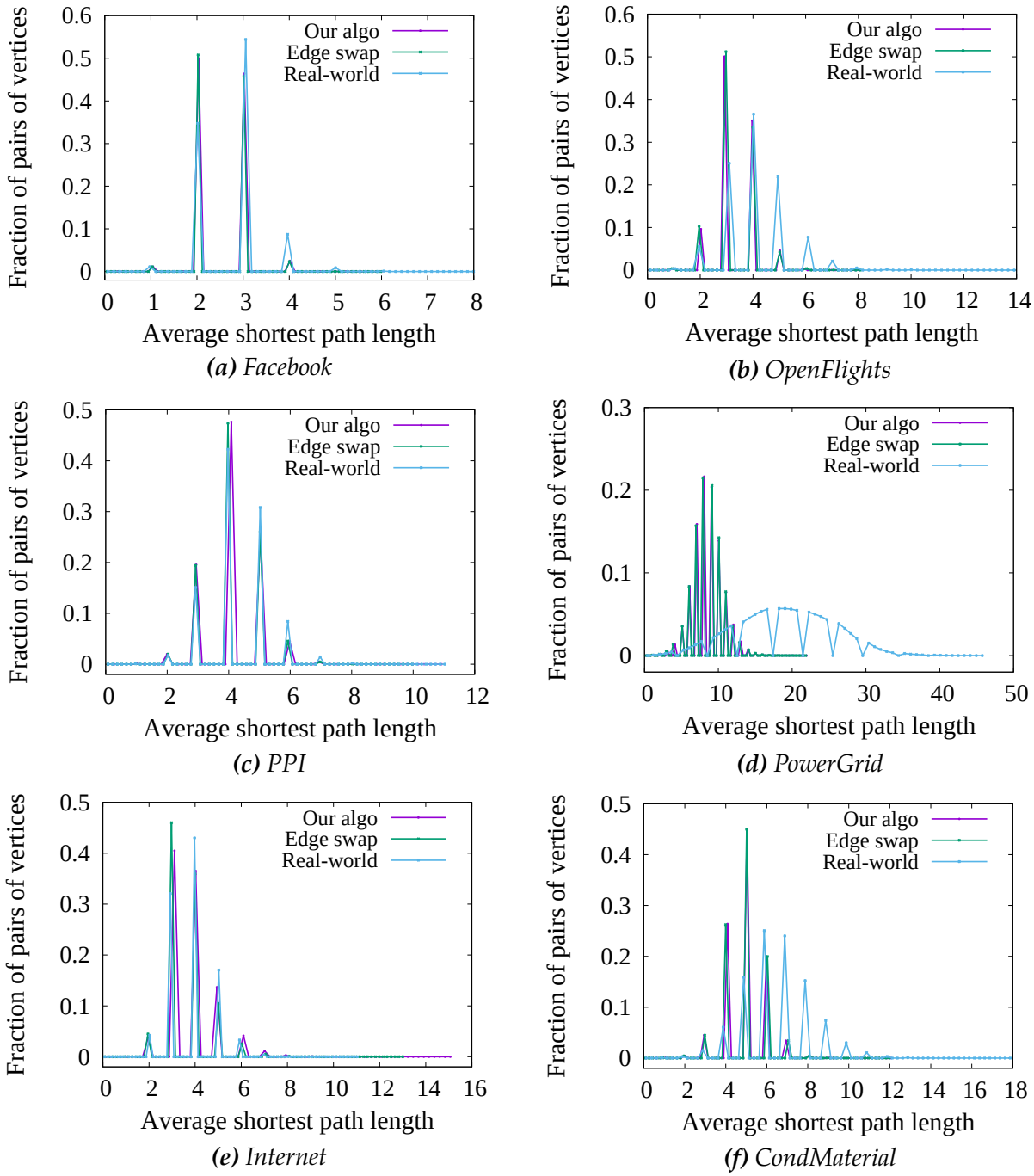


Figure 5.8: *A comparison of average shortest path length distributions in the real-world graphs with that in the random graphs generated by our parallel algorithm and by swapping edges of the real-world graphs.*

tions is shown in Figure 5.8. We observe low average shortest path lengths in the real-world networks, except for the PowerGrid network. The real-world Facebook network has a very high average degree and a good connectivity among the vertices, hence a low average shortest path length of 2.74. In the random networks, vertices get connected randomly by shorter paths, hence reducing the average shortest path length further. For example, the random networks corresponding to the Facebook network have average shortest path length around 2.50. We observe a similar phenomenon of reducing the average shortest path length in all the random networks. The PowerGrid network has a tree-like structure having a very low average degree of 2.67. As a result, the average shortest path length 18.99 is pretty high in the real-world PowerGrid network, and the corresponding random networks have that value around 8.50, which is much higher than that of the other random networks.

5.4 Parallel Algorithm for Checking the Erdős-Gallai Characterization

Many variants of the Erdős-Gallai characterization have been developed and proofs have been given (see [20] and [76] for a good discussion). Such a useful result has been presented in Theorem 3.4.1 in [76], which defines the *corrected Durfee number* C of the degree sequence $\mathbb{D} = (d_1, d_2, \ldots, d_n)$ (sorted in non-increasing order) as

$$C = \left|\{j : d_j \geq j - 1\}\right| \tag{5.3}$$

and showed that $\mathbb{D}$ is graphical if and only if it satisfies the first C inequalities of the Erdős-Gallai test. The corrected Durfee number C is often significantly smaller than the number of vertices n. For example, for the degree sequence $\mathbb{D} = (3, 2, 2, 2, 1)$, the corrected Durfee number C is 3, as shown in Figure 5.9; hence, it is sufficient to check only the first three Erdős-Gallai inequalities instead of checking all five inequalities of Eq. (5.2).

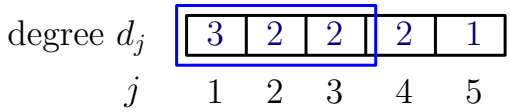


Figure 5.9: *For the degree sequence $\mathbb{D} = (3, 2, 2, 2, 1)$, the corrected Durfee number $C = 3$.*

The sequential algorithm for checking the Erdős-Gallai characterization is quite straightforward and has a time complexity of $O(n)$ [59]. A parallel algorithm for the problem has been presented in [40], which has a runtime of $O(\log n)$ using $O\left(\frac{n}{\log n}\right)$ EREW PRAM

processors. In addition to generating random graphs, the Erdős-Gallai characterization has important applications in many other graph theory problems as well. For example, Iványi et al. [60] applied the sequential algorithm for checking the Erdős-Gallai characterization to enumerate the distinct degree sequences of simple graphs in parallel. In this section, we present a shared-memory parallel algorithm for checking the Erdős-Gallai characterization of a given degree sequence with a time complexity of $O\left(\frac{n}{\mathcal{P}} + \log \mathcal{P}\right)$ using $\mathcal{P}$ processing cores. First we briefly review the current state-of-the-art sequential algorithm.

5.4.1 Sequential Algorithm

The sequential algorithm [59] is quite simple and consists of the following steps: (***i***) compute the corrected Durfee number C, (***ii***) compute the prefix sum of the degrees, (***iii***) check the parity, i.e., whether the sum of the degrees is even or odd, (***iv***) compute the weights, which are useful in computing the right hand side of Eq. (5.2) in linear time, and (***v***) check the first C Erdős-Gallai inequalities. If the sum of the degrees is even and all the inequalities are satisfied, then the degree sequence $\mathbb{D}$ is graphical; otherwise, $\mathbb{D}$ is not graphical. The pseudocode of the sequential algorithm is given in Algorithm 13.

5.4.2 Parallel Algorithm

Based on the sequential algorithm presented in Algorithm 13, we present a parallel algorithm for checking the Erdős-Gallai characterization. Below we describe the methodology to parallelize the steps of the sequential algorithm.

• ***Step 1*: Compute the Corrected Durfee Number.** The corrected Durfee number can be computed in parallel in a round robin fashion, as shown in Algorithm 14. Each core $\mathbb{P}_k$ computes its local corrected Durfee number C_k. Then all the cores synchronize and the maximum value of all C_k is reduced as the corrected Durfee number C.

• ***Step 2*: Compute the Prefix Sum of the Degrees.** We use a parallel version [6] of computing the prefix sum, as shown in Algorithm 15. Each core $\mathbb{P}_k$ works on a chunk of size $\left\lceil \frac{n}{\mathcal{P}} \right\rceil$ of the degree sequence. First, the sum s_k of the degrees in the chunk is computed (Line 5) and then a prefix sum S_k of the s_j $(0 \leq j \leq k-1)$ is computed in parallel (Line 6). Finally, each core gives a pass to the chunk and uses the value of S_k to compute the final prefix sum (Lines 7-10).

• ***Step 3*: Check the Parity.** The master core checks whether the sum of the degrees is even. If the sum is odd, then the degree sequence is not graphical. Otherwise, the algorithm proceeds to the next step.

Algorithm 13: Sequential Erdős-Gallai Characterization

```
1  Procedure SEQUENTIAL-ERDŐS-GALLAI(𝔻)
       /* Compute the corrected Durfee number                          */
2      i ← 1
3      while i <= n and d_i ≥ i − 1 do
4          C ← i                                   // corrected Durfee number
5          i ← i + 1
       /* Compute the prefix sum of the degrees                       */
6      H_0 ← 0
7      for i = 1 to n do
8          H_i ← H_{i−1} + d_i
       /* Check the parity of the sum of the degrees                  */
9      if H_n%2 = 1 then
10         return FALSE                         // not a graphical degree sequence
       /* Compute the weights                                         */
11     d_0 ← n − 1
12     for i = 1 to n do
13         if d_i < d_{i−1} then
14             for j = d_{i−1} downto d_i + 1 do
15                 w_j ← i − 1
16             w_{d_i} ← i
17     for j = d_n downto 1 do
18         w_j ← n
       /* Check the Erdős-Gallai inequalities                         */
19     for i = 1 to C do
20         if i ≤ w_i then
21             if H_i > i(i − 1) + i(w_i − i) + H_n − H_{w_i} then
22                 return FALSE                 // not a graphical degree sequence
23         else if H_i > i(i − 1) + H_n − H_i then
24             return FALSE                     // not a graphical degree sequence
25     return TRUE                              // a graphical degree sequence
```

• ***Step 4*: Compute the Weights.** The pseudocode of computing the weights in parallel is shown in Algorithm 16. We first initialize (Lines 3-4) the weight array w in parallel. Then the actual weights are computed inside a `for` loop (Lines 5-11) in parallel. Due to the simultaneous nature of the parallel algorithm, there is a possibility that the same weight

Algorithm 14: Parallel Computation of the Corrected Durfee Number

```
1  Procedure Parallel-Corrected-Durfee-No(𝔻)
2      k ← core id
       /* Each core ℙ_k executes the following in parallel:                    */
3      i ← k + 1
4      C_k ← 0
5      while i <= n and d_i ≥ i − 1 do
6          C_k ← i                                  // local corrected Durfee number
7          i ← i + 𝒫
       /* Reduce the corrected Durfee number                                   */
8      C ← Reduce-Max_k C_k
9      return C
```

Algorithm 15: Parallel Computation of the Prefix Sum Degrees

```
1  Procedure Parallel-Prefix-Sum-Degree(𝔻)
2      k ← core id
       /* Each core ℙ_k executes the following in parallel:                    */
3      x ← k⌈n/𝒫⌉ + 1
4      y ← min{(k + 1)⌈n/𝒫⌉, n}
5      s_k ← Σ_{i=x}^{y} d_i
6      In Parallel: S_k ← Σ_{j=0}^{k−1} s_j
7      Q ← S_k                                  // note that S_0 = 0
8      for i = x to y do
9          H_i ← Q + d_i
10         Q ← H_i
11     return H
```

w_j may be updated by multiple cores in an order different than that of the sequential algorithm. To deal with this difficulty, we add two additional `if` conditions (Lines 8 and 10) as the values of w_j are only updated with larger values in the sequential algorithm. These two conditions ensure the correctness of the weight values as well as allow simultaneous parallel computation of them. Finally, the larger weights are computed in parallel in the last `for` loop (Lines 12-13).

• ***Step 5*: Check the Erdős-Gallai Inequalities.** The Erdős-Gallai inequalities can be checked in parallel in a round robin fashion. We have to check only the first C inequalities instead of checking all the n inequalities. This significantly improves the performance of

Algorithm 16: Parallel Computation of the Weights

```
1  Procedure PARALLEL-WEIGHTS(𝔻)
2      d_0 ← n − 1
       /* Initialize the weight array                                  */
3      for i = 1 to n in parallel do
4          w_i ← 0
       /* Compute the weight values                                    */
5      for i = 1 to n in parallel do
6          if d_i < d_{i−1} then
7              for j = d_{i−1} downto d_i + 1 in parallel do
8                  if i − 1 > w_j then
9                      w_j ← i − 1
10             if i > w_{d_i} then
11                 w_{d_i} ← i
       /* Compute the larger weight values                             */
12     for j = d_n downto 1 in parallel do
13         w_j ← n
14     return w
```

the algorithm since $C << n$ in many degree sequences, as shown later in Table 5.4. If any of the inequalities is dissatisfied, then the degree sequence is not graphical; otherwise, it is a graphical sequence. The pseudocode of the algorithm is presented in Algorithm 17.

Theorem 4. *The time complexity of each of the core $\mathbb{P}_k$ in the parallel algorithm for checking the Erdős-Gallai characterization is $O\left(\frac{n}{\mathcal{P}} + \log \mathcal{P}\right)$.*

Proof. The `while` loop in Lines 5-7 in Algorithm 14 takes $O\left(\frac{C}{\mathcal{P}}\right)$ time, where C is the corrected Durfee number and $\mathcal{P}$ is the number of cores. The `reduction` in Line 8 takes $O\left(\log \mathcal{P}\right)$ time. Hence, the corrected Durfee number in Step 1 can be computed in $O\left(\frac{C}{\mathcal{P}} + \log \mathcal{P}\right)$ time. Lines 5, 6, and 8-10 in Algorithm 15 take $O\left(\frac{n}{\mathcal{P}}\right)$, $O\left(\log \mathcal{P}\right)$, and $O\left(\frac{n}{\mathcal{P}}\right)$ time, respectively, where n is the number of vertices. So, the prefix sum of the degrees in Step 2 can be computed in $O\left(\frac{n}{\mathcal{P}} + \log \mathcal{P}\right)$ time [6]. Checking the parity in Step 3 can be done in $O(1)$ time. Each of the three `for` loops (Lines 3-4, 5-11, and 12-13) in Algorithm 16 takes $O\left(\frac{n}{\mathcal{P}}\right)$ time. Although the two `for` loops in Lines 5 and 7 are nested, the total number of weights updated are $O\left(n\right)$. Thus computing the weights in Step 4 takes $O\left(\frac{n}{\mathcal{P}}\right)$ time. The `while` loop (Lines 5-11) in Algorithm 17 takes $O\left(\frac{C}{\mathcal{P}}\right)$ time. Therefore, the Erdős-Gallai inequalities in Step 5 are tested in $O\left(\frac{C}{\mathcal{P}}\right)$ time. Thus, the time complexity of the algorithm

Algorithm 17: Parallel Checking of the Erdős-Gallai Inequalities

```
1  Procedure Parallel-Erdős-Gallai-Check(H, w, C)
2      k ← core id
3      flag ← TRUE                                          // shared variable
       /* Each core P_k executes the following in parallel:                  */
4      i ← k + 1
5      while i <= C and flag = TRUE do
6          if i ≤ w_i then
7              if H_i > i(i − 1) + i(w_i − i) + H_n − H_{w_i} then
8                  flag ← FALSE
9          else if H_i > i(i − 1) + H_n − H_i then
10             flag ← FALSE
11         i ← i + P
12     Omp-Barrier
13     return flag
```

is $O\left(\frac{n}{\mathcal{P}} + \frac{C}{\mathcal{P}} + \log \mathcal{P}\right) = O\left(\frac{n}{\mathcal{P}} + \log \mathcal{P}\right)$. □

Theorem 5. *The space complexity of the parallel algorithm for checking the Erdős-Gallai characterization is $O(n)$.*

Proof. Storing the degree sequence and the prefix sum of the degrees take $O(n)$ space. □

5.4.3 Performance Evaluation

In this section, we present the data sets used in the experiments and the strong scaling and runtime of our parallel algorithm for checking the Erdős-Gallai characterization. We use the same experimental setup as described before in Section 5.3.3.

Data Sets. We use degree sequences of both artificial and real-world networks for the experiments. A summary of the networks is given in Table 5.4. Friendster, Twitter, and LiveJournal (LJ) are real-world online social networks [71]. New York (NY) and Los Angeles (LA) are synthetic, yet realistic social contact networks [12]. The SmallWorld random network follows the Watts-Strogatz small world network model [111]. Table 5.4 also shows that the corrected Durfee number C is significantly smaller than the number of vertices n for all six networks.

Strong Scaling. The strong scaling and runtime of the parallel algorithm are illustrated in

Table 5.4: *Data sets used in the experiments, where n, m, $\frac{2m}{n}$, and C denote the number of vertices, number of edges, average degree, and the corrected Durfee number of the networks, respectively.* ***M*** *and* ***B*** *denote millions and billions, respectively.*

Network	**Type**	n	m	$\frac{2m}{n}$	C
Friendster [71]	Social	65.6**M**	1.8**B**	55.06	2959
Twitter [71]	Social	40.56**M**	667.7**M**	32.93	6842
Los Angeles (LA) [12]	Contact	16.23**M**	459.3**M**	56.59	380
New York (NY) [12]	Contact	17.88**M**	480.1**M**	53.70	387
LiveJournal (LJ) [71]	Social	4.80**M**	42.85**M**	17.68	990
SmallWorld (SW) [111]	Random	4.80**M**	48.00**M**	20.00	31

Figures 5.10 and 5.11, respectively. The speedup increases almost linearly with the increase in the number of cores. We observe better speedups for the degree sequences of larger graphs and achieve a maximum speedup of 23 with 32 cores on the Friendster graph.

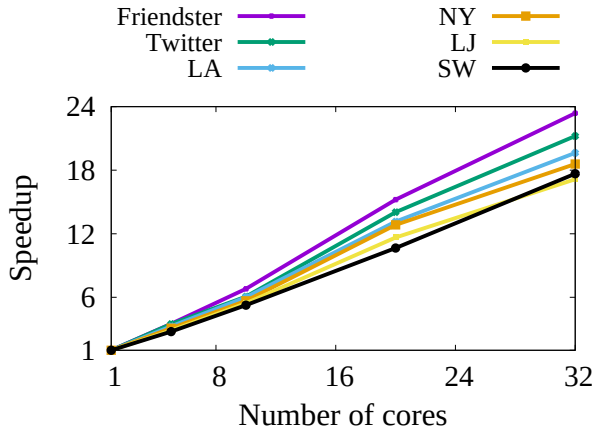


Figure 5.10: *Strong scaling of the parallel algorithm for checking the Erdős-Gallai characterization.*

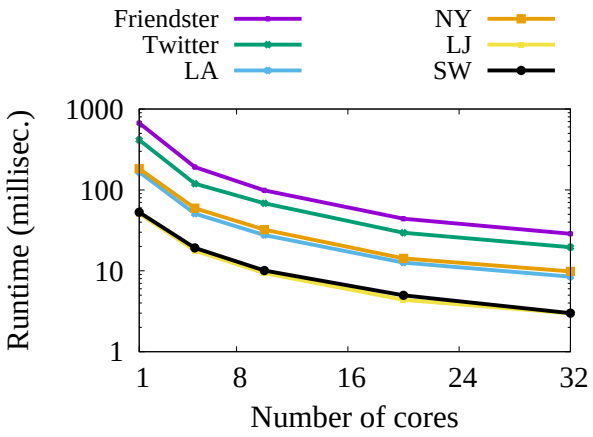


Figure 5.11: *Runtime of the parallel algorithm for checking the Erdős-Gallai characterization.*

5.5 Conclusion

We presented an efficient parallel algorithm for generating random graphs with prescribed degree sequences. It can be used in studying various structural properties of and dynamics over a network, sampling graphs uniformly at random from the graphical realizations of a given degree sequence and estimating the number of possible graphs with a given degree sequence. The algorithm never gets stuck, can generate every possible graph with a positive probability, and exhibits good speedup. We also compared several important structural properties of the random graphs generated by our parallel algorithm with that of the real-world graphs and random graphs generated by the edge swapping method. In addition, we developed an efficient parallel algorithm for checking the Erdős-Gallai characterization of a given degree sequence. This algorithm can be of independent interest and prove useful in parallelizing many other graph theory problems. We believe the parallel algorithms will contribute significantly in analyzing and mining emerging complex systems and understanding interesting characteristics of such networks.

Chapter 6

Concluding Remarks

We present efficient distributed memory parallel algorithms for switching edges in a network. Edge switch is a very important problem in mining and analyzing networks and can be used in modeling and simulation, generating random graphs by preserving the degrees of the vertices in a given network, and in studying various properties of dynamic networks. We studied several partitioning schemes in conjunction with the parallel algorithm to obtain the best performance through a well-balanced load distribution among the processors. One of the steps in our parallel algorithm requires generating multinomial random variables in parallel, and we present here the first non-trivial parallel algorithm for the problem. We also present a shared memory parallel algorithm for 1-Flipper, which maintains the connectivity of the graph while switching edges. An important variant of the edge switch problem is assortative edge switch. Assortative edge switch preserves the mixing pattern of the vertices based on the vertex labels in addition to preserving the degrees of the vertices in a given network. The algorithm for edge switch performs poor for the assortative edge switch due to the difficulty of maintaining the mixing pattern. We address this issue by presenting efficient sequential and distributed memory parallel algorithms for assortative edge switch in a labeled network. Our parallel algorithms show good speedups and scale to a large number of processors. We also present a shared memory parallel algorithm for generating a random network with a prescribed degree sequence with direct graph construction method. One of the steps in the parallel algorithm requires checking the graphicality of a given degree sequence using the Erdős-Gallai characterization in parallel. We also present a shared memory parallel algorithm for checking the Erdős-Gallai characterization of a given degree sequence. In addition, we present an extensive comparative study of several important structural properties of the random networks generated by our parallel algorithm with that of the random networks generated by our parallel algorithms for switching edges and that of the real-world networks. We believe the proposed parallel algorithms will prove useful in analyzing and mining large networks.

Bibliography

[1] Sherif Abdelhamid, Maksudul Alam, Richard Alo, Shaikh Arifuzzaman, Pete Beckman, Tirtha Bhattacharjee, Hasanuzzaman Bhuiyan, Keith Bisset, Stephen Eubank, Albert C Esterline, Edward A Fox, Geoffrey C Fox, et al. "CINET 2.0: A cyberinfrastructure for network science." In: *Proceedings of the 10th International Conference on E-Science (e-Science)*. IEEE. 2014, pp. 324–331 (page 56).

[2] Sherif Elmeligy Abdelhamid, Richard Alo, SM Arifuzzaman, Pete Beckman, Md Hasanuzzaman Bhuiyan, Keith Bisset, Edward A Fox, Geoffrey Charles Fox, et al. "CINET: A cyberinfrastructure for network science." In: *Proceedings of the 8th International Conference on E-Science (e-Science)*. IEEE. 2012, pp. 1–8 (pages 81, 82).

[3] Ilan Adler, Shmuel Oren, and Sheldon M Ross. "The coupon-collector's problem revisited." In: *Journal of Applied Probability* 40.2 (2003), pp. 513–518 (page 14).

[4] Sarita V Adve and Kourosh Gharachorloo. "Shared memory consistency models: A tutorial." In: *Computer* 29.12 (1996), pp. 66–76 (page 2).

[5] Maksudul Alam and Maleq Khan. "Parallel algorithms for generating random networks with given degree sequences." In: *International Journal of Parallel Programming* 45.1 (2017), pp. 109–127 (pages 5, 74).

[6] Srinivas Aluru. "Teaching parallel computing through parallel prefix." In: *The International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012 (pages 90, 93).

[7] *Apache Giraph*. `http://giraph.apache.org/`. [Online; accessed 28-Jun-2014] (pages 1, 2).

[8] *Apache Spark*. `http://spark.apache.org/`. [Online; accessed 28-Jun-2014] (page 1).

[9] Chidanand Apte, Bing Liu, Edwin PD Pednault, and Padhraic Smyth. "Business applications of data mining." In: *Communications of the ACM* 45.8 (2002), pp. 49–53 (page 1).

[10] Srinivasa R Arikati and Anil Maheshwari. "Realizing degree sequences in parallel." In: *SIAM Journal on Discrete Mathematics* 9.2 (1996), pp. 317–338 (page 73).

[11] Albert-László Barabási and Réka Albert. "Emergence of scaling in random networks." In: *Science* 286.5439 (1999), pp. 509–512 (pages 30, 56, 72).

[12] Christopher L Barrett, Richard J Beckman, Maleq Khan, VS Anil Kumar, Madhav V Marathe, Paula E Stretz, Tridib Dutta, and Bryan Lewis. "Generation and analysis of large synthetic social contact networks." In: *Proceedings of the 2009 Winter Simulation Conference (WSC)*. 2009, pp. 1003–1014 (pages 29, 56, 94, 95).

[13] Kevin E Bassler, Charo I Del Genio, Péter L Erdős, István Miklós, and Zoltán Toroczkai. "Exact sampling of graphs with prescribed degree correlations." In: *New Journal of Physics* 17.8 (2015), p. 083052 (page 54).

[14] Mohsen Bayati, Jeong Han Kim, and Amin Saberi. "A sequential algorithm for generating random graphs." In: *Algorithmica* 58.4 (2010), pp. 860–910 (pages 4, 72, 73).

[15] Claude Berge and Edward Minieka. *Graphs and Hypergraphs*. North-Holland Publishing Company, 1973 (pages 4, 72).

[16] Hasanuzzaman Bhuiyan, Maleq Khan, and Madhav Marathe. "A Parallel Algorithm for Generating a Random Graph with a Prescribed Degree Sequence." In: *arXiv preprint arXiv:1708.07290* (2017) (page vi).

[17] Hasanuzzaman Bhuiyan, Maleq Khan, and Madhav Marathe. "Efficient Algorithms for Assortative Edge Switch in Large Labeled Networks." In: *25th High Performance Computing Symposium (HPC 2017)*. 2017 (page vi).

[18] Hasanuzzaman Bhuiyan, Jiangzhuo Chen, Maleq Khan, and Madhav V Marathe. "Fast parallel algorithms for edge-switching to achieve a target visit rate in heterogeneous graphs." In: *Proceedings of the 43rd International Conference on Parallel Processing*. IEEE. 2014, pp. 60–69 (pages vi, 73).

[19] Hasanuzzaman Bhuiyan, Maleq Khan, Jiangzhuo Chen, and Madhav Marathe. "Parallel algorithms for switching edges in heterogeneous graphs." In: *Journal of Parallel and Distributed Computing* 104 (2017), pp. 19–35 (pages vi, 83).

[20] Joseph Blitzstein and Persi Diaconis. "A sequential importance sampling algorithm for generating random graphs with prescribed degrees." In: *Internet Mathematics* 6.4 (2011), pp. 489–522 (pages 4, 5, 11, 72–74, 77, 78, 80, 89).

[21] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. "Complex networks: Structure and dynamics." In: *Physics Reports* 424.4 (2006), pp. 175–308 (pages 4, 72).

[22] Paolo Boldi and Sebastiano Vigna. "The webgraph framework I: Compression techniques." In: *Proceedings of the 13th International Conference on World Wide Web*. ACM. 2004, pp. 595–602 (page 1).

[23] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks." In: *Proceedings of the 20th International Conference on World Wide Web*. ACM. 2011, pp. 587–596 (page 1).

[24] Béla Bollobás. *Random Graphs*. Springer, 1998 (pages 4, 30, 56, 72, 73).

[25] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph Theory with Applications*. Vol. 290. Macmillan Publishers, 1976 (page 6).

[26] Clay P Breshears and Phu Luong. "Comparison of OpenMP and Pthreads within a coastal ocean circulation model code." In: *Workshop on OpenMP Applications and Tools*. 2000 (page 2).

[27] Michele Catanzaro, Guido Caldarelli, and Luciano Pietronero. "Social network growth with assortative mixing." In: *Physica A: Statistical Mechanics and its Applications* 338.1 (2004), pp. 119–124 (pages 4, 53, 54).

[28] Rohit Chandra. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001 (page 2).

[29] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. Vol. 10. MIT press, 2008 (page 2).

[30] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. "One trillion edges: Graph processing at Facebook-scale." In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1804–1815 (pages 1, 2).

[31] Shumo Chu and James Cheng. "Triangle listing in massive networks and its applications." In: *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2011, pp. 672–680 (page 1).

[32] Fan Chung and Linyuan Lu. "The average distances in random graphs with given expected degrees." In: *Proceedings of the National Academy of Sciences* 99.25 (2002), pp. 15879–15882 (page 72).

[33] Brian Cloteaux. "Fast sequential creation of random realizations of degree sequences." In: *Internet Mathematics* 12.3 (2016), pp. 205–219 (pages 4, 72).

[34] Colin Cooper, Martin Dyer, and Catherine Greenhill. "Sampling regular graphs and a peer-to-peer network." In: *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2005, pp. 980–988 (pages 3, 11, 12, 73).

[35] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd ed. MIT press, 2009 (pages 21, 22).

[36] Leonardo Dagum and Rameshm Enon. "OpenMP: An industry standard API for shared-memory programming." In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55 (page 2).

[37] Charles S Davis. "The computer generation of multinomial random variates." In: *Computational Statistics & Data Analytics* 16.2 (1993), pp. 205–217 (pages 25, 43, 44, 61).

[38] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters." In: *Communications of the ACM* 51.1 (2008), pp. 107–113 (page 1).

[39] Charo I Del Genio, Hyunju Kim, Zoltán Toroczkai, and Kevin E Bassler. "Efficient and exact sampling of simple graphs with given arbitrary degree sequence." In: *PLoS One* 5.4 (2010), e10012 (pages 4, 72, 73).

[40] Anders Dessmark, Andrzej Lingas, and Oscar Garrido. "On parallel complexity of maximum f-matching and the degree sequence problem." In: *Mathematical Foundations of Computer Science* (1994), pp. 316–325 (pages 74, 89).

[41] Rick Durrett. *Probability: Theory and Examples*. Cambridge University Press, 2010 (page 6).

[42] David Easley and Jon Kleinberg. *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Cambridge University Press, 2010 (page 1).

[43] P Erdős and T Gallai. "Gráfok előírt fokú pontokkal (Graphs with points of prescribed degrees, in Hungarian)." In: *Mat. Lapok* 11 (1961), pp. 264–274 (pages 75, 76).

[44] Stephen Eubank, Anil Vullikanti, Maleq Khan, Madhav V Marathe, and Christopher L Barrett. "Beyond degree distributions: Local to global structure of social contact graphs." In: *Proceedings of the Third International Conference on Social Computing, Behavioral Modeling, and Prediction (SBP)*. 2010, p. 1 (pages 3, 12, 54).

[45] Tomás Feder, Adam Guetz, Milena Mihail, and Amin Saberi. "A local switch Markov chain on given degree graphs with application in connectivity of peer-to-peer networks." In: *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*. IEEE. 2006, pp. 69–76 (pages 3, 11, 12, 73).

[46] Willliam Feller. *An Introduction to Probability Theory and Its Applications*. Vol. 2. John Wiley & Sons, 2008 (page 6).

[47] Michelle Girvan and Mark EJ Newman. "Community structure in social and biological networks." In: *Proceedings of the National Academy of Sciences* 99.12 (2002), pp. 7821–7826 (page 72).

[48] Christos Gkantsidist, Milena Mihail, and Ellen Zegura. "The Markov chain simulation method for generating connected power law random graphs." In: *Proceedings of the Fifth Workshop on Algorithm Engineering and Experiments*. Vol. 111. SIAM. 2003, pp. 16–25 (pages 3, 11, 12, 73).

[49] Sandra González-Bailón, Javier Borge-Holthoefer, Alejandro Rivero, and Yamir Moreno. "The dynamics of protest recruitment through an online network." In: *Scientific Reports* 1.197 (2011), pp. 1–7 (page 1).

[50] Ananth Grama. *Introduction to Parallel Computing*. Pearson Education, 2003 (page 61).

[51] *GraphX*. `http://spark.apache.org/graphx/`. [Online; accessed 28-Jun-2014] (page 2).

[52] Douglas Gregor and Andrew Lumsdaine. "The parallel BGL: A generic library for distributed graph computations." In: *Parallel Object-Oriented Scientific Computing (POOSC)* 2 (2005), pp. 1–18 (pages 19, 20).

[53] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Vol. 1. MIT press, 1999 (pages 1, 2).

[54] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT press, 1999 (pages 1, 2).

[55] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. "A high-performance, portable implementation of the MPI message passing interface standard." In: *Parallel Computing* 22.6 (1996), pp. 789–828 (pages 1, 2).

[56] Aric Hagberg, Pieter Swart, and Daniel Schult. "Exploring network structure, dynamics, and function using NetworkX." In: *Proceedings of the 7th Python in Science Conferences (SciPy)*. 2008, pp. 11–16 (pages 12, 54).

[57] S Louis Hakimi. "On realizability of a set of integers as degrees of the vertices of a linear graph. I." In: *Journal of the Society for Industrial and Applied Mathematics* 10.3 (1962), pp. 496–506 (pages 3, 12, 73, 76).

[58] S Louis Hakimi and Edward F Schmeichel. "Graphs and their degree sequences: A survey." In: *Theory and Applications of Graphs*. Springer, 1978, pp. 225–235 (pages 4, 72).

[59] Antal Iványi and Loránd Lucz. "Erdős-Gallai test in linear time." In: *Combinatorica* (2011) (pages 80, 89, 90).

[60] Antal Iványi, Lóránd Lucz, Péter Sótér, and Shariefuddin Pirzada. "Parallel Erdős-Gallai algorithm." In: *Central European Journal of Operations Research* (2011) (page 90).

[61] Mark Jerrum and Alistair Sinclair. "Fast uniform generation of regular graphs." In: *Theoretical Computer Science* 73.1 (1990), pp. 91–100 (pages 3, 11).

[62] Voratas Kachitvichyanukul and Bruce W Schmeiser. "Binomial random variate generation." In: *Communications of the ACM* 31.2 (1988), pp. 216–222 (page 44).

[63] Ravi Kannan, Prasad Tetali, and Santosh Vempala. "Simple Markov-chain algorithms for generating bipartite graphs and tournaments." In: *Random Structures and Algorithms* 14.4 (1999), pp. 293–308 (pages 3, 11, 12).

[64] Ulas Karaoz, TM Murali, Stan Letovsky, Yu Zheng, Chunming Ding, Charles R Cantor, and Simon Kasif. "Whole-genome annotation by using evidence integration in functional-linkage networks." In: *Proceedings of the National Academy of Sciences* 101.9 (2004), pp. 2888–2893 (page 1).

[65] Richard M Karp and Vijaya Ramachandran. "A survey of parallel algorithms for shared-memory machines." In: *Handbook of Theoretical Computer Science* (1988) (pages 2, 73).

[66] George Karypis, Kirk Schloegel, and Vipin Kumar. "ParMETIS: Parallel graph partitioning and sparse matrix ordering library." In: *Version 1.0, Dept. of Computer Science, University of Minnesota* (1997) (pages 22, 29).

[67] JH Kim and VH Vu. "Sandwiching random graphs: Universality between random graph models." In: *Advances in Mathematics* 188.2 (2004), pp. 444–469 (pages 11, 73).

[68] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson Education, 2006 (page 61).

[69] Christopher James Kuhlman, VS Anil Kumar, Madhav V Marathe, Samarth Swarup, Gaurav Tuli, SS Ravi, and Daniel J Rosenkrantz. "Inhibiting the diffusion of contagions in bi-threshold systems: Analytical and experimental results." In: *Proceedings of the AAAI Fall Symposium: Complex Adaptive Systems*. 2011 (page 1).

[70] Vito Latora and Massimo Marchiori. "Vulnerability and protection of infrastructure networks." In: *Physical Review E* 71.1 (2005), p. 015103 (page 72).

[71] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. [Online; accessed 28-Jun-2014] (pages 29, 30, 94, 95).

[72] Bil Lewis and Daniel J Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., 1998 (page 2).

[73] Fredrik Liljeros, Christofer R Edling, Luis A Nunes Amaral, H Eugene Stanley, and Yvonne Aberg. "The web of human sexual contacts." In: *Nature* 411 (2001), pp. 907–908 (page 72).

[74] Jun S Liu and Rong Chen. "Sequential Monte Carlo methods for dynamic systems." In: *Journal of the American Statistical Association* 93.443 (1998), pp. 1032–1044 (pages 73, 77).

[75] László Lovász and Michael D Plummer. *Matching Theory*. Vol. 367. American Mathematical Society, 2009 (pages 4, 72, 73).

[76] Nadimpalli VR Mahadev and Uri N Peled. *Threshold Graphs and Related Topics*. Vol. 56. Elsevier, 1995 (pages 76, 89).

[77] Priya Mahadevan, Dmitri Krioukov, Marina Fomenkov, Xenofontas Dimitropoulos, Amin Vahdat, et al. "The Internet AS-level topology: Three data sources and one definitive metric." In: *ACM SIGCOMM Computer Communication Review* 36.1 (2006), pp. 17–26 (page 81).

[78] Peter Mahlmann and Christian Schindelhauer. "Peer-to-peer networks based on random transformations of connected regular undirected graphs." In: *Proceedings of the 17th Annual Symposium on Parallelism in Algorithms and Architectures*. ACM. 2005, pp. 155–164 (page 48).

[79] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A system for large-scale graph processing." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. ACM. 2010, pp. 135–146 (pages 1, 2).

[80] Joel C Miller. "Percolation and epidemics in random clustered networks." In: *Physical Review E* 80.2 (2009), p. 020901 (page 54).

[81] Ron Milo, Nadav Kashtan, Shalev Itzkovitz, Mark EJ Newman, and Uri Alon. "On the uniform generation of random graphs with prescribed degree sequences." In: *arXiv preprint cond-mat/0312028* (2003) (page 54).

[82] Martina Morris and Mirjam Kretzschmar. "Concurrent partnerships and transmission dynamics in networks." In: *Social Networks* 17.3 (1995), pp. 299–318 (page 54).

[83] Aaftab Munshi. "The OpenCL specification." In: *Proceedings of the 2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–314 (page 2).

[84] Mark Newman. "Assortative mixing in networks." In: *Physical Review Letters* 89.20 (2002), p. 208701 (pages 4, 9, 41, 53).

[85] Mark Newman. "Mixing patterns in networks." In: *Physical Review E* 67.2 (2003), p. 026126 (pages 4, 9, 41, 53, 54).

[86] Mark EJ Newman. "The structure and function of complex networks." In: *SIAM Review* 45.2 (2003), pp. 167–256 (pages 5, 11, 73).

[87] Mark EJ Newman. "The structure of scientific collaboration networks." In: *Proceedings of the National Academy of Sciences* 98.2 (2001), pp. 404–409 (page 82).

[88] Mark EJ Newman and Juyong Park. "Why social networks are different from other types of networks." In: *Physical Review E* 68.3 (2003), p. 036122 (pages 4, 9, 53).

[89] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007 (page 2).

[90] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, Inc., 1996 (page 2).

[91] CUDA Nvidia. *Compute unified device architecture programming guide*. 2007 (page 2).

[92] CUDA Nvidia. *Programming guide*. 2008 (page 2).

[93] Tore Opsahl, Filip Agneessens, and John Skvoretz. "Node centrality in weighted networks: Generalizing degree and shortest paths." In: *Social Networks* 32.3 (2010), pp. 245–251 (pages 81, 82).

[94] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. "GPU computing." In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899 (page 2).

[95] Dov A Pechenick, Jason H Moore, and Joshua L Payne. "The influence of assortativity on the robustness and evolvability of gene regulatory networks upon gene birth." In: *Journal of Theoretical Biology* 330 (2013), pp. 26–36 (page 54).

[96] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005 (page 2).

[97] Jaideep Ray, Ali Pinar, and Comandur Seshadhri. "Are we there yet? When to stop a Markov chain while generating random graphs." In: *Proceedings of the 9th Workshop on Algorithms and Models for the Web Graph (WAW)*. Springer. 2012, pp. 153–164 (pages 3, 11, 12).

[98] Timothy C Reluga, Jan Medlock, and Alan S Perelson. "Backward bifurcations and multiple equilibria in epidemic models with structured immunity." In: *Journal of Theoretical Biology* 252.1 (2008), pp. 155–165 (page 1).

[99] Zhihai Rong, Xiang Li, and Xiaofan Wang. "Roles of mixing patterns in cooperation on a scale-free networked game." In: *Physical Review E* 76.2 (2007), p. 027101 (page 54).

[100] Ryan A. Rossi and Nesreen K. Ahmed. "An interactive data repository with visual analytics." In: *ACM SIGKDD Explorations Newsletter* 17.2 (2016), pp. 37–41 (pages 81, 82).

[101] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010 (page 2).

[102] Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. "Navigating the maze of graph analytics frameworks using massive graph datasets." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM. 2014, pp. 979–990 (page 2).

[103] Georgos Siganos, Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. "Power laws and the AS-level internet topology." In: *IEEE/ACM Transactions on Networking (TON)* 11.4 (2003), pp. 514–524 (page 72).

[104] Rohit Singh, Jinbo Xu, and Bonnie Berger. "Global alignment of multiple protein interaction networks with application to functional orthology detection." In: *Proceedings of the National Academy of Sciences* 105.35 (2008), pp. 12763–12768 (pages 81, 82).

[105] Marc Snir. *MPI–The Complete Reference: The MPI Core*. Vol. 1. MIT press, 1998 (pages 1, 2).

[106] Isabelle Stanton and Ali Pinar. "Constructing and sampling graphs with a prescribed joint degree distribution." In: *Journal of Experimental Algorithmics (JEA)* 17 (2012), pp. 3–5 (pages 3, 11).

[107] Alexandre O Stauffer and Valmir C Barbosa. "A study of the edge-switching Markov-chain method for the generation of random graphs." In: *arXiv preprint cs/0512105* (2005) (pages 3, 11).

[108] Angelika Steger and Nicholas C Wormald. "Generating random regular graphs quickly." In: *Combinatorics, Probability and Computing* 8.04 (1999), pp. 377–396 (pages 11, 73).

[109] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." In: *Computing in Science & Engineering* 12.1-3 (2010), pp. 66–73 (page 2).

[110] Lionel Tabourier, Camille Roth, and Jean-Philippe Cointet. "Generating constrained random graphs using multiple edge switches." In: *Journal of Experimental Algorithmics (JEA)* 16 (2011), pp. 1–7 (pages 3, 11).

[111] Duncan J Watts and Steven H Strogatz. "Collective dynamics of 'small-world' networks." In: *Nature* 393.6684 (1998), pp. 440–442 (pages 30, 56, 72, 94, 95).

[112] Douglas Brent West. *Introduction to Graph Theory*. Vol. 2. Prentice Hall, 2001 (page 6).

[113] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012 (page 1).

[114] Nicholas C Wormald. "Models of random regular graphs." In: *London Mathematical Society Lecture Note Series* (1999), pp. 239–298 (pages 11, 73).

[115] R Xulvi-Brunet and IM Sokolov. "Reshuffling scale-free networks: From random to assortative." In: *Physical Review E* 70.6 (2004), p. 066102 (pages 41, 42, 54, 71).

[116] Jaewon Yang and Jure Leskovec. "Defining and evaluating network communities based on ground-truth." In: *Knowledge and Information Systems* 42.1 (2015), pp. 181–213 (page 72).

[117] A Yasin Yazıcıoğlu, Magnus Egerstedt, and Jeff S Shamma. "Decentralized formation of random regular graphs for robust multi-agent networks." In: *Proceedings of the 53rd Annual Conference on Decision and Control*. IEEE. 2014, pp. 595–600 (page 48).

[118] A Yasin Yazıcıoğlu, Magnus Egerstedt, and Jeff S Shamma. "Formation of robust multi-agent networks through self-organizing random regular graphs." In: *IEEE Transactions on Network Science and Engineering* 2.4 (2015), pp. 139–151 (page 48).

[119] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster computing with working sets." In: *HotCloud* 10 (2010), pp. 10–10 (page 1).