A Parallel Algorithm for Generating a Random Graph with a Prescribed Degree Sequence

Hasanuzzaman Bhuiyan*
Department of Computer Science
Network Dynamics and Simulation Science Laboratory (NDSSL)
Biocomplexity Institute of Virginia Tech
Blacksburg, VA, USA
mhb@vt.edu

Maleq Khan
Department of Electrical Engineering and Computer Science
Texas A&M University–Kingsville
Kingsville, TX, USA
maleq.khan@tamuk.edu

Madhav Marathe
Department of Computer Science
Network Dynamics and Simulation Science Laboratory (NDSSL)
Biocomplexity Institute of Virginia Tech
Blacksburg, VA, USA
mmarathe@vt.edu

***Abstract*—Random graphs (or networks) have gained a significant increase of interest due to its popularity in modeling and simulating many complex real-world systems. Degree sequence is one of the most important aspects of these systems. Random graphs with a given degree sequence can capture many characteristics like dependent edges and non-binomial degree distribution that are absent in many classical random graph models such as the Erdős-Rényi graph model. In addition, they have important applications in uniform sampling of random graphs, counting the number of graphs having the same degree sequence, as well as in string theory, random matrix theory, and matching theory. In this paper, we present an OpenMP-based shared-memory parallel algorithm for generating a random graph with a prescribed degree sequence, which achieves a speedup of 20.4 with 32 cores. We also present a comparative study of several structural properties of the random graphs generated by our algorithm with that of the real-world graphs and random graphs generated by other popular methods. One of the steps in our parallel algorithm requires checking the Erdős-Gallai characterization, i.e., whether there exists a graph obeying the given degree sequence, in parallel. This paper presents a non-trivial parallel algorithm for checking the Erdős-Gallai characterization, which achieves a speedup of 23 with 32 cores.**

***Index Terms*—graph theory, random graph generation, degree sequence, Erdős-Gallai characterization, parallel algorithms**

I. Introduction

Random graphs are widely used for modeling many complex real-world systems such as the Internet [1], biological [2], social [3], and infrastructure [4] networks to understand how the systems work through obtaining rigorous mathematical and simulation results. Many random graph models such as the Erdős-Rényi [5], the Preferential Attachment [6], the small-world [7], and the Chung-Lu [8] 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 [9–11]. 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 [12].

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 [13–16]. For example, in an epidemiology study of sexually transmitted diseases [17], 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 [5] graph model. They also have important applications in string theory, random matrix theory, and matching theory [10].

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 paper, 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 [18–21] uses the Markov chain Monte Carlo (MCMC) scheme on a given graph having the degree sequence. An edge swap operation replaces two edges $e_1 = (a, b)$ and $e_2 = (c, d)$, selected uniformly at random from the graph, by new edges $e_3 = (a, d)$ and $e_4 = (c, b)$, 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 the degree of each vertex remains invariant under an edge swap process. Unfortunately, very little theoretical results have been rigorously shown about the mixing time [18, 22] of the edge swap process and they are ill-controlled. Moreover, most of the results are heuristic-based.

On the other hand, among the swap-free stub-matching methods, the *configuration* or *pairing* method [23] is very

*The author is now with Microsoft Inc.

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

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 [24–26] of the configuration models have been studied to avoid parallel edges for the regular graphs. By using the Havel-Hakimi method [27], a deterministic graph can be generated following a given degree sequence. Bayati et al. [15] 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. [16] 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. [14] also proposed a sequential importance sampling [28] algorithm to generate random graphs with an exact 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. [29], which runs in $O(\log n)$ time using $O(n + m)$ CRCW PRAM [30] 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 $O(\log^4 n)$ using $O(n^{10})$ EREW PRAM processors, has been presented in [31], 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 [32]. 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 paper, 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. [14] 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.

Our Contributions. In this paper, we present an efficient shared-memory parallel algorithm for generating random graphs with exact given 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 various structural properties of the random graphs generated by the parallel algorithm with that of the real-world graphs. 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 [33] in parallel. We present here a novel parallel algorithm for checking the Erdős-Gallai characterization, which achieves a speedup of 23 using 32 cores.

Organization. The rest of the paper is organized as follows. Section II describes the preliminaries and notations used in the paper. Our main parallel algorithm for generating random graphs along with the experimental results are presented in Section III. We present a parallel algorithm for checking the Erdős-Gallai characterization of a given degree sequence accompanied by the performance evaluation of the algorithm in Section IV. Finally, we conclude in Section V.

II. Preliminaries

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

Notations. We use $G = (V, E)$ to denote a simple graph, where V is the set of vertices and E is the set of edges. 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 graph* is an undirected graph with no self-loops or parallel edges. We are given a *degree sequence* $D = (d_1, d_2, \ldots, d_n)$. There are a total of $n = |V|$ 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 D and distinct $u, v \in \{1, 2, \ldots, n\}$, we

TABLE I: Notations used frequently in the paper.

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

define $\ominus^{D}_{u,v}$ to be the degree sequence obtained from 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 $\ominus^{D}_{u,v}$, then

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

If there is a simple graph G having the degree sequence D, then there are $m = \frac{1}{2}\sum_i d_i$ edges in G, where $2m = \sum_i d_i$. The terms graph and network are used interchangeably throughout the paper. 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 P be the number of processing cores, and P_k the core with rank k, where $0 \leq k < P$. A summary of the frequently used notations (some of them are introduced later for convenience) is provided in Table I.

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 D of non-negative integers is called *graphical* 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 D. Note that there can be several graphs having the same degree sequence. Eight equivalent necessary and sufficient conditions for testing the graphicality of a degree sequence are listed in [34]. Among them, the Erdős-Gallai characterization [33] is the most famous and frequently used criterion. Another popular recursive test for checking a graphical sequence is the Havel-Hakimi method [27].

Erdős-Gallai Characterization [33]. Assuming a given degree sequence D is sorted in non-increasing order, i.e., $d_1 \geq d_2 \geq \ldots \geq d_n$, the sequence D is graphical if and only if $\sum_{i=1}^{n} d_i$ is even and

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

For example, $D_1 = (3, 3, 2, 2, 2)$ is a graphical sequence and there is a realization of D_1 as it satisfies the Erdős-Gallai

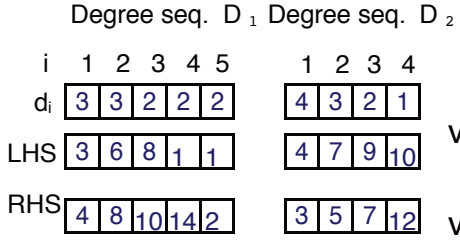

Fig. 1: Graphicality check for the degree sequences $D_1 = (3, 3, 2, 2, 2)$ and $D_2 = (4, 3, 2, 1)$ using the Erdős-Gallai characterization, where LHS and RHS denote the left hand side and right hand side values of Eq. (2), respectively.

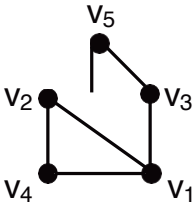

Fig. 2: A simple graph realizing the degree sequence $D_1 = (3, 3, 2, 2, 2)$.

sequence and there is no simple graph realizing D_2, as shown in Figs. 1 and 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.

III. Generating Random Graphs with Prescribed Degree Sequences

We briefly discuss the sequential algorithm in Section III-A. Then we present our parallel algorithm in Section III-B and the experimental results in Section III-C.

A. Sequential Algorithm

Blitzstein et al. [14] presented a sequential importance sampling [28] 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 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 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 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 $D = (3, 3, 2, 2, 2)$, the algorithm starts by assigning edges incident on vertex v_3. It computes the candidate list $C = \{v_1, v_2, v_4, v_5\}$. Say it chooses the vertex v_5 from C and assigns the edge (v_3, v_5). Then the new degree sequence is $D = (3, 3, 1, 2, 1)$, and the new candidate list for assigning the remaining edge incident on vertex v_3 is $C = \{v_1, v_2\}$. Say the algorithm selects v_1 from C and assigns the edge (v_3, v_1). Now the new degree sequence is $D = (2, 3, 0, 2, 1)$, and the algorithm will proceed to assign edges incident on vertex v_5 and so on. One possible characterization, whereas $D_2 = (4, 3, 2, 1)$ is not a graphical

sequence of degree sequences is

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

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

Fig. 3: A sequential algorithm [14] for generating a random graph with a given degree sequence.

with the corresponding edge set

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

The corresponding graph is shown in Fig. 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 Fig. 3. Since a total of m edges are generated for the graph G and computing the candidate list (Line 5) 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 [14]) 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 [14]). For additional details about the importance sampling and estimating the number of graphs for a given degree sequence, see Sections 8 and 9 in [14]; and we omit the details in this paper due to space constraints.

B. 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, the vertices are considered (to assign their incident edges) in the same order in the parallel algorithm, i.e., in ascending order of their degrees. Hence, we emphasize parallelizing the computation of the candidate list C, i.e., Line 5 of the sequential algorithm in Fig. 3. For computing the candidate list to assign edges incident on a vertex u, we need to consider all other vertices v with non-zero degrees d_v as potential candidates; and we parallelize this step. While considering a particular vertex v as a candidate, we need to check whether $\ominus^D_{u,v}$ is a graphical sequence using the Erdős-Gallai characterization. If $\ominus^D_{u,v}$ is graphical, then v is added to the candidate list C. The time complexity of the best known sequential algorithm for testing the Erdős-Gallai characterization is $O(n)$ [14, 35]. 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 IV, we present an efficient parallel algorithm for checking the Erdős-Gallai characterization that runs in $O(\frac{n}{P} + \log P)$ time.

```
1: E ← ∅        ▷ initially empty set of edges
   ▷ Assign the edges until the degree of
   each vertex reduces to 0
2: while D ≠ 0 do
3:     Select the least u such that d_u is a minimal positive
       degree in D
4:     C ← ∅                    ▷ candidate list
       ▷ Assign all d_u edges incident on u
5:     while d_u ≠ 0 do
6:         if C = ∅ then
7:             F ← {v ≠ u : (u, v) ∉ E ∧ d_v > 0}
8:         else
9:             F ← C
10:            C ← ∅
           ▷ Compute the candidate list
11:        for each v ∈ F in parallel do
12:            flag ← PARALLEL-ERDŐS-GALLAI (⊖^D_{u,v})
13:            if flag = TRUE then
14:                C ← C ∪ {v}      ▷ v is a candidate
15:        if d_u = |C| then
16:            for each v ∈ C in parallel do
17:                E ← E ∪ {(u, v)}
18:                D ← ⊖^D_{u,v}
19:            break
           ▷ Assign an edge (u, v) from C
20:        v ← a random candidate in C where probability of
           selecting v is proportional to d_v
21:        E ← E ∪ {(u, v)}
22:        D ← ⊖^D_{u,v}
23:        C ← C − {v}
24: Output E        ▷ final set of edges
```

Fig. 4: A parallel algorithm for generating a random graph with a prescribed degree sequence.

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 $|C|$ of the candidate list, then new edges are assigned between u and all candidate vertices v in the candidate list C in parallel. Otherwise, like the sequential algorithm, a candidate vertex v is chosen randomly from C, a new edge (u, v) is assigned, the degree sequence D is updated by reducing the degrees of each of u and v by 1, and the process is repeated until d_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 D and so on. We present the pseudocode of our parallel algorithm for generating random graphs in Fig. 4.

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 3), assigns d_u edges incident on u (Lines 5-23), and then proceeds with the