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

Maksudul Alam*†, Maleq Khan*, Anil Vullikanti*†, and Madhav Marathe*†
*Network Dynamics and Simulation Science Laboratory, Biocomplexity Institute of Virginia Tech
†Department of Computer Science, Virginia Tech, Blacksburg, Virginia 24061 USA
Email: {maksud, maleq, akumar, mmarathe}@vbi.vt.edu

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

***Index Terms*—network theory, random graphs, parallel programming, distributed computing**

I. Introduction

The advancements of hardware technologies, software, and algorithms have enabled the detailed study of complex systems. These systems, such as the Internet [1, 2], biological networks [3], and social networks [4–6] are sometimes modeled by random graphs for the purpose of studying their behavior. Degree distribution is one of the most prominent features of these networks. Some well-understood graph models have been developed to capture the diversity of the degree distributions such as Erdős–Rényi [7], stochastic block models [8], small-world [9], Barabási–Albert [10, 11], exponential random graph [12, 13], recursive matrix [14], stochastic Kronecker graph [15, 16], and HOT [17] models.

Each of these models has been developed considering some aspect of the networks. The Erdős–Rényi model [7] was the first attempt to perform a systematic study of networks. The stochastic block model [8] has been studied for a long time to study the community structures found in many real–world networks. The small-world model [9] was proposed to capture the small-world property found in many real–world systems. The Barabási–Albert model [10] famously captured the preferential attachment and power–law degree distribution properties. However, these models generate graphs with a specific type of degree distributions. The Chung-Lu model [18, 19] can generate a random graph with a given sequence of expected degrees and is capable of generating networks from almost any real-world degree distribution. Another model called the block two–level Erdős–Rényi (BTER) [20, 21] had been developed recently to study community structure, which can capture both degree distribution and clustering coefficients. A generalization of BTER was proposed in [22]. A joint degree distribution model [23] was also presented to study the assortativity of real–world networks.

As the complex systems are growing larger and larger, it requires the generation of massive random networks efficiently. Analyzing a very large complex system using a smaller model may not produce accurate results. As the interactions in a larger network lead to complex collective behavior [24], a smaller network may not exhibit the same behavior, even if both networks are generated using the same model. In [24], by experimental analysis, it was shown that the structure of larger networks is fundamentally different from small networks, and many patterns emerge only in massive datasets.

Demand for large random graphs necessitates efficient algorithms, in terms of both time and space requirements, to generate such graphs. However, even efficient sequential algorithms for generating such graphs were nonexistent until recently. Although recently some efficient sequential algorithms have been developed [14, 15, 25, 26], these algorithms can generate graphs with only millions of vertices in a reasonable time. Generating graphs with billions of vertices can take an undesirably long time. Further, a large memory requirement may even prohibit the generations of such large graphs. As a result, distributed-memory parallel algorithms are now desirable in dealing with these problems.

There have been some efforts to deal with massive networks. In one such effort, the Graph500 group [27] choose the SKG model in the supercomputer benchmark due to its simpler parallel implementation. The CL model is very similar to the

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

SKG model [28], which could replace the SKG model due to similar properties and ability to generate a wider range of degree distributions. In this paper, our main focus is on the CL model. The previous best known sequential algorithm for the CL model is given in [26] which takes $O(m+n)$ expected time and $O(n)$ space, where m and n are the number of edges and nodes in the graph. Based on this sequential algorithm, a distributed-memory parallel algorithm is presented in [29], which is the only known parallel algorithm for the CL model. This parallel algorithm takes $O\left(\frac{m+n}{P}+P\right)$ time with high probability (w.h.p.) and $O(n)$ space using P processors.

In this paper, we present a novel method (called the *DG method*), based on grouping the vertices by their degrees, that leads to space and time efficient algorithms for several random graph models, including the CL model, with rigorous guarantees. Our main contributions are summarized below.

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

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

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

The rest of the paper is organized as follows. In Section II, we describe the problem and the DG method. In Section III, we present the efficient parallel algorithm along with the optimal load balancing technique. In Section IV and V, we present the algorithms for BTER and SBM models, respectively, applying the DG method. Finally, we conclude in Section VI.

II. Generating random graphs with a desired degree distribution

Many models have been proposed to generate random graphs from a desired degree distribution or sequence. The configuration model [30, 31] is one of the first models that generates a graph with a given degree sequence. This model can generate every possible graph with the given degree sequence with equal probability [31]. However, it can produce graphs with some undesirable properties such as parallel edges and self-loops that are unacceptable for many applications. The Chung–Lu (CL) model [18, 19] is another widely used model that generates random graphs from a given sequence of expected degrees by avoiding these undesirable properties.

A. The Chung–Lu Model

Assume that we are given n vertices labeled as $1, 2, \ldots, n$ and a sequence of expected degrees $W = \langle w_1, w_2, \ldots, w_n \rangle$ such that a vertex u has an expected degree of w_u. In the Chung–Lu (CL) model, any pair of vertices u and v are connected by an edge with the probability $p_{u,v} = \frac{w_u w_v}{S}$, where $S = \sum_u w_u$ (assuming $\max_u w_u^2 \leq S$, we have $p_{u,v} \leq 1$ for all u and v) [18, 19]. For simple graphs without self-loops, i.e., $u \neq v$, the expected degree of a vertex u is $\sum_v \frac{w_u w_v}{S} = w_u - \frac{w_u^2}{S}$, which converges to w_u for large graphs.

A naïve algorithm for the CL model is: individually consider each of the $\frac{n(n-1)}{2}$ possible pairs of vertices $\{u, v\}$ and create edge (u, v) with probability $p_{u,v}$. It requires $O\left(n^2\right)$ time and $O(n)$ space. Pinar et al. [28] presented a sampling-based algorithm (henceforth referred to as the PSK algorithm), where each of the m edge is created by selecting two end points randomly and independently using the degree distribution as the probability distribution. Each end point is sampled in constant time using $O(m)$ memory. This algorithm requires $O(m)$ time and $O(m)$ space. Although this algorithm is simple, it does not eliminate self-loops and parallel edges and requires large memory.

Miller and Hagberg presented an $O(m+n)$ time algorithm (referred to as the MH algorithm) that avoids self-loops or parallel edges and requires $O(n)$ memory [26]. They used an *accept–reject sampling* along with the *edge skipping technique* introduced in [25] for the Erdős–Rényi model. Although this is the fastest known sequential algorithm for the CL model, it has some limitations. It requires the sequence W be sorted in a non-increasing order leading to some computational overhead. Additionally, due to the rejection sampling, some potential edges are rejected. The number of such edges has been shown to be $O(m)$, which also incurs significant computational overhead, especially for skewed degree distributions found in most real–world graphs [26]. Moreover, the MH algorithm needs to generate two random numbers per edge (in contrast to one random number per edge in our algorithm).

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

of our algorithm leads to smaller constant associated with the time complexity and makes our algorithm approximately three times faster than the MH algorithm.

TABLE I: A comparison of the algorithms for the CL model

Algorithm	Runtime	Space
Naïve	$O(n^2)$	$O(n)$
PSK [28]	$O(m)$	$O(m)$
MH [26]	$O(m+n)$	$O(n)$
DG (this paper)	$O(m)$	$O(\Lambda)$

B. DG: A New Time and Memory Efficient Algorithm

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

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

Creating Intra Edges. For any $u, v \in V_i$, the edge (u, v) is created with probability $p_{u,v} = \frac{w_u w_v}{S} = \frac{d_i^2}{S}$, since $w_u = w_v = d_i$. Notice that for all pairs of $u, v \in V_i$, the probabilities $p_{u,v}$ are equal. Thus generating the intra edges on V_i is equivalent to generating an Erdős-Rényi (ER) random graph $G(n, p)$ with $n = n_i = |V_i|$ and $p = \frac{d_i^2}{S}$. The ER model $G(n, p)$ generates a random graph with n vertices where each of $\frac{n(n-1)}{2}$ possible potential edges is selected and added to the generated graph with probability p. We generate the intra edges on V_i for all i by generating ER random graphs $G(n_i, \frac{d_i^2}{S})$. A Naïve algorithm to generate random graph $G(n, p)$ is: for each of the $\frac{n(n-1)}{2}$ potential edges, toss a biased coin and select the edge with probability p. This Naïve algorithm takes $O(n^2)$ time. An efficient algorithm for the ER model is given in [25], which runs in $O(m)$ time. This algorithm uses a technique called edge skipping technique. From a sequence of all potential edges, it selects a subset of the edges using the skipping technique. We also use this edge skipping technique to generate the inter edges. Below we briefly describe the edge skipping technique (see [25] for details).

Edge Skipping Technique. Consider a sequence of potential edges as shown in Fig. 1. The goal is to select a subset of the edges such that each potential edge is selected with probability p. In Fig. 1, each circle represents a potential edge, and a black circle represents a selected edge. Notice that before selecting an edge, a sequence of zero or more potential edges is discarded (white circles), e.g., in Fig. 1, edge e_1 is discarded and e_2 is selected. Then e_3 and e_4 are discarded, and e_5 is selected, and so on. Let ℓ be a random variable, which denotes the number of consecutive edges discarded before selecting the next edge. Then the probability that ℓ edges are discarded is

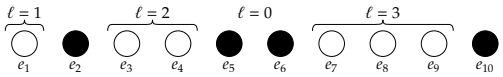


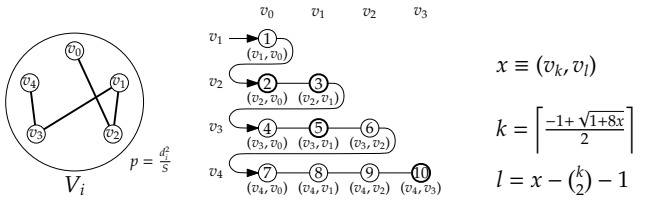
Fig. 1: Selecting edges from a sequence of potential edges. The black circles represents the selected edges.

$$\Pr\{\ell \text{ edges are discarded}\} = (1-p)^{\ell} p. \tag{1}$$

Notice that ℓ is a geometric random variable. A geometric random number can be generated in constant time from a uniform random number $r \in (0, 1]$ using the inverse transform method [25], which gives

$$\ell = \left\lfloor \frac{\log r}{\log(1-p)} \right\rfloor \tag{2}$$

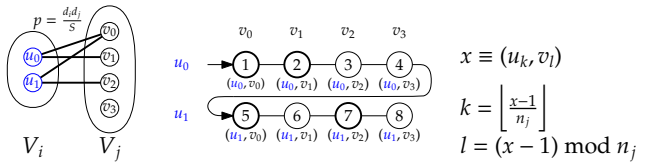
Now to generate the intra edges, i.e., ER random graph $G(n_i, \frac{d_i^2}{S})$, we apply the edge skipping technique on a sequence of the potential edges. To save memory space, we do not create any explicit sequence of the edges. Instead, the edges are represented by a set of consecutive integers $1, 2, \ldots, M$, where $M = \binom{|V_i|}{2} = \binom{n_i}{2}$, following a lexicographic order of the edges as shown in Fig. 2(a) and 2(b). Now we select a subset of the integers from $1, 2, \ldots, M$ by applying the skipping technique with the probability $p = \frac{d_i^2}{S}$ as follows. Let x be the last selected edge (initially $x = 0$). Then the skip length ℓ is computed using the Equation 2. The next selected edge is given by $x \leftarrow x + \ell + 1$. The selected edge number x is converted into an edge using the equations shown in Fig. 2(c). This process is repeated until $x \ge M$.



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

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

Creating Inter Edges. For any $u \in V_i, v \in V_j$, the edge (u, v) is created with probability $p_{u,v} = \frac{d_i d_j}{S}$. Note that for all pairs of $u \in V_i, v \in V_j$, the probabilities $p_{u,v}$ are equal. Therefore, generating the inter edges between V_i and V_j is equivalent to generating a random bipartite graph [32] with n_i and n_j vertices and $p = \frac{d_i d_j}{S}$ edge probability (see Fig. 3(a)).



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

Fig. 3: Inter edges between V_i and V_j ($n_i = 4$ and $n_j = 2$).

The edge skipping technique can also be applied to generate the inter edges using the random bipartite model (Fig. 3(a)). In this case, the potential edges are represented by consecutive integers $1, 2, \ldots, M$, where $M = |V_i||V_j| = n_i n_j$ (Fig. 3(b)). Next, the edge skipping technique is applied on this sequence with probability $p = \frac{d_i d_j}{S}$. The selected numbers x are converted to the edges using the equations shown in Fig. 3(c).

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

Implementation. The pseudocode of our algorithm is presented in Algorithms 1 and 2. The procedure EDGE-SKIPPING creates edges using the edge skipping technique. It takes two group indices i, j, probability p, and the $start$ and end of a sequence of potential edges as input. Using a random number $r \in (0, 1]$, the skip length ℓ is computed in line 5. The next selected edge x is computed in line 5 and converted to edge (u, v) in line 8 (intra edges) and line 10 (inter edges). The global labels of the endpoints u and v are denoted by $\lambda_i + u$ and $\lambda_j + v$, respectively.

Algorithm 1 Generating edges using edge skipping

1: **procedure** EDGE-SKIPPING($i, j, p, start, end$)
2: $x \leftarrow start - 1$
3: **while** $x < end$ **do**
4: $r \leftarrow$ a uniform random number in $[0, 1)$
5: $\ell \leftarrow \left\lfloor \frac{\log r}{\log(1-p)} \right\rfloor$; $x \leftarrow x + \ell + 1$
6: **if** $x \leq end$ **then**
7: **if** $i = j$ **then**
8: $u \leftarrow \left\lceil \frac{-1+\sqrt{1+8x}}{2} \right\rceil$; $v \leftarrow x - \binom{u}{2} - 1$
9: **else**
10: $u \leftarrow \left\lfloor \frac{x-1}{n_j} \right\rfloor$; $v \leftarrow (x-1) \bmod n_j$
11: Output edge $(\lambda_i + u, \lambda_j + v)$

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

Algorithm 2 The DG algorithm for the CL model

1: **procedure** DG-CL($(\mathbb{D}, \{n_i\}_{i \in \mathbb{D}})$)
2: $\lambda_1 \leftarrow 1$
3: **for** $i = 2$ **to** Λ **do** ▷ Starting Labels
4: $\lambda_i \leftarrow \lambda_{i-1} + n_i$
5: $S \leftarrow \sum_{i=1}^{\Lambda} (n_i \times d_i)$
6: **for** $i = 1$ **to** Λ **do**
7: **for** $j = 1$ **to** j **do**
8: **if** $i = j$ **then** ▷ Intra edges for V_i
9: EDGE-SKIPPING($i, i, \frac{d_i^2}{S}, 1, \binom{n_i}{2}$)
10: **else** ▷ Inter edges between V_i and V_j
11: EDGE-SKIPPING($i, j, \frac{d_i d_j}{S}, 1, n_i n_j$)

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

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

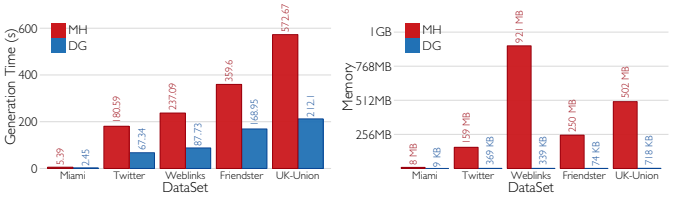
When $\Lambda^2 < O(m)$, our algorithm is asymptotically as good as the MH algorithm. In fact experimental results show that our algorithm about 3∼4 times faster than the MH algorithm. Note that the CL model is only applicable when $w_{\max}^2 < S$, where $w_{\max}$ is the maximum expected degree. For integral expected degrees, we have $\Lambda < w_{\max}$, i.e., $\Lambda^2 < S = O(m)$. Therefore, whenever the CL model is applicable, our algorithm has a runtime of $O(m)$. Clearly, we will use the CL model only when it is applicable. In fact, in most of the real–world graphs Λ^2 is significantly smaller than m as shown in Table II. Even for power–law networks that have very skewed degree distributions and few vertices with very high degrees, the maximum degree is $O(\sqrt[\gamma]{n})$ where γ is the power–law exponent [33]. Typical values of γ is between 2 and 3.

TABLE II: Number of distinct degrees in real–world graphs

Network	Type	n	m	Λ	$\frac{\Lambda^2}{m}$
Miami [34]	Contact	2M	51M	398	0.003
Weblinks	Real-world	276M	1B	14K	0.190
Twitter [5]	Real-world	41M	1B	20K	0.207
Friendstar [6]	Real-world	65M	2B	3K	0.005
UK-Union [35]	Real-world	131M	4B	30K	0.201
Erdős-Rényi (ER)	Synthetic	1M	200M	117	0.00007
Power-Law (PL)	Synthetic	1B	249B	10K	0.0004

Experimental Evaluation. Now we experimentally evaluate

the performance of our algorithm against the MH algorithm [26], which is the best known sequential algorithm, using both real-world and synthetic networks. We extracted the degree sequences of these networks, and then generated new graphs from these degree sequences. Fig. 4 demonstrates the runtime and memory required by the algorithms. We observe that our DG algorithm is approximately 3 times faster than the MH algorithm as we discussed before. A huge improvement made by our DG algorithm is on the memory requirement, by a factor of 440–3474 for the networks shown in Fig. 4. Thus, the DG algorithm is more efficient in both time and space requirements. Moreover, our DG algorithm leads to a better parallel algorithm, which is presented in the next section.



(a) Runtime (b) Consumed Memory

Fig. 4: Performance of the sequential algorithms

We also compare the performance of our algorithm against the Graph500 generator (Version 2.1.4), one of the most used random graph generators [27] for benchmarking HPC systems. Generating a graph with $1B$ edges requires 650.46 seconds using the sequential Graph500 generator. In contrast, our algorithm takes only 67.34 seconds. Further, the Graph500 generator requires $\Theta(m)$ space whereas our algorithm requires $\Theta(\Lambda)$ space, where $\Lambda \ll m$. Moreover, to fit the degree distribution of an existing real-world network, the probability matrix of the Graph500 generator has to be determined using a maximum likelihood based fitting algorithm (KronFit), which can take a significant amount of time, and often the fit is not perfect [16, 28]. Fitting the degree distribution of a real–world network (75K vertices, 508K edges) requires approximately 45 minutes in KronFit [16]. Our algorithm requires only the degree distribution that can be extracted in a few seconds.

III. Parallelization of the DG Algorithm

In this section, we present the parallelization of the DG algorithm. We assume that the input degree distribution of the graph is available for every processor. Let P be the number of processors. Efficient parallelization of Algorithm 2 requires:

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

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

A. Task Distribution and Load Balancing

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

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

$$E[c_{i,j}] = \alpha + \beta m_{i,j} = \begin{cases} \alpha + \beta \frac{n_i(n_i-1)}{2} \frac{d_i^2}{S}, & i = j \\ \alpha + \beta n_i n_j \frac{d_i d_j}{S}, & i \neq j. \end{cases} \quad (3)$$

Therefore, the total expected computational cost $\mathcal{C}$ is given by

$$\mathcal{C} = \sum_{1 \le j \le i \le \Lambda} E[c_{i,j}] = \sum_{1 \le j \le i \le \Lambda} (\alpha + \beta m_{i,j}) = \alpha\tau + \beta m \quad (4)$$

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

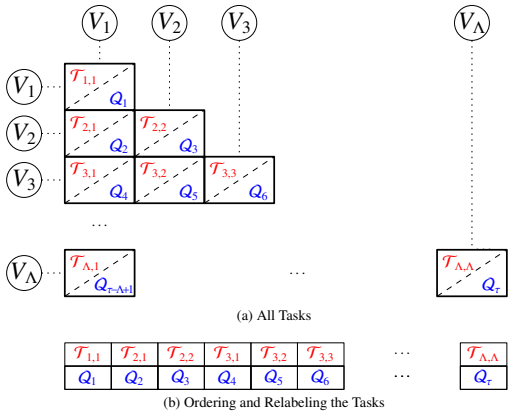


Fig. 5: Relabeling the tasks. The red and blue texts represent the original and new labels of the tasks respectively.

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

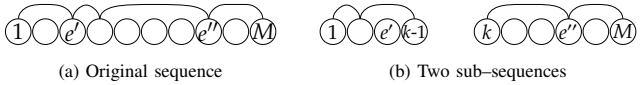
i.e., $\Pi = \{\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \ldots, \mathcal{Q}_\tau\}$. The relabeled task $\mathcal{Q}_x$ can be converted to the original label $\mathcal{T}_{i,j}$ using the functions:

$$i = \left\lceil \frac{-1+\sqrt{1+8x}}{2} \right\rceil \quad \text{and} \quad j = x - \frac{i(i-1)}{2}. \tag{5}$$

Task relabeling is depicted visually in Fig. 5. Let c_x be the cost of the task $\mathcal{Q}_x$, i.e., $c_x = c_{i,j}$ for the original task $\mathcal{T}_{i,j}$.

Task Distribution. To generate the edges in parallel, first, the set of tasks Π is divided into P disjoint subsets $\Pi_0, \Pi_1, \ldots, \Pi_{P-1}$; i.e., $\Pi_k \subset \Pi$, such that for any $k \neq l$, $\Pi_k \cap \Pi_l = \emptyset$ and $\bigcup_k \Pi_k = \Pi$. Each processor P_k is assigned the subset Π_k and executes the tasks $\{\mathcal{Q}_x \in \Pi_k\}$. The computational cost of a processor P_k is given by $c(P_k) = \sum_{\mathcal{Q}_x \in \Pi_k} c_x$. We need to find the subsets Π_k such that each processor has almost equal cost, i.e., $c(P_k) \approx \hat{\mathcal{C}}$. Finding such subsets is a well-known problem called *chains-on-chains partitioning* (CCP) problem [36–38]. For better speedup the CCP problem has to be computed in parallel. In [29], the authors presented an efficient $O(\frac{\tau}{P} + P)$ time distributed memory parallel algorithm, called the *uniform cost partitioning* (UCP) for the CCP problem. Their algorithm uses cumulative cost $\mathcal{C}_x = \sum_{i=1}^{x} c_x$ for each task $\mathcal{Q}_x$ for distributing tasks. Let any subset Π_k starts with the task $\mathcal{Q}_{q_k}$ and ends with the task $\mathcal{Q}_{q_{k+1}-1}$ where q_k is called the *lower boundary* of Π_k. The lower boundary q_k satisfies the following condition: $\mathcal{C}_{q_k-1} < k\hat{\mathcal{C}} \leq \mathcal{C}_{q_k}$ for $0 < k \leq P-1$. Then $q_k = \arg\min_x(\mathcal{C}_x \geq i\hat{\mathcal{C}})$, i.e., a task $\mathcal{Q}_x$ is executed by the processor P_k where $k = \left\lfloor \frac{\mathcal{C}_x}{\hat{\mathcal{C}}} \right\rfloor$. However, a task in the UCP algorithm is non–divisible, i.e., the entire task is assigned to a processor. If some boundary tasks are very large, it can lead to imbalanced loads. If we can break those tasks into arbitrary smaller subtasks, we can achieve a fine-grained granularity on load balancing. In fact, we can show a task $\mathcal{Q}_x$ can be divided into arbitrary smaller subtasks. As a result, we present an extension of the UCP algorithm, called UCP–DIV that achieves fine-grained load balancing by dividing the tasks.

Dividing Tasks. Using the edge-skipping technique, a task repeatedly computes the skip lengths with a probability p (using the geometric distribution) and generates edges from a sequence of potential edges (see Fig. 6(a)). We can also divide the sequence into two arbitrary disjoint sub-sequences and apply the edge skipping technique on those sub-sequences individually (see Fig. 6(b)), with the same result and effect. Both of these two processes are stochastically equivalent due to the memoryless property of the geometric random variable.



(a) Original sequence (b) Two sub–sequences

Fig. 6: Dividing a task into multiple sub–tasks

Consider two edges e' and e'' in Fig. 6(a). In the original sequence, both e' and e'' are selected with probability p regardless of their distance from the beginning edge of the sequence. In other words, regardless where the sequence begins, every edge is selected with probability p. That is, we can arbitrarily break the sequence into two subsequences at any edge k and use that as the beginning of the second subsequence (Fig. 6(b)). Any edge $e'' \geq k$ will still be selected with probability p if the edge skipping technique is applied on that subsequence. In fact, any sequence can be divided into any number of subsequences.

Uniform Cost Partitioning with Task Division. Now, we present the UCP–DIV algorithm. Similar to the original UCP algorithm, we also find the lower boundary x of a partition Π_k such that $\mathcal{C}_{x-1} < k\hat{\mathcal{C}} \leq \mathcal{C}_x$. Instead of assigning the entire boundary task $\mathcal{Q}_x$ to processor P_k where $k = \left\lfloor \frac{\mathcal{C}_x}{\hat{\mathcal{C}}} \right\rfloor$ (as done in the UCP algorithm), we break it into two or more subtasks (see Fig. 7). Let $\mathcal{Q}_{x,s,t}$ be a subtask of the task $\mathcal{Q}_x$ with the subsequence starting from edge number s to t. Note that the subtask $\mathcal{Q}_{x,1,M}$ represents the entire task $\mathcal{Q}_x$ where M is the number of potential edges in $\mathcal{Q}_x$.

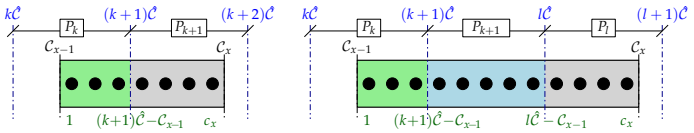


Fig. 7: Dividing the boundary tasks. The blue and green texts represent the cost boundaries among the processors and the subtask partitions within the task respectively.

The first part of the boundary task $\mathcal{Q}_x$ is executed by P_k where $k = \left\lfloor \frac{\mathcal{C}_{x-1}}{\hat{\mathcal{C}}} \right\rfloor$. To make $c(P_k) = \hat{\mathcal{C}}$, we assign $(k+1)\hat{\mathcal{C}} - \mathcal{C}_{x-1}$ more loads to P_k. Therefore, we divide the task $\mathcal{Q}_x$ into a subtask $\mathcal{Q}_{x,1,t}$ such that $t = \left\lfloor \frac{(i+1)\hat{\mathcal{C}} - C_{x-1}}{c_x} M \right\rfloor$ (see Fig. 7).

Now, if the remaining part of the task $\mathcal{C}_x - (k+1)\hat{\mathcal{C}} \leq \hat{\mathcal{C}}$ (see Fig. 7(a)), P_{k+1} executes the last part of the task, i.e., the subtask $\mathcal{Q}_{x,t+1,M}$ is assigned to P_{k+1}. Otherwise, we divide the remaining part of the task again (see Fig. 7(b)). Let P_l be the processor that executes the last part of the task $\mathcal{Q}_x$ where $l = \left\lfloor \frac{\mathcal{C}_x}{\hat{\mathcal{C}}} \right\rfloor$. Each processor from P_{k+1} to P_{l-1} executes $\hat{\mathcal{C}}$ amounts of loads, i.e., P_z is assigned with the subtask $\mathcal{Q}_{x,s_z,t_z}$ where $s_z = \left\lfloor \frac{z\hat{\mathcal{C}} - \mathcal{C}_{x-1}}{c_x} M \right\rfloor + 1$ and $t_z = \left\lfloor \frac{(z+1)\hat{\mathcal{C}} - \mathcal{C}_{x-1}}{c_x} M \right\rfloor$ for $z = (k+1), \ldots (l-1)$ (see Fig. 7(b)). The last part of the task, i.e., the subtask $\mathcal{Q}_{x,s_l,M}$ is assigned to the processor P_l where $s_l = \left\lfloor \frac{\mathcal{C}_x - l\hat{\mathcal{C}}}{c_x} M \right\rfloor + 1$. Thus, the algorithm leads to optimal load balancing.

B. Parallel Implementation

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

Parallel UCP–DIV Algorithm. The pseudocode of the parallel UCP–DIV algorithm is presented in Algorithm 3. The procedure UCP–DIV computes the task boundaries using the procedure FIND–BOUNDARIES. The algorithm assumes that

Algorithm 3 Finding task boundaries using UCP–DIV

1: **procedure** UCP–DIV()
2: $k \leftarrow$ Processor Id
3: ▷ Executed by processor P_k in parallel
4: FIND-BOUNDARIES($\frac{k\tau}{P}+1, \frac{(k+1)\tau}{P}$)
5: $A \leftarrow$ Receive Message $\langle start, q_k, s, t\rangle$
6: $B \leftarrow$ Receive Message $\langle end, q_{k+1}-1, s', t'\rangle$
7: **return** $\langle A, B\rangle$
8: **procedure** FIND-BOUNDARIES(b, e)
9: **if** $b > e$ **then return** ▷ No boundary
10: $x \leftarrow \frac{b+e+1}{2}$;
11: $M \leftarrow$ # of potential edges in $\mathcal{Q}_x$
12: $l \leftarrow \frac{\mathcal{C}_x}{\hat{\mathcal{C}}}$; $k \leftarrow \frac{\mathcal{C}_{x-1}}{\hat{\mathcal{C}}}$
13: **if** $k \neq l$ **then**
14: $t \leftarrow \left\lfloor \frac{(k+1)\hat{\mathcal{C}}-\mathcal{C}_{x-1}}{c_x} M \right\rfloor$
15: Send Message $\langle start, x, 1, t\rangle$ to P_k
16: **for** $z \leftarrow k+1$ **to** $l-1$ **do**
17: $s \leftarrow \left\lfloor \frac{z\hat{\mathcal{C}}-\mathcal{C}_{x-1}}{c_x} M \right\rfloor + 1$; $t \leftarrow \left\lfloor \frac{(z+1)\hat{\mathcal{C}}-\mathcal{C}_{x-1}}{c_x} M \right\rfloor$
18: Send Message $\langle start, x, s, t\rangle$ to P_z
19: Send Message $\langle end, x, s, t\rangle$ to P_z
20: $s \leftarrow \left\lfloor \frac{l\hat{\mathcal{C}}-\mathcal{C}_{x-1}}{c_x} M \right\rfloor + 1$;
21: Send $\langle end, x, s, M\rangle$ to P_l
22: FIND-BOUNDARIES($b, x-1$)
23: FIND-BOUNDARIES($x+1, e$)

the cumulative costs $\mathcal{C}_x$ and average cost per processor $\hat{\mathcal{C}}$ are already computed (discussed in the next section).

Each processor P_k executes the procedure UCP–DIV in-parallel. P_k is responsible for finding the boundary tasks among $\frac{\tau}{P}$ tasks from $\frac{k\tau}{P}+1$ to $\frac{(k+1)\tau}{P}$. A boundary task is a task $\mathcal{Q}_x$ where $\lfloor \frac{C_{x-1}}{\hat{\mathcal{C}}} \rfloor \neq \lfloor \frac{C_x}{\hat{\mathcal{C}}} \rfloor$. The boundary tasks are found using the procedure FIND–BOUNDARIES (line 4). The procedure FIND–BOUNDARIES takes parameters b, e and finds all the boundaries among the tasks from $\mathcal{Q}_b$ to $\mathcal{Q}_e$ using a recursive divide and conquer based algorithm. Each boundary task $\mathcal{Q}_x$ is divided into subtasks and assigned to processors as discussed earlier. Let $\mathcal{Q}_{x,s,t}$ be such a subtask assigned to processor P_z. A message $\langle type, x, s, t\rangle$ representing the subtask $\mathcal{Q}_{x,s,t}$ is sent to P_z (from P_k), where $type, x, s, t$ represents the type of subtask (either *start* or *end*), the task number, the starting edge, and the ending edge of the task, respectively. Note that each processor P_z receives two such messages. The pair of the messages is returned as output (line 7). The runtime of the algorithm is $O\left(\frac{\tau}{P}+P\right)$ in the worst case as shown in Theorem 1.

Theorem 1. *The parallel UCP–DIV algorithm to distribute τ tasks into P processors runs in $O\left(\frac{\tau}{P}+P\right)$ time.*

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

Algorithm 4 Parallel DG Algorithm for the CL Model

1: **procedure** PDG–CL($\mathbb{D}, \{n_k\}_{k\in\mathbb{D}}$)
2: $k \leftarrow$ processor id
3: ▷ Executed by processor P_k in parallel
4: **In-Parallel:** Compute $S = \sum_{l=1}^{\Lambda} n_l d_l$
5: $\lambda_1 = 1$
6: **for** $l \leftarrow 2$ to Λ **do**
7: $\lambda_l \leftarrow \lambda_{l-1} + n_l$
8: **for** $x \leftarrow \frac{k\tau}{P}+1$ **to** $\frac{(k+1)\tau}{P}$ **do**
9: $\mathcal{C}_x \leftarrow \mathcal{C}_{x-1} + c_x$
10: $z_k \leftarrow \mathcal{C}_{\frac{(k+1)\tau}{P}}$
11: **In Parallel:** $\mathcal{C} \leftarrow \sum_{l=0}^{P-1} z_l$
12: $\hat{\mathcal{C}} \leftarrow \frac{\mathcal{C}}{P}$
13: $\langle A, B\rangle \leftarrow$ UCP-DIV()
14: **for** $\mathcal{Q}_{x,s,t} = A$ **to** B **do**
15: $i = \left\lceil \frac{-1+\sqrt{1+8x}}{2} \right\rceil$; $j = x - \frac{i(i-1)}{2}$
16: EDGE–SKIPPING($i, j, \frac{d_i d_j}{S}, s, t$)

Parallel DG Algorithm. The pseudocode of the parallel DG algorithm using the UCP–DIV algorithm is presented in Algorithm 4. The procedure PDG–CL is executed by each processor P_k. It takes a degree distribution as input and generates the edges in parallel. P_k computes the sum (line 4), starting labels of each group (lines 5–7), and the total cost and average computational costs (line 8–12) in parallel. Next, the procedure UCP–DIV (Line 13) is called which return the pair of messages $\langle A, B\rangle$. Recall that A and B represents two subtasks $\mathcal{Q}_{x,s,t}$ and $\mathcal{Q}_{x',s',t'}$ assigned to P_k. P_k executes the subtasks, and all the tasks from $\mathcal{Q}_{x+1}$ to $\mathcal{Q}_{x'}$ using the procedure EDGE–SKIPPING from Algorithm 1 (line 14).

The runtime of parallel Algorithm 4 is $O\left(\frac{m+\Lambda^2}{P} + \Lambda + P\right)$ w.h.p. as shown in Theorem 3. With $\Lambda^2 = O(m)$, the runtime of the algorithm is $O\left(\frac{m}{P} + \Lambda + P\right)$. To prove Theorem 3, we need a bound on computational cost shown in Theorem 2.

Theorem 2. *The computational cost in each processor is $O\left(\frac{m+\tau}{P}\right)$ w.h.p.*

Proof. Let x be the number of potential edges processed in processor P_k, and these are denoted by $g_1, g_2, \ldots, g_x$ (in any arbitrary order). Let X_i be an indicator random variable such that $X_i = 1$ if P_k creates g_i and $X_i = 0$ otherwise. Then the number of edges created by P_k is $X = \sum_{i=1}^{x} X_i$. As discussed in Section II-B, generating the edges efficiently by grouping and applying the edge skipping technique is stochastically equivalent to generating each edge (u, v) independently with

probability $p_{u,v} = \frac{w_u w_v}{S}$. Let μ_k be the expected number of edges generated by P_k, i.e., $\mu = E[X] = m_k$. Using the standard Chernoff bound for independent indicator random variables for any $0 < \delta < 1$ with $\delta = \frac{1}{2}$ we have:

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

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

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

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

Theorem 3. *Our parallel algorithm using UCP–DIV for generating random graphs with the CL model runs in* $O\left(\frac{m+\Lambda^2}{P} + \Lambda + P\right)$ *time w.h.p.*

Proof. Computing the sum S in parallel takes $O\left(\frac{\Lambda}{P} + \log P\right)$ time. Computing the starting labels of each group requires $O(\Lambda)$ time. Computing the costs require $O\left(\frac{\tau}{P} + \log P\right)$ time. Using the UCP–DIV algorithm, task distribution takes $O\left(\frac{\tau}{P} + P\right)$ time (Theorem 1). In the UCP–DIV scheme, each partition has $O\left(\frac{m+\tau}{P}\right)$ computation cost w.h.p. (Theorem 2). Thus creating edges using procedure EDGE–SKIPPING requires $O\left(\frac{m+\tau}{P}\right)$ time, and the total time is $O\left(\frac{m+\tau}{P} + P + \Lambda\right) = O\left(\frac{m+\Lambda^2}{P} + P + \Lambda\right)$ w.h.p. □

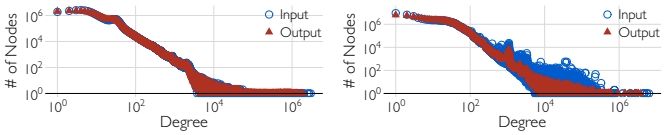
C. Experimental Results

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

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

Degree Distribution. Fig. 8 shows the input and generated degree distributions of Twitter and UK–Union graphs (see Table II). As observed from the figures, the generated degree distributions closely follow the input, which visually validates the correctness of our parallel algorithm. Some variations in the distribution are due to the randomness. We also experimented with several other graphs and observed the same result.

As a formal test, we use the Kullback–Leibler (KL) divergence [39] to compute the statistical difference between the input and output degree distributions. The KL divergence



(a) Twitter (b) UK–Union

Fig. 8: Degree distributions of input and generated graph

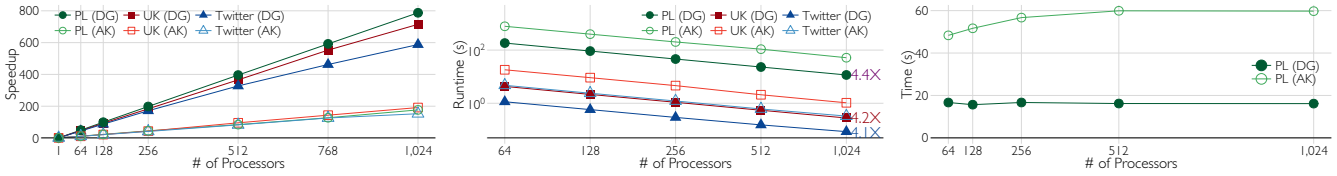
measures the difference between two probability distributions Q (input) and R (output) as information gain defined as:

$$D_{\mathrm{KL}}(Q\|R) = \sum_i Q(i) \log \frac{Q(i)}{R(i)}. \quad (6)$$

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

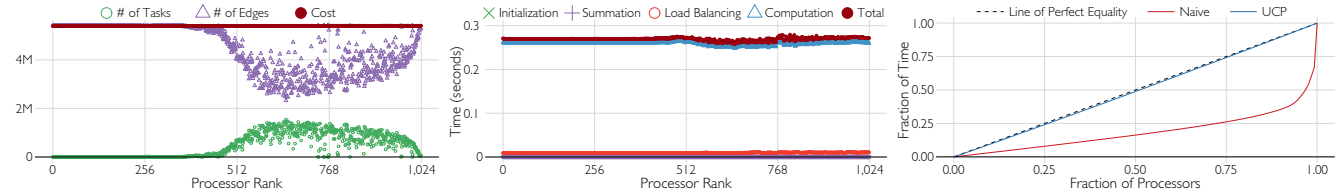
Strong and Weak Scaling. Strong scaling of a parallel algorithm shows its performance with the increasing number of processors while keeping the problem size fixed. Fig. 9(a) shows the speedup of our parallel algorithm along with the best known parallel algorithm [29] (referred as the AK algorithm) for a massive synthetic (PL) and two large real–world graphs (Twitter and UK-Union). Speedups are measured as $\frac{T_s}{T_p}$, where T_s and T_p are the running time of the sequential and the parallel algorithm, respectively. The number of processors is varied from 1 to 1024. As shown in Fig. 9(a), our algorithm achieves almost linear speedup for each graph. The AK algorithm also has a linear speedup. But our algorithm is approximately four times faster than the AK algorithm (see Fig. 9(b)). Moreover, our algorithm requires less memory ($O(\Lambda)$ memory) than the AK algorithm ($O(n)$ memory). For example, for the Twitter, UK-Union, and PL graphs, the DG algorithm takes about 440, 716, and 16000 times less memory than the AK algorithm, respectively. Thus, our algorithm scales to a large number of processors.

The weak scaling measures the performance of a parallel algorithm when the input size per processor remains constant. For this experiment, we varied the number of processors from 64 to 1024. For P processors, a PL graph with $10^6 P$ vertices and $10^8 P$ edges is generated. Note that weak scaling can only be performed on artificial graphs. Fig. 9(c) shows that our algorithm also achieves very good weak scaling compared to the AK algorithm with almost constant runtime.



(a) Strong Scaling (b) Runtime (log – log scale) (c) Weak Scaling

Fig. 9: Strong and weak scaling of the parallel algorithms



(a) Estimated cost (b) Time required among the processors (c) Lorenz Curve

Fig. 10: Load balancing of the parallel DG algorithm

Load Balancing. Our parallel algorithms provide very good load balancing. Fig. 10 demonstrates the quality of our load balancing approaches described in Section III-A. We formally quantify the quality of load balancing using Lorenz curves and Gini coefficients as described below. *Lorenz curves* [40], often used in economics for representing inequality of the distribution of wealth, visualizes the load disparities [41]. In a Lorenz curve, the cumulative proportion of a distribution is plotted against the cumulative proportion of ordered individuals. In our context, the ordered individuals are the processors ordered by their computational time. Let the computational times of the P processors are denoted and ordered by y_i, where $1 \leq i \leq P$ and $y_i \leq y_{i+1}$. Then the Lorenz curve is the continuous piecewise linear function joining the points $\left(\frac{x}{P}, \frac{\mathcal{Y}_x}{\mathcal{Y}_P}\right)$, where $x = 0, 1, 2, \ldots, P$, $\mathcal{Y}_0 = 0$, and $\mathcal{Y}_x = \sum_{i=1}^{x} y_i$. If all processors require the same amount of time, the Lorenz curve is a straight line called *the line of perfect equality*. For imbalanced loads, the curve falls below that line. The *Gini coefficient* $G \in [0, 1]$ [42] is defined as:

$$G = \left(\frac{2 \sum_{i=1}^{P} i y_i}{P \sum_{i=1}^{P} y_i} - \frac{P+1}{P} \right). \tag{7}$$

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

Balancing the workloads for the real-world graphs are more challenging than the synthetic graphs. Therefore, in this experiment we demonstrate for the UK-Union graph, the largest public real–world graph [35] using 1024 processors. We experimentally determined $\alpha = 2$ and $\beta = 1$ for the cost function (see Equation 4), which achieves the best load-balancing. Fig. 10(a) shows the number of tasks, edges, and the estimated cost of each processor. Note that the estimated cost is almost the same in each processor. Fig. 10(b) demonstrates the time required by each processor for initialization, summation, load–balancing and graph computation steps. Fig. 10(c) also shows the Lorenz curve based on the total time required by each processor. Our algorithm using the UCP–DIV task distribution scheme has a Gini coefficient of 0.015 indicating near perfect load balancing. In contrast, using a naïve scheme where each processor executes equal number of tasks, has a Gini coefficient of 0.63. The results strongly favor our choice of the cost function and the UCP–DIV task distribution algorithm. Thus, our algorithm achieves very good load–balancing, where each processor spends almost an equal amount of time.

IV. Block Two-Level Erdős–Rényi

The Block Two-Level Erdős–Rényi (BTER) is another model to generate random graphs using two fundamental properties: degree distribution and clustering coefficients [20, 21]. The clustering coefficient C_u of a vertex u is defined as the ratio of the number of edges among its neighbors to the maximum number of all possible such edges. More formally, $C_u = \frac{|\{(v,w) \in E : v, w \in \mathcal{N}_u\}|}{\binom{\delta_u}{2}}$, where $\mathcal{N}_u$ is the set of neighbors, and δ_u is the degree of u. The BTER model takes as input (1) the desired degree distribution $\{n_d\}_{d \in \mathbb{D}}$, and (2) the desired average clustering coefficients by degree $\{c_d\}_{d \in \mathbb{D}}$ where $c_d = \frac{1}{n_d} \sum_{\{u : \delta_u = d\}} C_u$. First, the vertices are divided into many blocks called *affinity blocks* based on their expected degrees. An affinity block with degree d contains $d+1$ vertices (except the last block). Typically, there are many small blocks with a low degree and a few large blocks with high degree vertices. Next, the edges are generated in two phases. In Phase 1, edges within each block are generated. Phase 1 generates triangle rich non-overlapping communities. Each block is represented by an ER model with probability p. For a block involving degree-d vertices, the probability is given as $p = \sqrt[3]{c_d}$. In Phase 2, edges across the blocks are created. Consider some vertex i with expected degree δ_i. Suppose, in Phase 1, the vertex created δ'_i edges. Then, $w_i = \delta_i - \delta'_i$ denotes the excess degree of the vertex i. To get the desired degree δ_i of a vertex i, w_i more edges need to be incident on i. The Chung-Lu (CL) model is applied to the expected degree sequence $\{w_i\}_{1 \leq i \leq n}$ to get the desired degree distribution.

A complete, scalable implementation of BTER model is given in [21]. We analyzed the BTER implementation and observed that both Phase 1 and Phase 2 can be improved significantly incorporating the DG method. To generate edges in both Phase 1 and Phase 2, the implementation uses sampling based edge generation similar to the approach proposed in [28]. In the sampling based algorithm, each edge is generated by randomly choosing two end-points based on their degrees. Therefore, there is a possibility that an edge is selected multiple times. Consider an affinity block b with n_b vertices and edge probability p_b in Phase 1. The expected number of edges generated by the ER model for the block b is $m_b = p_b\binom{n_b}{2}$. To ensure that m_b distinct edges are generated, their algorithm samples $w_b = \binom{n_b}{2} \ln \frac{1}{1-p_b}$ edges [21]. Note that $w_b \geq m_b$, because $\ln \frac{1}{1-p_b} \geq p_b$ for $0 \leq p_b \leq 1$. When the edge probability is relatively high w_b can be several factor larger than m_b. For example, during the generation of UK-Union graph, Phase 1 produces approximately 3.2B distinct edges. To generate those many edges, the original BTER implementation generates 8.9B edges, which is about 2.8 times more than the required number of edges (about 64% edges are discarded). Moreover, as the number of duplicate edges is very large, removing duplicates edges becomes very costly. In contrast, the edge skipping technique requires no unnecessary operation. All the edges generated in Phase 1 are distinct. Therefore, duplicate removal step is not required. Similar arguments can also be made for Phase 2. Moreover, as we saw earlier, edge skipping technique is highly scalable due to the ability to break a large task into multiple subtasks.

A. A DG–based Algorithm for the BTER Model

In this section, we present the DG–based BTER algorithm by incorporating the DG method. Due to the lack of space, we provide a brief outline of the algorithm here.

Sequential Algorithm. The sequential algorithm runs in three steps: a) Initialization, b) Phase 1, and c) Phase 2. In the initialization step, the affinity blocks are created using the procedure BTER_SETUP (Algorithm 1 in [21]). In Phase 1, edges are produced using the edge skipping technique using only the ER model. Next, in Phase 2, the DG algorithm for the CL model is applied. Here, the vertices are grouped based on their excess degrees as defined earlier. Note that the expected excess degrees can contain fractions. After grouping, the rest of the algorithm is similar to Algorithm 2.

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

B. Experimental Evaluation

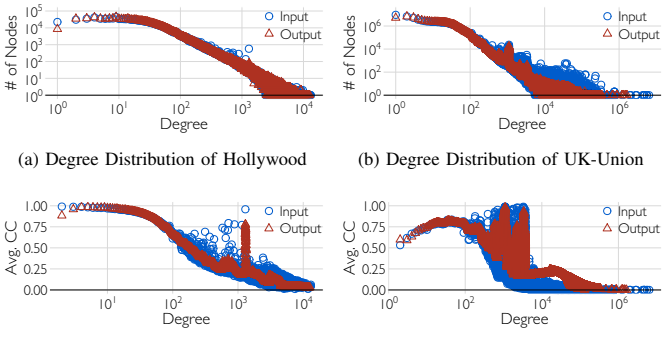
In this section, we evaluate the accuracy and performance of both our sequential and parallel algorithms with the original BTER implementation [21]. For fairness, we used the same set of graphs used in the original paper listed in Table III. Henceforth, we refer BTER as the original implementation [21], and DG–BTER as our algorithm.

TABLE III: Performance of the sequential algorithm for BTER

Graph	Vertices	Edges	Runtime (s)	
			BTER [21]	DG–BTER
LJournal [35]	5M	49M	160.21	1.99
Hollywood [35]	2M	115M	450.79	5.34
Twitter [5, 35]	41M	1.2B	230.00	48.38
UK-Union [35]	131M	4.6B	1350.00	209.74

Performance of the Sequential Algorithm. The runtimes of the BTER and the sequential DG–BTER algorithms are also shown in Table III. Runtimes for the LJournal and Hollywood graphs are collected from the MATLAB–based implementation of BTER. Twitter and UK–Union graphs are generated by scalable BTER on a Hadoop cluster with 32 computing nodes [21]. Our sequential DG–BTER algorithm not only outperforms the MATLAB–based BTER, it also outperforms the Hadoop–based scalable BTER implementation.

Degree Distribution and Average Clustering Coefficients. Due to page limit we only show the input and generated degree distributions and average clustering coefficients for two real–world graphs Hollywood and UK-Union in Fig. 11. The graphs are generated using our parallel DG–BTER algorithm. As observed from the plots, both the generated degree distributions and average clustering coefficients closely follow the input. Our experiments with other networks also give similar results. The corresponding plots in the BTER paper [21] are exactly the same as ours, verifying the correctness of DG–BTER.

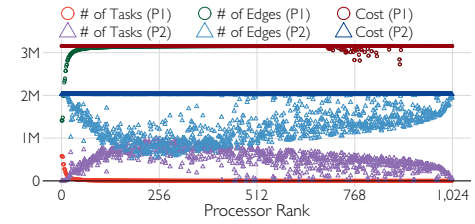


(a) Degree Distribution of Hollywood (b) Degree Distribution of UK-Union

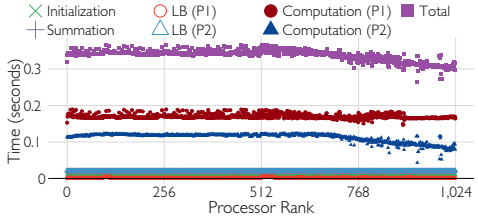
(c) Avg. CC of Hollywood (d) Avg. CC of UK-Union

Fig. 11: Degree distributions and average clustering coefficient per degree of input and generated networks

Load Balancing. Fig. 12 demonstrates the load balancing performance of our parallel BTER algorithm for the UK-Union graph. We experimentally determined the parameters of the cost functions where $\alpha_1 = 2, \alpha_2 = 1.5, \beta_1 = \beta_2 = 1$. Fig. 12(a) shows that the costs are distributed uniformly among the processors for both phases. Fig. 12(b) demonstrates that each processor takes almost the same amount of time in every step of the algorithms, i.e., loads are well balanced.



(a) Estimated Cost



(b) Time (per processor)

Fig. 12: Load Balancing of Parallel BTER. P1 and P2 denotes Phase 1 and Phase 2 respectively.

Strong Scaling. Fig. 13 shows the speedups of our parallel BTER algorithm. As Fig. 13 demonstrates, we also achieve a linear speedup for a large number of processors. For example using 1024 processors, our algorithm took only 0.37 seconds for the UK-Union graph, with a speed-up of about 572 (in contrast to the 1350 seconds required by the BTER algorithm). Also, note that the speed up increases with graph size. Therefore, our algorithm is suitable for fast generation of massive graphs, significantly faster than the existing algorithms.

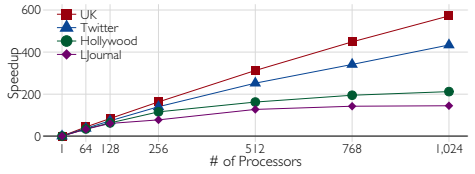


Fig. 13: Strong scaling of the parallel algorithms for BTER

V. Stochastic Block Model

Stochastic block model (SBM) is another popular model first studied in mathematical sociology [8, 43]. A stochastic block model is defined with the following three parameters: *1)* a set of n vertices, *2)* k disjoint subset of vertices $\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_k$, called communities, and *3)* a $k \times k$ matrix M, where $M_{i,j}$ denotes the probability that a vertex of community $\mathbb{C}_i$ is connected to a vertex of community $\mathbb{C}_j$. With a given set of parameters n, $\{\mathbb{C}_i\}_{1 \le i \le k}$, and M the edges are created as follows: any two nodes $u \in \mathbb{C}_i, v \in \mathbb{C}_j$ is connected with probability $M_{i,j}$. Note that for any pair of communities $\{\mathbb{C}_i, \mathbb{C}_j\}$, any possible edge $(u, v) : u \in \mathbb{C}_i, v \in \mathbb{C}_j$ is created with probability $p_{u,v} = M_{i,j}$, i.e., all potential edge in a pair of communities are independent and identically distributed. Observe that the groups in the DG algorithm are remarkably similar to the SBM communities. The main difference is that in the SBM model the probability p of an edge between two communities is provided in M, whereas in the DG algorithm, the probability depends on the degree of the groups. As a result, we can use the Algorithm 2 for generating edges efficiently using the SBM model by replacing the lines 9 and 11 with Edge–Skipping$(i, i, M_{i,i}, 1, \binom{|\mathbb{C}_i|}{2})$ and Edge–Skipping$(i, j, M_{i,j}, 1, |\mathbb{C}_i||\mathbb{C}_j|)$ respectively. Note that the parallel algorithm for the CL model can be applied to the SBM model in a similar fashion.

Performance of the Parallel Algorithm. Fig. 14 shows the speedup of our parallel algorithm for generating edges using the SBM model for two graphs with 300 and 1000 communities with 2.4B and 17.8B edges, respectively. We can see that the strong scaling of the algorithm is also linear. Therefore, our algorithm is efficient and scalable to a large number of processors. Our algorithm also achieves very good load balancing. The result of the load balancing is omitted due to lack of space.

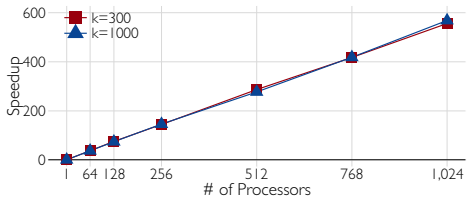


Fig. 14: Strong scaling of the parallel algorithm for the SBM

VI. Conclusion

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

VII. Acknowledgments

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

References

[1] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *SIGCOMM Comp. Comm. Rev.*, 1999.

[2] G. Siganos, M. Faloutsos, P. Faloutsos, and C. Faloutsos, "Power laws and the AS-level internet topology," *IEEE/ACM Tran. on Networking*, 2003.

[3] M. Girvan and M. Newman, "Community structure in social and biological networks," *Proc. Nat. Acad. Sci. U.S.A.*, 2002.

[4] J. Leskovec and E. Horvitz, "Planetary-scale views on a large instant-messaging network," in *Proc. of the 17th Intl. Conf. on World Wide Web*, 2008.

[5] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. of the 19th Intl. Conf. on World Wide Web*, 2010.

[6] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proc. of the Intl. Conf. on Data Mining*, 2012.

[7] P. Erdős and A. Rényi, "On the evolution of random graphs," in *Publ. of the Math. Inst. of the Hungarian Acad. of Sci.*, 1960.

[8] P. Holland, K. Laskey, and S. Leinhardt, "Stochastic blockmodels: First steps," *Social networks*, 1983.

[9] D. Watts and S. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, 1998.

[10] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, 1999.

[11] R. Albert, H. Jeong, and A.-L. Barabási, "Error and attack tolerance of complex networks," *Nature*, 2000.

[12] O. Frank and D. Strauss, "Markov graphs," *J. Amer. Statist. Assoc.*, 1986.

[13] G. Robins, P. Pattison, Y. Kalish, and D. Lusher, "An introduction to exponential random graph (p*) models for social networks social networks," *Soc. Net.*, 2007.

[14] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SIAM Data Mining*, 2004.

[15] J. Leskovec and C. Faloutsos, "Scalable modeling of real graphs using Kronecker multiplication," in *Proc. of the 24th Intl. Conf. on Machine Learning*, 2007.

[16] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, 2010.

[17] J. Carlson and J. Doyle, "Highly optimized tolerance: a mechanism for power laws in designed systems," *Phys. Rev. E*, 1999.

[18] F. Chung and L. Lu, "Connected components in random graphs with given expected degree sequences," *Annals of Combinatorics*, 2002.

[19] ——, "The average distance in a random graph with given expected degrees," *Internet Mathematics*, 2002.

[20] C. Seshadhri, T. Kolda, and A. Pinar, "Community structure and scale-free collections of Erdős-Rényi graphs," *Phys. Rev. E*, 2012.

[21] T. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *SIAM J. Sci. Comput.*, 2014.

[22] R. Bridges, J. Collins, E. Ferragut, J. Laska, and B. Sullivan, "Multi-level anomaly detection on time-varying graph data," in *Adv. in Soc. Net. Anlys. and Mining*, 2015.

[23] I. Stanton and A. Pinar, "Constructing and sampling graphs with a prescribed joint degree distribution," *J. of Experiment. Algorithmics*, 2012.

[24] J. Leskovec, "Dynamics of large networks," Ph.D. dissertation, CMU, 2008.

[25] V. Batagelj and B. Ulrik, "Efficient generation of large random networks," *Phys. Rev. E*, 2005.

[26] J. Miller and A. Hagberg, "Efficient generation of networks with given expected degrees," in *Proc. of Algorithms and Models for the Web-Graph*, 2011.

[27] "Graph500," 2015, http://www.graph500.org/.

[28] A. Pinar, C. Seshadhri, and T. Kolda, "The similarity between stochastic Kronecker and Chung-Lu graph models," in *SIAM Data Mining*, 2012.

[29] M. Alam and M. Khan, "Parallel algorithms for generating massive random networks with given degree sequences," *Intl. J. Par. Prog.*, 2015.

[30] E. Bender and E. Canfield, "The asymptotic number of labeled graphs with given degree sequences," *J. Comb. Theory, Series A*, 1978.

[31] M. Molloy and B. Reed, "A critical point for random graphs with a given degree sequence," *Random Structures & Algorithms*, 1995.

[32] Y. Shang, "Groupies in random bipartite graphs," *Applicable Analysis and Discrete Mathematics*, 2010.

[33] W. Aiello, F. Chung, and L. Lu, "A random graph model for power law graphs," *Experimental Math*, 2000.

[34] C. Barrett, R. Beckman, M. Khan, V. Kumar, M. Marathe, P. Stretz, T. Dutta, and B. Lewis, "Generation and analysis of large synthetic social contact networks," in *Proc. of the Winter Sim. Conf.*, 2009.

[35] P. Boldi, M. Santini, and S. Vigna, "A large time-aware graph," *SIGIR Forum*, 2008.

[36] A. Pinar and C. Aykanat, "Fast optimal load balancing algorithms for 1D partitioning," *J. Par. Dist. Comp.*, 2004.

[37] F. Manne and T. Sørevik, "Optimal partitioning of sequences," *J. of Algorithms*, 1995.

[38] B. Olstad and F. Manne, "Efficient partitioning of sequences," *IEEE Trans. on Comput.*, 1995.

[39] S. Kullback and R. Leibler, "On information and sufficiency," *Ann. Math. Stat.*, 1951.

[40] M. Lorenz, "Methods of measuring the concentration of wealth," *J. Amer. Statist. Assoc.*, 1905.

[41] T. Pitoura, N. Ntarmos, and P. Triantafillou, "Replication, load balancing and efficient range query processing in DHTs," in *Intl. Conf. on Adv. in Db. Tech.*, 2006.

[42] C. Gini, "Variabilità e mutabilità," *Memorie di Metodologica Statistica*, 1912.

[43] Y. Wang and G. Wong, "Stochastic blockmodels for directed graphs," *J. Amer. Statist. Assoc.*, 1987.