Generating Massive Scale-free Networks: Novel Parallel Algorithms using the Preferential Attachment Model

MAKSUDUL ALAM, Oak Ridge National Laboratory, USA
MALEQ KHAN, Texas A&M University-Kingsville, USA
KALYAN S. PERUMALLA, Oak Ridge National Laboratory, USA
MADHAV MARATHE, Network Systems Science and Advanced Computing Division, Biocomplexity Institute and Initiative & Department of Computer Science, University of Virginia, USA

Recently, there has been substantial interest in the study of various random networks as mathematical models of complex systems. As real-life complex systems grow larger, the ability to generate progressively large random networks becomes all the more important. This motivates the need for efficient parallel algorithms for generating such networks. Naïve parallelization of sequential algorithms for generating random networks is inefficient due to inherent dependencies among the edges and the possibility of creating duplicate (parallel) edges. In this article, we present message passing interface-based distributed memory parallel algorithms for generating random scale-free networks using the preferential-attachment model. Our algorithms are experimentally verified to scale very well to a large number of processing elements (PEs), providing near-linear speedups. The algorithms have been exercised with regard to scale and speed to generate scale-free networks with one trillion edges in 6 minutes using 1,000 PEs.

CCS Concepts: • **Theory of computation → Random network models**; **Parallel algorithms**; **Distributed algorithms**; • **Mathematics of computing → Random graphs**; **Graph algorithms**;

Additional Key Words and Phrases: Network science, random networks, preferential attachment, distributed algorithms

This article has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.
This work has been partially supported by NSF DIBBs Grant No. ACI-1443054, DTRA Grant No. HDTRA1-11-1-0016, DTRA CNIMS Contract No. HDTRA1-11-D-0016-0001, NSF NetSE Grant No. CNS-1011769, NSF BIGDATA Grant No. IIS-1633028, and ORNL Post Doctoral Education Investment No. 3410PDED.
Authors' addresses: M. Alam and K. S. Perumalla, Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, TN, 37831, USA; emails: {alamm,perumallaks}@ornl.gov; M. Khan, Texas A&M University-Kingsville, 700 University Blvd., Kingsville, TX, 78363, USA; email: maleq.khan@tamuk.edu; M. Marathe, Network Systems Science and Advanced Computing Division, Biocomplexity Institute and Initiative & Department of Computer Science, University of Virginia, Research Park Blvd., Charlottesville, VA, 22904, USA; email: marathe@virginia.edu.

2329-4949/2020/05-ART13 $15.00
https://doi.org/10.1145/3391446

ACM Reference format:
Maksudul Alam, Maleq Khan, Kalyan S. Perumalla, and Madhav Marathe. 2020. Generating Massive Scale-free Networks: Novel Parallel Algorithms using the Preferential Attachment Model. *ACM Trans. Parallel Comput.* 7, 2, Article 13 (May 2020), 35 pages.
https://doi.org/10.1145/3391446

1 INTRODUCTION

1.1 Motivation

Advances in hardware, software, and algorithms have enabled the detailed study of complex networks. Complex networks such as the Internet [23, 44], biological networks [27], social networks [33, 36], and various infrastructure networks [13, 18, 34] are abstracted as random graphs for obtaining rigorous mathematical results; see, e.g., Reference [18]. The study of these complex systems have significantly increased the interest in various random graph models [14]. With the growth of complex networks, it has become necessary to generate massive random networks efficiently.

Many random graph models have been developed in the past to analyze complex systems. Among them, the first and well-studied model is the Erdős–Rényi model [21]. However, the Erdős–Rényi model does not exhibit the characteristics observed in many real-world complex systems [14]. As a result, many other random graph models, such as small-world [45], Barabási–Albert [8, 11], Chung-Lu [39], exponential random graph [25, 42], R-MAT [17], and HOT [16] models, have been proposed. Furthermore, a smaller network may not exhibit the same behavior of larger networks, even if both of the networks are generated using the same model. The structure of larger networks is fundamentally different from small networks, and many patterns emerge only in massive datasets [36]. In the areas of network science and data mining as well as social sciences and physics, large-scale network analysis is becoming a dominant field [10].

Demand for large random networks necessitates efficient, both in terms of running time and memory consumption, algorithms to generate such networks. Although various random graph models are being used and studied over the last several decades, even efficient sequential algorithms for generating such graphs were nonexistent until recently. As a step toward meeting this goal, recently efficient sequential algorithms have been developed to generate certain classes of random graphs: Erdős–Rényi [14], small world [14], Preferential Attachment [14, 40], and Chung-Lu [39]. However, although efficient sequential algorithms are able to generate networks up to millions of vertices and edges quickly, generating networks with billions of vertices and edges can take substantially longer. Further, a large memory requirement often makes generation of such large networks using these sequential algorithms infeasible. Shared memory parallel machines provide one alternative to overcome the problems. Distributed memory parallel algorithms provide another natural alternative.

Preferential attachment is a model that generates random scale-free networks, where a new vertex makes connections to some existing vertices that are chosen preferentially based on some of the properties of those vertices [11]. The preferential attachment model largely explains many structural properties observed in real-world networks such as power-law degree distribution, long tails, and high degree vertices or hubs. The model is fundamental to understand how a simple process can lead to the formation of real-world networks. The model has been widely used for community detection [27], biological modeling [12], epidemic spreading [41], evaluating electric grid [18], scientific collaboration [47], and many other areas. For the preferential attachment model, the earliest known distributed-memory parallel algorithm is given by Yoo and Henderson [46]. Although useful, the algorithm has two weaknesses: (i) to deal with dependencies and the required complex synchronization, they presented an approximation algorithm rather than an exact algorithm; and

(ii) the accuracy of their algorithm depends on several control parameters, which are manually adjusted by running the algorithm repeatedly. Several other studies were done on the preferential attachment-based models. Machta and Machta [37] described how an evolving network can be generated in parallel. Dorogovtsev et al. [20] proposed a model that can generate graphs with fat-tailed degree distributions. In this model, starting with some random graphs, edges are randomly rewired according to some preferential choices. A literature review of the recent developments is presented in Section 6.

In this article, we study the problem of designing a distributed memory parallel algorithm for generating massive scale-free networks based on the preferential attachment (PA) model. The rest of the article is organized as follows. Notations and a description of the parallel computation model are given in Section 1.2. In Section 1.3, we describe the problem along with sequential algorithms. In Section 2, we present our parallel algorithm in a distributed memory architecture for the case where each vertex connects a single edge to the existing network. In Section 3, we extend the algorithm for the general case where each vertex contributes $x \geq 1$ edges to the existing network. In Section 4, we present and analyze the partitioning and load balancing techniques. Experimental results showing the performance of our parallel algorithms are presented in Section 5. In Section 6, we present a literature review of recent developments on scale-free network generators. We summarize the findings and indicate potential future work in Section 7.

1.2 Preliminaries and Notations

In the rest of the article, we use the following notations. We denote a network $G(V, E)$, where V and E are the sets of vertices and edges, respectively, with $m = |E|$ edges and $n = |V|$ vertices labeled as $0, 1, 2, \ldots, n-1$. We denote $(u, v) \in E$ as an undirected edge where $u, v \in V$. If $(u, v) \in E$, then we say u and v are *neighbors* of each other. The set of all neighbors of $v \in V$ is denoted by $N(v)$, i.e., $N(v) = \{u \in V | (u, v) \in E\}$. The degree of v is $d_v = |N(v)|$. If u and v are neighbors, then sometime we say that u is *connected* to v and vice versa.

We develop parallel algorithms for the message passing interface (MPI)-based distributed memory system, where each processing element (PE) performs a single process, does not have any shared memory, and has its own local memory. The PEs can exchange data and communicate with each other by exchanging messages. The PEs can read and write data from files in a shared file system. However, such reading and writing of the files are done independently.

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

1.3 Background: Preferential Attachment Model

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

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

Barabási-Albert Model. One way to generate a random PA network is to use a generative model proposed by Barabási and Albert. Many real-world networks have two important characteristics: (i) they are evolving in nature and (ii) the network tends to be scale free [11]. They provided a model, known as the Barabási-Albert (BA) model, where a new vertex is connected to an existing vertex that is chosen with probability directly proportional to its current degree.

The BA model works as follows. Starting with a small connected graph of $\hat{x}$ vertices, in every time step, a new vertex t is added to the network and connected to $x \leq \hat{x}$ randomly chosen distinct existing vertices: $F_t(k)$ for $1 \leq k \leq x$ with $F_t(k) < t$; that is, $F_t(k)$ denotes the kth vertex to which t is connected. Thus, each phase adds x new edges $(t, F_t(1)), (t, F_t(2)), \ldots, (t, F_t(x))$ to the network, which exhibits the evolving nature of the model. For each of the x new edges, vertices $F_t(1), F_t(2), \ldots, F_t(x)$ are randomly selected based on the degrees of the vertices in the current network. In particular, a new vertex t connects to x distinct existing vertices where each vertex $i < t$ is selected with probability $P_t(i) = \frac{d_i}{\sum_j d_j}$.

The networks generated by the BA model are called the BA networks, which bear the two characteristics of a real-world network mentioned above, i.e., evolving nature and scale-free property. BA networks have power law degree distribution. A degree distribution is called power law if the probability that a vertex has degree d is given by $\Pr[d] \propto d^{-\gamma}$, where γ is a positive constant. Barabási and Albert showed this preferential attachment method of selecting vertices results in a power-law degree distribution [11].

First, we assume $x = 1$, and for this case, we use F_t for $F_t(1)$. We discuss the general case $x \geq 1$ later. One naïve approach is to maintain a list of the degrees of the vertices, and in each time step t, generate a uniform random number in $\left[1, \sum_{i=0}^{t-1} d_i\right]$ and scan the list of the degrees sequentially to find F_t. In this case, phase t takes $\Theta(t)$ time, and the total time is $\Omega(n^2)$. Batagelj and Brandes give an efficient algorithm with running time $O(m)$ [14]. This algorithm maintains a list of vertices such that each vertex i appears in this list exactly d_i times. The list can easily be updated dynamically by simply appending u and v to the list whenever a new edge (u, v) is added to the network. Now to find F_t, a vertex is chosen from the list uniformly at random. Since each vertex i occurs exactly d_i times in the list, we have $\Pr[F_t = i] = \frac{d_i}{\sum_j d_j}$. Although we have described the algorithm using list for clarity, in practice we use an array for performance instead of list.

Copy Model. As it turns out, the BA model does not easily lend itself to an efficient parallelization. Another algorithm called the *copy model* [31, 32] preserves preferential attachment and power-law degree distribution. The copy model works as follows. Similar to the BA model, it starts with a small connected graph of $\hat{x}$ vertices and in every time step, a new vertex t is added to the network to create $x \leq \hat{x}$ connections to existing vertices $F_t(\ell)$ for $1 \leq \ell \leq x$ with $F_t(\ell) < t$. For each connection $(t, F_t(\ell))$ from vertex t the following steps are executed:

Step 1: First a random vertex $k \in [0, t-1]$ is chosen with uniform probability.

Step 2: Then $F_t(\ell)$ is determined as follows:

$$F_t(\ell) = \begin{cases} k & \text{with probability } p \text{ (Direct Edge)} \qquad (1) \\ F_k(l) & \text{with probability } 1-p \text{ (Copy Edge),} \qquad (2) \end{cases}$$

where l is the index of a random outgoing connection from vertex k. Note that for a disconnected vertex k in the initial graph, we also assume $F_k(l) = k$ for any l where $k < \hat{x}$. We also denote $\mathbb{F}_t = \{F_t(1), F_t(2), \ldots, F_t(x)\}$ to be the set of outgoing vertices from vertex t.

It can be easily shown that a connection from vertex t to vertex i is made with probability $\Pr[i \in \mathbb{F}_t] = \frac{d_i}{\sum_j d_j}$ when $p = \frac{1}{2}$. Thus, when $p = \frac{1}{2}$, this algorithm follows the Barabási-Albert model as shown in Theorem 1.1 [2, 3].

Theorem 1.1. *The Barabási-Albert model is a special case of the copy model when* $p = \frac{1}{2}$.

Table 1. Symbols Used in this Article

Symbol	Description
n	The number of vertices
V	The set of vertices
m	The number of edges
E	The set of edges
x	The number of outgoing edges generated from each new vertex
p	The probability of creating a direct edge in the copy model
$N(v)$	The set of neighbors of vertex v
d_v	The degree of vertex v
$F_t(k)$	The outgoing end of kth edge from vertex t
$\mathbb{F}_t$	The set of outgoing ends of edges from vertex t

Proof. A vertex i can be selected in $\mathbb{F}_t$ in two mutually exclusive ways: (i) i is chosen in the first step and assigned to an outgoing edge of t in the second step (Equation (1)), which occurs with probability $\frac{1}{t} \cdot p$ or (ii) a neighbor of i, $v \in \{u | i \in \mathbb{F}_u\}$, is chosen in the first step ($d_i - x$ possible such neighbors), and the outgoing edge to i is selected (out of x outgoing edges from v) in the second step (Equation (2)), which occurs with probability $\frac{d_i - x}{t} \cdot (1-p) \cdot \frac{1}{x}$, where d_i is the total degree of vertex i. Thus, we have

$$\begin{aligned}\Pr[i \in \mathbb{F}_t] &= \frac{1}{t} \cdot p + \frac{d_i - x}{t} \cdot (1-p) \cdot \frac{1}{x} \\ &= \frac{xp + (d_i - x)(1-p)}{xt} \\ &= \frac{xp + (d_i - x)(1-p)}{\frac{1}{2}\sum_j d_j} \qquad \left[\sum_j d_j = 2xt\right]. \end{aligned} \tag{3}$$

When $p = \frac{1}{2}$, $\Pr[i \in \mathbb{F}_t] = \frac{d_i}{\sum_j d_j}$. □

The copy model is more general than the BA model and produces networks with degree distribution following a power law $d^{-\gamma}$, where the value of the exponent γ depends on the choice of p [32]. Further, it is easy to see that the running time of the copy model is $O(m)$. We found that the copy model leads to more efficient parallel algorithms for generating preferential attachment networks and develop our parallel algorithm based on the copy model.

To summarize, Table 1 lists the symbols used in this article.

2 SIMPLIFIED PARALLEL APPROACH FOR $x = 1$

The dependencies among the edges pose a major challenge in parallelizing preferential attachment algorithms. In phase t, to determine F_t requires that F_i is known for each $i < t$. As a result, any algorithm for preferential attachment seems to be highly sequential in nature: phase t cannot be executed until all previous phases are completed. However, a careful observation reveals that F_t can be partially, or sometimes completely, determined even before completing the previous phases. The copy model helps us exploit this observation in designing a parallel algorithm. However, it requires complex synchronizations and communications among the PEs. To keep the algorithm efficient, such synchronizations and communications must be done carefully. In this section, we present a parallel algorithm based on the copy model. For ease of discussion, we first present our algorithm for the case $x = 1$. We present the general case $x \geq 1$ in Section 3.

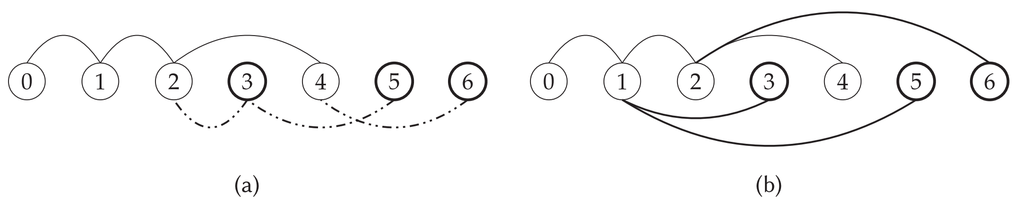


Fig. 1. A network with seven vertices generated by Algorithm 1: (a) an intermediate instance of the network in the middle of the execution of the algorithm, (b) the final network. Solid lines show final resolved edges, and dashed lines show waiting of the vertices. For example, for vertex $t = 4$, k is chosen to be 2, F_4 is chosen to be set to $k = 2$ (in Lines 2–5), and thus edge $(4, 2)$ is finalized immediately. For vertex $t = 5$, k is 3 and F_5 is set to be F_3 (in Line 7); as a result, determination of F_5 is waited until F_3 is known. At the end, we have $F_5 = F_3 = F_2 = 1$.

Let P be the number of PEs. The set of vertices V is partitioned into P disjoint subsets of vertices $V_0, V_1, \ldots, V_{P-1}$; that is, $V_i \subset V$, such that for any i and j, $V_i \cap V_j = \emptyset$ and $\bigcup_i V_i = V$. PE $\mathcal{P}_i$ is responsible for computing and storing F_t for all $t \in V_i$. The load balancing and performance of the algorithm crucially depend on how V is partitioned. The details of vertex partitioning are presented in Section 4.

2.1 Parallel Algorithm

The basic principle behind our parallel algorithm is as follows. Recall the sequential algorithm for the copy model. Each PE $\mathcal{P}_i$ can independently compute step 1 for each $t \in V_i$, as a random $k \in [0, t-1]$ is chosen with uniform probability (independent of the vertex degrees). Also, in step 2, if F_t is chosen to be k, F_t is determined immediately. If F_t is chosen to be F_k, then determination of F_t needs to wait until F_k is known. If $k \in V_j$ where $i \neq j$, then PE $\mathcal{P}_i$ sends a *request* message to PE $\mathcal{P}_j$ to find F_k. Note that at the time when PE $\mathcal{P}_j$ receives this message, F_k can still be unknown. If so, then $\mathcal{P}_j$ keeps this message in a queue called *waiting queue* until F_k is known. Once F_k is known, $\mathcal{P}_j$ sends back a *resolved* message to $\mathcal{P}_i$. The basic method executed by a PE $\mathcal{P}_i$ is given in Algorithm 1. An example instance of the execution of this algorithm with seven vertices is depicted in Figure 1.

2.2 Analysis of the Simplified Algorithm: Dependency Chains

In our parallel algorithm, it is possible that computation of F_t for some vertex t can wait until F_k for some other vertex k is known. Such waiting can form a chain, namely, a *dependency chain*. For example, as demonstrated in Figure 1, computation of F_5 is waiting for F_3, which in turn is waiting for F_2, and thus we have chain of dependency $\langle 5, 3, 2 \rangle$. If the lengths of these chains are large, then the waiting period for some vertices can be quite long, leading to poor performance of the parallel algorithm. Fortunately, the length of a dependency chain is small, and the performance of the algorithm is hardly affected by such waiting.

For the ease of analysis, first we formally define a dependency chain for $x = 1$ and provide a rigorous analysis showing that the maximum length of a dependency chain is at most $O(\log n)$ with high probability (w.h.p.). For large n, $O(\log n)$ is small compared to n. Moreover, while $O(\log n)$ is the maximum length, most of the chains have much smaller length. It is easy to see that for a constant p, the average length of a dependency chain is also constant, which is at most $\frac{1}{p}$. For an arbitrary p, the average length is still bounded by $\log n$ as shown in Theorem 2.3. Thus, while for some vertices a PE may need to wait for $O(\log n)$ steps, the PE hardly remains idle as it has other vertices to work with.

ALGORITHM 1: Simplified Parallel Algorithm for $x = 1$

```
/* Each PE $\mathcal{P}_i$ executes the following in parallel:                    */
1  foreach $t \in V_i$ do
2      $k \leftarrow$ a uniform random vertex in $[0, t-1]$
3      $c \leftarrow$ a uniform random real number in $[0, 1]$
4      if $c < p$ then                                   // i.e., with probability $p$
5          $F_t \leftarrow k$
6      else
7          $F_t \leftarrow$ NULL                              // to be set later to $F_k$
8          send message $\langle$request, $t, k\rangle$ to $\mathcal{P}_j$, where $k \in V_j$
/* Next, PE $\mathcal{P}_i$ receives messages sent to it and processes them as follows:   */
9  Upon receipt of message $\langle$request, $t', k'\rangle$ from $\mathcal{P}'_j$:      // note that $k' \in V_i$
10 if $F_{k'} \neq$ NULL then
11     send message $\langle$resolved, $t', F_{k'}\rangle$ to $\mathcal{P}'_j$
12 else
13     store $t'$ in queue $Q_{k'}$
14 Upon receipt of message $\langle$resolved, $t, v\rangle$:
15 $F_t \leftarrow v$
16 foreach $t' \in Q_t$ do
17     send message $\langle$resolved, $t', v\rangle$ to $\mathcal{P}_j$ where $t' \in V_j$
```

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

In the next step (see Equation (2) and Lines 2–5 of Algorithm 1), F_t is computed by assigning k or F_k to it. If F_k is selected to be assigned to F_t, then F_t cannot be determined until F_k is known; that is, the computation of F_t for vertex t depends on vertex k. In such a case, we say vertex t is dependent on k; otherwise, we say vertex t is independent. In turn, vertex k can depend on some other vertex, and eventually such successive dependencies can form a dependency chain. Formally, a *dependency chain* D_t starting at vertex t is a sequence of vertices $\langle v_0, v_1, v_2, \ldots, v_i, \ldots v_y\rangle$ such that $v_0 = t$, v_i depends on v_{i+1} for $0 \le i < y$, and v_y is independent. Notice that if $v_i \in D_t$, D_{v_i} is a subsequence and a suffix of D_t. Also it is easy to see that D_t is a subsequence and a prefix of S_t, and we have $|D_t| \le |S_t|$. Examples of a selection chain and a dependency chain are shown in Figure 2. Bounds on the length of dependency chains are given in Theorem 2.3. The following lemmas, Lemmas 2.1 and 2.2, are needed to prove Theorem 2.3.

Lemma 2.1. *Let $P_t(i)$ be the probability that vertex i is in selection chain S_t starting at vertex t. Then for any $1 \le i < t$, $P_t(i) = \frac{1}{i}$.*

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

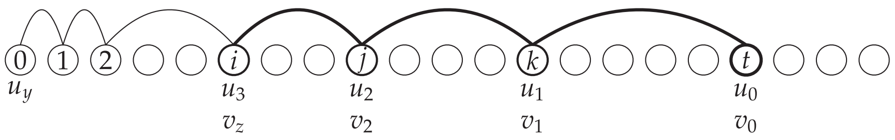


Fig. 2. Selection chain and dependency chain. The entire chain, which is marked by the solid lines, is a selection chain $\langle t, k, j, i, 2, 1, 0\rangle$, and the sub-chain marked by the thick solid lines is a dependency chain $\langle t, k, j, i\rangle$.

$\frac{1}{t}$, and i is in S_k. Hence, for $1 \le i < t$, we have

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

$$tP_t(i) = 1 + \sum_{k=i+1}^{t-1} P_k(i). \tag{4}$$

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

$$(t+1)P_{t+1}(i) = 1 + \sum_{k=i+1}^{t} P_k(i). \tag{5}$$

By subtracting Equation (4) from Equation (5),

$$(t+1)P_{t+1}(i) - tP_t(i) = P_t(i),$$

$$P_{t+1}(i) = P_t(i). \tag{6}$$

From Equation (6) by induction, we have $P_k(i) = P_t(i)$ for any k and t such that $1 \le i < \min\{k, t\}$. Now consider $k = i+1$. Notice that i is in S_{i+1} if and only if i is selected for vertex $i+1$; that is, $P_{i+1}(i) = \frac{1}{i}$. Hence, for any $t > i$, we have

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

□

Lemma 2.2. *Let A_i denote the event that $i \in S_t$. Then the events A_i for all i, where $1 \le i < t$, are mutually independent.*

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

$$\Pr\left[\bigcap_{k=1}^{\ell} A_{i_k}\right] = \prod_{k=1}^{\ell} \Pr\left[A_{i_k}\right]. \tag{7}$$

We know

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

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

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

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

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

By Lemma 2.1, we have $\Pr\left[i_1 \in S_{i_2}\right] = \frac{1}{i_1} = \Pr\left[i_1 \in S_t\right] = \Pr\left[A_{i_1}\right]$, and thus,

$$\Pr\left[\bigcap_{k=1}^{\ell} A_{i_k}\right] = \Pr\left[A_{i_1}\right] \cdot \Pr\left[\bigcap_{k=2}^{\ell} A_{i_k}\right]. \tag{8}$$

Next, by using Equation (8) and applying induction on ℓ, we prove Equation (7). The base case, $\ell = 2$, follows immediately from Equation (8):

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

By induction hypothesis, for $\ell - 1$ events $A_{i_k}, 2 \leq k \leq \ell$, we have $\Pr\left[\bigcap_{k=2}^{\ell} A_{i_k}\right] = \prod_{k=2}^{\ell} \Pr\left[A_{i_k}\right]$. Then, using Equation (8) for case $2 < \ell < t$, we have

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

□

Theorem 2.3. *Let L_t be the length of the dependency chain starting at vertex t and $L_{\max} = \max_t L_t$. Then the expected length $E[L_t] \leq \log n$ and $L_{\max} = O(\log n)$ w.h.p., where n is the number of vertices.*

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

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

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

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

where H_{t-1} is the $(t-1)$th harmonic number.

By Lemma 2.2, the random variables $X_t(i)$, for $1 \leq i < t$, are mutually independent. Applying the Chernoff bound on independent Poisson trials, we have

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

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

$$\begin{aligned}\Pr\left[L_t \geq 6\log n\right] &= \Pr\left[L_t \geq (1+\delta)\mu\right] \\ &\leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \leq \left(\frac{e}{1+\delta}\right)^{\mu(1+\delta)} \leq \left(\frac{e\mu}{6\log n}\right)^{6\log n} \\ &\leq \left(\frac{e\log n}{6\log n}\right)^{\log n^6} \leq \frac{1}{\left(\frac{6}{e}\right)^{\log n^6}} \leq \frac{1}{n^4}.\end{aligned}$$

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

3 GENERALIZED PARALLEL SOLUTION FOR $x \geq 1$

In Section 2, we presented the algorithm for the simpler case $x = 1$. In this section, we modify this algorithm for the general case where each vertex creates $x \geq 1$ edges. The basic structure of the algorithm for the general case is the same as that of the special case $x = 1$. We focus our discussion only on the modifications required and the differences between the two cases. The main difference is that for each vertex t, instead of computing one edge (t, F_t), we need to compute x edges $(t, F_t(1)), (t, F_t(2)), \ldots, (t, F_t(x))$. For this general case, the set of vertices $\{F_t(1), F_t(2), \ldots, F_t(x)\}$ is denoted by $\mathbb{F}_t$. Note that the preferential attachment model does not require all of the x new edges to be distinct, i.e., the graph may contain duplicate edges. However, depending on the context, a duplicate-free simple graph is often desired. Therefore, we present two parallel algorithms for generating the graphs with and without duplicate edges. Both of the algorithms start with an initial graph with the first x vertices labeled $0, 1, 2, \ldots, x-1$. Each of the other vertices from x to $n-1$ generates x new edges.

3.1 Parallel Algorithm with Duplicate Edges

We present the pseudo-code of the parallel algorithm with duplicate edges in Algorithm 2. The flow of the algorithm closely resembles the Algorithm 1. The major difference is that we execute the copy model x times in Line 2 and pick a random outgoing edge in Line 8. The waiting queue, request and response messages are processed in the same fashion as done in Algorithm 1.

3.2 Parallel Algorithm without Duplicate Edges

Now, we present the pseudo-code of the parallel algorithm without duplicate edges in Algorithm 3. There are fundamentally two important issues that need to be handled for the general case: (i) how we select $F_t(\ell)$ for vertex t where $1 \leq \ell \leq x$, and (ii) how we avoid duplicate edge creation. Multiple edges for a vertex t are created by repeating the same procedure x times (Line 2), and duplicate edges are avoided by simply checking if such an edge already exists and redoing the copy model. Such checking is done whenever a new edge is created.

For the ℓth edge of a vertex t, another vertex k is uniformly chosen at random from $[0, t-1]$ (Line 3). Edge (t, k) is created with probability p (Line 5). However, before creating such an edge (t, k) in Line 7, the existence of such an edge is checked immediately before creating it in Line 6. We used an array-based binary search tree (BST) to store and check the existence of the duplicate edges. Note that the search operation of the BST is $O(\log x)$. If the edge already exists at that time, then the process is repeated again (Line 9). With the remaining $1-p$ probability, t is connected to some vertex in $\mathbb{F}_k$; that is, we make an edge $(t, F_k(\ell))$, such that ℓ is uniformly chosen at random

ALGORITHM 2: Generalized Parallel Algorithm for $x \geq 1$ with Duplicate Edges

```
   /* Each PE $\mathcal{P}_i$ executes the following in parallel:                    */
 1 foreach $t \in V_i$ do
 2     for $\ell = 1$ to $x$ do
 3         $k \leftarrow$ a uniform random vertex in $[0, t-1]$
 4         $c \leftarrow$ a uniform random real number in $[0, 1]$
 5         if $c < p$ then                                  // i.e., with probability $p$
 6             $F_t(\ell) \leftarrow k$
 7         else
 8             $l \leftarrow$ a uniform random number in $[1, x]$
 9             $F_t(\ell) \leftarrow$ NULL                        // to be set later to $F_l(k)$
10             send message ⟨request, $F_t(\ell), F_k(l)$⟩ to $\mathcal{P}_j$, where $k \in V_j$
   /* Next, PE $\mathcal{P}_i$ receives messages sent to it and processes them as follows:   */
11 Upon receipt of message ⟨request, $F_{t'}(\ell'), F_{k'}(l')$⟩ from $\mathcal{P}_{j'}$:    // note that $k' \in V_i$
12 if $F_{k'}(l') \neq$ NULL then
13     send message ⟨resolved, $F_{t'}(\ell'), F_{k'}(l')$⟩ to $\mathcal{P}_{j'}$
14 else
15     store ⟨$F_{t'}(\ell'), F_{k'}(l')$⟩ in queue $Q_{k'}$
16 Upon receipt of message ⟨resolved, $F_t(\ell), v$⟩:
17 $F_t(\ell) \leftarrow v$
18 foreach ⟨$F_{t'}(\ell'), F_t(\ell)$⟩ $\in Q_t$ do
19     send message ⟨resolved, $F_{t'}(\ell'), v$⟩ to $\mathcal{P}_j$ where $t' \in V_j$
```

from $[1, x]$. Similar to the special case $x = 1$, if k is in another PE, a request message is sent to that PE to find $F_k(\ell)$ (Line 14). The request and response messages are also processed in the same way.

Duplicate edges can also be created during the execution of Line 19. For example, suppose vertex t creates two edges $(t, F_k(\ell))$ and $(t, F_{k'}(\ell'))$. Also, assume both k and k' are not in the same PE as t. Hence, request messages are sent to the PEs containing k and k' to resolve $F_k(\ell)$ and $F_{k'}(\ell')$. If the ℓth edge of k and ℓ'th edge of k' both connect to the same vertex u, then $F_k(\ell) = F_{k'}(\ell') = u$. Hence, t may create a duplicate edge (t, u), which could not be detected early. To deal with such duplicate edges, after receiving a resolved message ⟨resolved, $F_t(\ell), v$⟩, the adjacency list of t is checked to find whether edge (t, v) already exists (Line 20). If the edge does not exist, then it is created. Otherwise, the copy model is re-executed for this edge (Lines 25–34). In case of direct edge (Lines 28–31), a new random k is selected and connected if it does not exist in $\mathbb{F}_t$. In case of copy edge, a new l is selected (Line 33), and a new request message is sent (Line 34). Then the process goes to Line 14 to receive the re-sent message. Note that each PE maintains a counter of the number of messages it sent and yet to receive. The message processing part of the algorithm (Line 14 to Line 34) resumes until all the PEs have received the response messages for all the request messages it sent. We do not show the details of the counters in the algorithm for the sake of simplicity.

3.3 Analysis of Dependency Chains

For the general case $x \geq 1$, each new vertex creates x new edges. Similar to the earlier case, each of these edges forms a selection and a dependency chain. Notice that all of the x selection chains originating from a new vertex are independent of each other, because they independently execute the copy model (irrespective of other outgoing edges from the same vertex) and follow the exact

ALGORITHM 3: Generalized Parallel Algorithm for $x \geq 1$

```
/* Each PE $\mathcal{P}_i$ executes the following in parallel:                */
1  foreach $t \in V_i$ do
2      for $\ell = 1$ to $x$ do
3          $k \leftarrow$ a uniform random vertex in $[0, t-1]$
4          $c \leftarrow$ a uniform random real number in $[0, 1]$
5          if $c < p$ then                              // i.e., with probability $p$
6              if $k \notin \mathbb{F}_t$ then
7                  $F_t(\ell) \leftarrow k$
8              else
9                  go to line 3
10         else
11             $l \leftarrow$ a uniform random number in $[1, x]$
12             $F_t(\ell) \leftarrow$ NULL                 // to be set later to $F_l(k)$
13             send message $\langle$request, $F_t(\ell), F_k(l)\rangle$ to $\mathcal{P}_j$, where $k \in V_j$
/* Next, PE $\mathcal{P}_i$ receives messages sent to it and processes them as follows:   */
14 Upon receipt of message $\langle$request, $F_{t'}(\ell'), F_{k'}(l')\rangle$ from $\mathcal{P}_{j'}$:     // note that $k' \in V_i$
15 if $F_{k'}(l') \neq$ NULL then
16     send message $\langle$resolved, $F_{t'}(\ell'), F_{k'}(l')\rangle$ to $\mathcal{P}_{j'}$
17 else
18     store $\langle F_{t'}(\ell'), F_{k'}(l')\rangle$ in queue $Q_{k'}$
19 Upon receipt of message $\langle$resolved, $F_t(\ell), v\rangle$:
20 if $v \notin \mathbb{F}_t$ then
21     $F_t(\ell) \leftarrow v$
22     foreach $\langle F_{t'}(\ell'), F_t(\ell)\rangle \in Q_t$ do
23         send message $\langle$resolved, $F_{t'}(\ell'), v\rangle$ to $\mathcal{P}_j$ where $t' \in V_j$
24 else                                              // Re-executing copy model
25     $k \leftarrow$ a uniform random vertex in $[0, t-1]$
26     $c \leftarrow$ a uniform random real number in $[0, 1]$
27     if $c < p$ then                                  // i.e., with probability $p$
28         if $k \notin \mathbb{F}_t$ then
29             $F_t(\ell) \leftarrow k$
30         else
31             go to line 25
32     else
33         $l \leftarrow$ a uniform random number in $[1, x]$
34         re-send message $\langle$request, $F_t(\ell), F_k(l)\rangle$ to $\mathcal{P}_j$, where $k \in V_j$
```

same procedures with the same probabilities as shown in Lemmas 2.1 and 2.2. We already showed that the maximum length of a selection chain is at most $6 \log n$ with probability $1 - \frac{1}{n^4}$ in Theorem 2.3. For the general case, there are $O(nx)$ such chains. Using the union bound, the probability that the maximum length is $6 \log n$ for any of the $O(nx)$ selection chains is at least $O(1 - \frac{x}{n^3})$. As

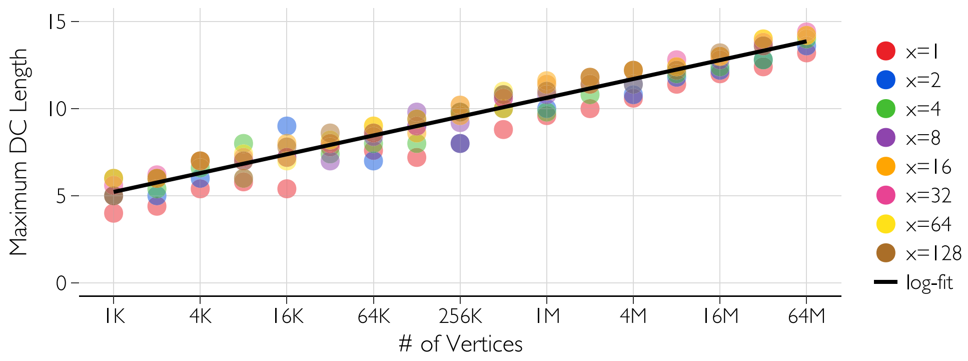


Fig. 3. Experimental result shows that the maximum length of dependency chain is $O(\log n)$. The horizontal axis (in log scale) represents the number of vertices and the vertical axis represents the length of the dependency chain. Filled circles show the maximum length of dependency chain for each pair of n and x. The solid line represents a logarithmic fit of the function $y = a \log n + c$. Here, we used $p = \frac{1}{2}$.

$x \leq n$, we can say that the length of the dependency chain is still $O(\log n)$ w.h.p. Note that this bound holds only when parallel edges are allowed. However, when parallel edges are avoided, for a large graph the effect is negligible as our experimental results show (see Figure 3).

Experimental Validation. We also experimentally evaluated the maximum length of dependency chain using our general algorithm (Algorithm 3). In this experiment, we varied the number of vertices n from $1K$ to $64M$. For each n, we also varied x from 1 to 128. For each possible combination of values of n and x, we calculated the maximum length of dependency chain by repeating the algorithm several times. Figure 3 shows the maximum length of the dependency chain for each combination of n and x. We also plotted a fitted line of the function $y = a \log n + c$ using logarithmic regression. The fitted line has a correlation of 0.97. Therefore, the figure clearly suggests that the maximum length of dependency chain varies logarithmically with n and is independent of x.

3.4 Validating the Degree Distribution

During the execution of copy model, a new vertex t has to select x distinct vertices out of t existing vertices $0, 1, 2, \ldots, t-1$ to make x edges. Let $P_t(i)$ be the probability that vertex i is connected to vertex t. Then,

$$P_t(i) = \begin{cases} \frac{1}{t} + (1-p) \sum_{k=x}^{t-1} \frac{1}{t} P_k(i) & i < \hat{x} \text{ (initial vertices)}, \\ \frac{p}{t} + (1-p) \sum_{k=i+1}^{t-1} \frac{1}{t} P_k(i) & i \geq \hat{x}. \end{cases} \tag{9}$$

Therefore, during the generation of edges from vertex t, vertex i is selected with probability $P_t(i)$. To demonstrate the degree distribution of the expected network from the probability distribution defined in Equation (9), we compute the probabilities numerically for $n = 10{,}000$, $x = 4$, and $p = \{0.01, 0.5, 0.99\}$. The expected degree of a vertex t is given by

$$E[d_i] = \begin{cases} \sum_{k=i+1}^{n-1} P_k(i) & i < \hat{x} \text{ (initial vertices)}, \\ x + \sum_{k=i+1}^{n-1} P_k(i) & i \geq \hat{x}. \end{cases} \tag{10}$$

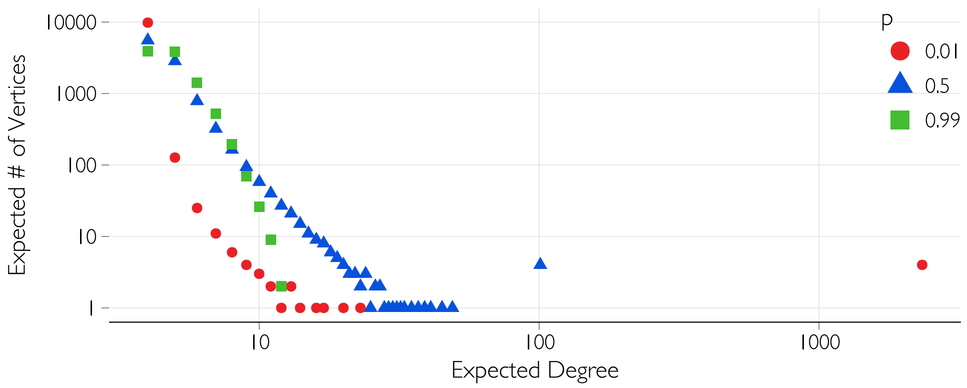


Fig. 4. Expected degree distribution of the algorithm for $n = 10{,}000$, $x = 4$, and $p = \{0.01, 0.5, 0.99\}$. With different values of p, different shapes of the degree distribution are achieved.

Figure 4 shows the expected degree distribution by rounding off the expected degrees of each vertex to its nearest integer value. As demonstrated in the figure, with $p = 0.5$, we have the typical shape of the degree distribution of a Barabási-Albert network. By varying p, we can generate different shapes of degree distributions. In the experimental section, we will show that our implementation produces a similar form of degree distribution for massive generated networks validating the implementation.

3.5 Bounding the Maximum Number of Regeneration of Edges

As noted earlier, with $x > 1$, there is a possibility of making a duplicate edge during the edge creation process. As we are interested in only generating simple graphs with no self-loop or no parallel edges, we avoid the creation of duplicate edges by checking for potential duplicate edge and executing the copy model again if necessary. However, if the number of regeneration of edges is very large, then the algorithm will be inefficient and parallelization will suffer. Fortunately, as we show in this section, the number of regeneration of edges is not very large for most of the practical scenarios and parallelization does not suffer.

Let $\langle u_{i_1}, u_{i_2}, \ldots u_{i_x} \rangle$ denotes the x unique vertices to be picked, i.e., $u_{i_1} \neq u_{i_2} \neq \cdots \neq u_{i_x}$, with the probability distribution shown in Equation (9). We are interested to know how many trials would be required to pick the x unique vertices from t available vertices. It is not difficult to see that the problem is a variation of the famous Coupon Collector's Problem [15, 22], where the probability is not the same for different objects and $x \leq t$ distinct objects have to be picked instead of the whole t objects. Unfortunately, there are no close form results on the expected number of trials required. In [24], the authors presented a formulation of the problem as follows. Let X_k denote the number of trials to get kth unique vertices given that $k - 1$th vertices are unique. Note that $X_1 = 1$, the first vertex, is always unique. X_2 denotes the number of trials required to get a different vertex than u_{i_1}. Therefore, the total number of trials are: $X = X_1 + X_2 + X_3 + \cdots + X_x$. Then, the expected number of trials is given by Reference [24]:

$$\mathbb{E}[X_k] = \sum_{i_1 \neq i_2 \neq \cdots \neq i_{k-1} = 1}^{t-1} \frac{p_{i_1} p_{i_2} \ldots p_{i_{k-1}}}{p(i_1) p(i_1, i_2) p(i_1, i_2, i_3) \ldots p(i_1, i_2, i_3, \ldots, i_{k-1})}, \tag{11}$$

where, $p(i_1, i_2, i_3, \ldots, i_k) = 1 - P_t(u_{i_1}) - P_t(u_{i_2}) - \ldots - P_t(u_{i_k})$. Unfortunately, the formulation is too complex to compute beyond several hundreds vertices.

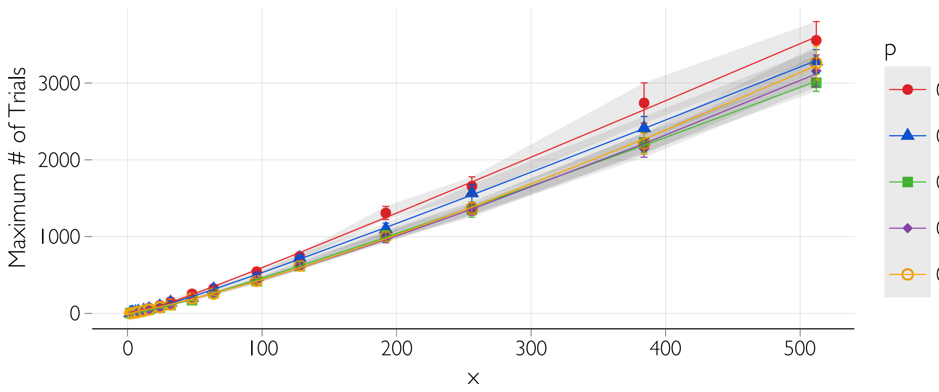


Fig. 5. Experimental results show that the maximum number of required trials is $O(x \log x)$. We used n = 100K.

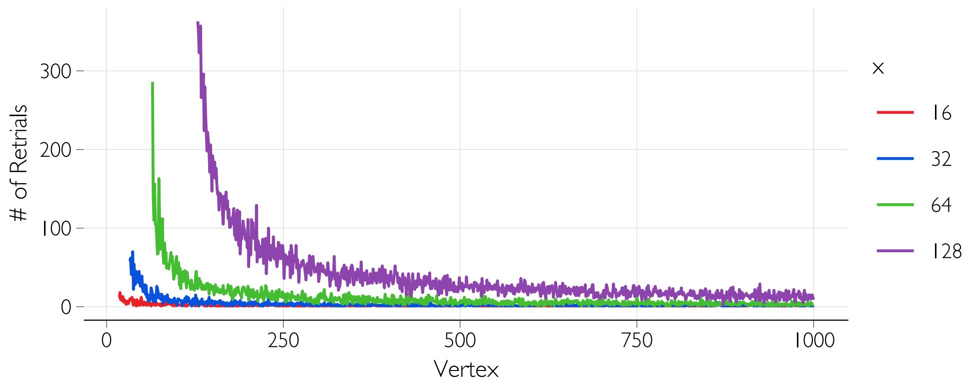


Fig. 6. Experimental results show that the number of required trials reduces as t becomes larger and larger than x. We used n = 100K and $p = \frac{1}{2}$.

Due to the lack of close form solutions and intractable form of exact solution, we instead analyze the maximum number of trials using the copy model itself. First, we analyze the maximum number of trials required per vertex to select x distinct vertices. To do this, we ran the copy model for different x and p values. For each combination of x and p the copy model is executed 15 times. In each of the executions of the copy model, we collected the maximum number of trials needed for a vertex. In Figure 5, we present the average of the maximum number of trials vs. x for different set of p probabilities. We also added error bars and shades denoting 95% confidence intervals. Next, we fitted a line with an equation of the form $x \log x$ that explains 99.94% of the variability. Therefore, we can say that the maximum number of trials required is $O(x \log x)$ for any value of p with at least 95% confidence.

Although the maximum number of trials is $O(x \log x)$, not all vertices require that many trials. In fact, as t becomes larger than x the number of trials required becomes smaller. In Figure 6, we show the number of trials required in each vertex for different values of x with $p = 0.5$. Only the first 1,000 vertices are shown for clarity. As observed from the figure, the number of trials reduces significantly within the first few vertices and becomes very small as $t \gg x$. Therefore, the retrial policy to avoid duplicate edges does not affect the algorithm significantly.

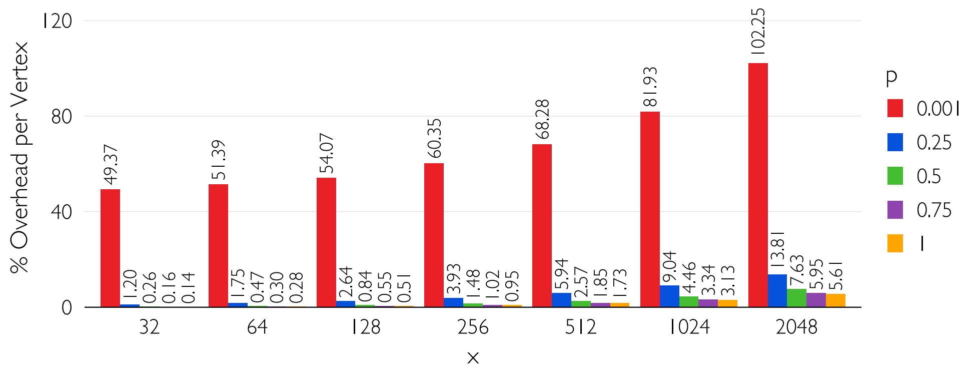


Fig. 7. Experimental results show that the maximum number of retrials is $O(x \log x)$. We used $n = 100$K.

Finally, we demonstrate the overhead incurred for executing copy model with $x > 1$. The overhead is defined as the average number of trials required in excess of x per vertex. For comparing the overhead for different configurations of n, x, and p, we denote the overhead as a percentage over x. For the best cases, the overhead should be close to 0%. In Figure 7, we show the average percentage overhead per vertex for $n = 100$K vertices and different x and p values. As observed from the figure, overhead varies with x and p. The overhead is very big when p is very close to 0. That is because most of the vertices are copy edges and few of the vertices have very high and skewed selection probability compared to other vertices. However, values of p close to 0 are mostly of theoretical interest rather than the more practical scenarios that have large values of p. For almost all practical purposes, the average overhead per vertex is very small.

3.6 Analysis of Waiting Queue Size

In our parallel algorithm, after receiving a request message for an edge $F_k(l)$ (Line 14 of Algorithm 3), a PE sends a corresponding response message immediately if the edge $F_k(l)$ is already known. Otherwise, the request message is stored in a queue called the *waiting queue* (Line 18 of Algorithm 3). If a PE receives a large number of such request messages whose responses could not be sent immediately, then the size of the waiting queues becomes large, leading to a large memory requirement and the parallel algorithm yields poor performance. Fortunately, the number of such request messages is not large. In this section, we provide a rigorous analysis showing that the maximum number of items for the waiting queue of a vertex is $O(x \log n)$ with high probability as shown in Theorem 3.1.

Theorem 3.1. *The maximum number of items to be stored in the waiting queue of a vertex is $O(x \log n)$ with high probability.*

Proof. Assume that the lth outgoing edge of a vertex t executes the copy model and creates an edge with the endpoint of the ℓth edge of a vertex k, i.e., $F_t(l) = F_k(\ell)$ (Copy Edge). A request message $\langle F_t(l), F_k(\ell)\rangle$ is sent to PE $\mathcal{P}_j$ where $k \in V_j$. If $F_k(\ell)$ is not known at the time of receiving the message, then the request will be put on a queue Q_k for vertex k in PE $\mathcal{P}_j$. The queue Q_k is called the *waiting queue* for vertex k. Once $F_k(\ell)$ is known, all the messages in Q_k for that edge will be processed and a corresponding response message will be sent (Line 23 of Algorithm 3).

Therefore, while creating a copy edge $(t, F_k(\ell))$, the request message will be put in the waiting queue Q_k consisting of three events: *(1)* t selects $F_k(\ell)$, *(2)* t chooses to make the copy edge with probability $1 - p$, and *(3)* $F_k(\ell)$ is not known. According to the step 1 of the copy model, t picks $F_k(\ell)$ with probability $\frac{1}{t-1}\frac{1}{x}$. Furthermore, $F_k(\ell)$ is already known with probability at least p (Direct

Edge). Therefore, $F_k(\ell)$ is not known with probability at most $1-p$. Let $P_t(k_\ell)$ denote the probability that any outgoing edge from vertex t makes a copy edge $(t, F_k(\ell))$ and the corresponding request message is put in the waiting queue Q_k. Therefore, we have

$$\begin{aligned} P_t(k_\ell) &= \Pr\left[F_k(\ell) \text{ is selected}\right] \times \Pr\left[\text{copy edge is created}\right] \times \Pr\left[F_k(\ell) \text{ is not known}\right] \\ &\le \frac{1}{t-1}\frac{1}{x}(1-p)(1-p) \\ &\le (1-p)^2\frac{1}{x(t-1)}. \end{aligned} \tag{12}$$

Let $X_t(k_\ell)$ be an indicator random variable such that $X_t(k_\ell) = 1$ if a request message from vertex t is stored in Q_k for the copy edge $F_k(\ell)$, and $X_t(k_\ell) = 0$ otherwise. Vertex t creates x edges independently and each of these edges stores a request message in Q_k for the edge $F_k(\ell)$ with probability $\Pr[X_t(k_\ell) = 1] = P_t(k_\ell)$. Therefore, we have

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

Let Y_k be a random variable that denotes the total number of messages stored in Q_k. According to the parallel algorithm, Q_k can store messages from vertex $k+1$ to $n-1$. Each of the vertex creates x edges independently. Thus, by the definition of Y_k, we have

$$Y_k = \sum_{t=k+1}^{n-1}\sum_{\ell=1}^{x} X_t(k_\ell).$$

Therefore, Y_k is simply a sum of independent Bernoulli random variables. The expected number of request messages stored in the queue Q_k is given by

$$\begin{aligned} E\left[Y_k\right] &= \sum_{t=k+1}^{n-1}\sum_{\ell=1}^{x} E\left[X_t(k_\ell)\right] \\ &\le \sum_{t=k+1}^{n-1}\sum_{\ell=1}^{x}(1-p)^2\frac{1}{(t-1)} \\ &\le (1-p)^2 x \sum_{t=k+1}^{n-1}\frac{1}{t-1} \\ &\le (1-p)^2 x \sum_{i=k}^{n-2}\frac{1}{i} \\ &\le (1-p)^2 x\left(H_{n-2} - H_{k-1}\right) \\ &\le (1-p)^2 x H_{n-2} \\ &\le (1-p)^2 x \log n. \end{aligned} \tag{13}$$

Applying the Chernoff bound on independent Bernoulli random variables, we have

$$\Pr\left[\sum_{t=k+1}^{n-1}\sum_{\ell=1}^{x} X_t(k_\ell) \ge (1+\delta)\mu\right] \le \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu.$$

In the Chernoff bound, we set $\delta = \frac{5x\log n}{\mu} - 1$. In this case, $\mu \le (1-p)^2 x\log n$, where $0 \le p \le 1$. Note that when $p = 1$ no copy edge will be created, therefore, no item will be placed in the waiting

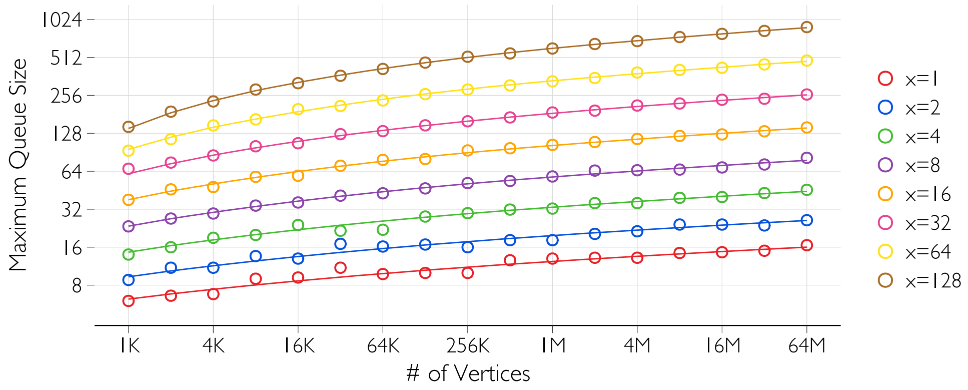


Fig. 8. The maximum size of the waiting queues changes logarithmically with n.

queue and the maximum number of items in Q_k is 0. For $p < 1$, we have $\delta > 0$. Then,

$$
\begin{aligned}
\Pr\left[Y_k \geq 5x \log n\right] &= \Pr\left[\sum_{t=k+1}^{n-1} \sum_{\ell=1}^{x} X_t(k_\ell) \geq (1+\delta)\mu\right] \\
&\leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \leq \left(\frac{e}{1+\delta}\right)^{\mu(1+\delta)} \\
&\leq \left(\frac{e\mu}{5x\log n}\right)^{5x\log n} \leq \left(\frac{e(1-p)^2 x \log n}{5x \log n}\right)^{\log n^{5x}} \\
&\leq \frac{1}{\left(\frac{5x}{e}\right)^{\log n^{5x}}} && (1-p) < 1 \\
&\leq \frac{1}{n^3} && [x \geq 1].
\end{aligned}
$$

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

3.7 Experimental Validation of Waiting Queue Size

In this section, we experimentally evaluate how the maximum size of the waiting queues varies with n, p, and x as shown in Theorem 3.1.

In Figure 8, we plot the maximum size of the waiting queues by varying n for a set of different x. We set $p = \frac{1}{2}$ in these experiments. In the figure, the circles represent the maximum size of the waiting queues collected experimentally, and the solid lines present a fit function $y = a \log n + c$ for different values of x. Both axes are plotted in log scale. The figure demonstrates that the maximum size of a waiting queue is proportional to $\log n$.

In Figure 9, we plot the maximum size of the waiting queues by varying x for a set of different n. We also set $p = \frac{1}{2}$ in these experiments. In the figure, the circles represent the maximum waiting queue size collected experimentally, and the solid lines present a linear fit function $y = ax + c$ for different values of n. The figure demonstrates that the maximum size of a waiting queue is proportional to x.

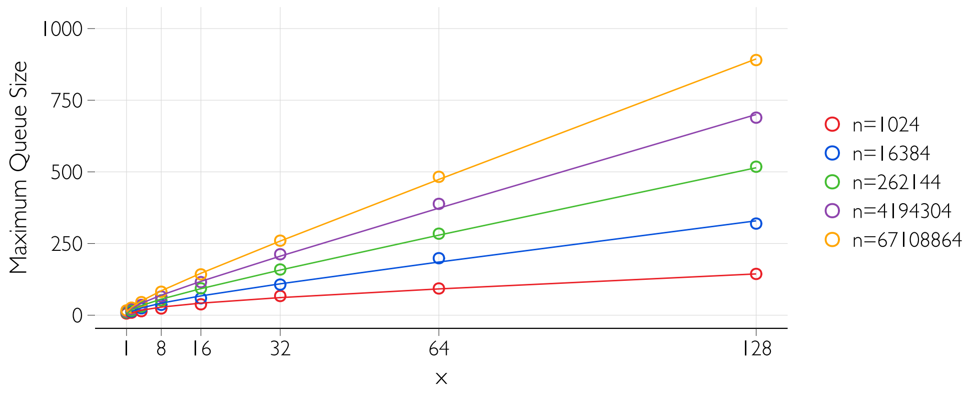


Fig. 9. The maximum size of the waiting queues changes linearly with x.

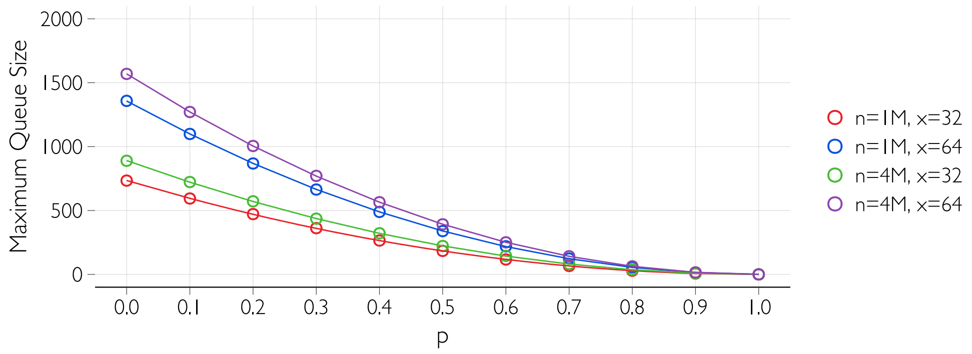


Fig. 10. The maximum size of the waiting queues changes with $(1-p)^2$.

In Figure 10, we plot the maximum size of the waiting queues by varying p for a set of different n and x. In the figure, the circles represent the maximum waiting queue size collected experimentally, and the solid lines present a quadratic fit function $y = a(1-p)^2 + c$ for different values of n and x. The figure demonstrates that the maximum size of a waiting queue is proportional to $(1-p)^2$.

4 PARTITIONING AND LOAD BALANCING OF PARALLEL EXECUTION

Recall the formal definition of partitioning of the set of vertices $V = \{0, 1, \ldots, n-1\}$ into P subsets $V_0, V_1, \ldots, V_{P-1}$ as described at the beginning of Section 2. A good load balancing is achieved by properly partitioning the set of vertices V and assigning each subset to one PE. Vertex partitioning has significant effects on the performance of the algorithm. In this section, we study several partitioning schemes and their effects on load balancing and the performance of the algorithm. In our algorithm, we measure the computational load in terms of the number of vertices per PE, the number of outgoing messages (request message) from a PE, and the number of incoming messages (response messages) to a PE.

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

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

- The partitioning should lead to a good load balancing. The degrees of the vertices vary significantly, and a vertex with a larger degree causes more messages to work with. As a result, naïve partitioning may lead to poor load balancing.
- As we discuss later, combining multiple messages (to the same destination) and using an `MPI_send` operation for them can increase the efficiency of the algorithm. However, combining multiple messages may not be possible with an arbitrary partitioning as it may cause deadlocks.

With the objective of satisfying the above criteria, we study the three partition schemes:

(1) Consecutive Partitioning,
(2) Round-robin Partitioning,
(3) Segmented Partitioning.

4.1 Consecutive Partitioning

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

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

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

A. generating random numbers and some other processing for each vertex $t \in V_i$,
B. sending request messages for the vertices in V_i and receiving their replies, and
C. receiving request messages from other PEs and sending their replies.

The computational load for load type A and B above is directly proportional to the number of vertices in partition V_i. Computational load for load type C depends not only on the number of vertices in a PE but also on i, the rank of the PE. With simple consecutive vertex partitioning (SCP), a lower ranked PE receives more request messages than a higher ranked PE, because with $j < k$, we have $E[M_j] > E[M_k]$, where M_k is the number of request messages received for vertex k (see Lemma 4.1).

Lemma 4.1. *Let M_k be the number of request messages received for vertex k. Then $E[M_k] = (1 - p)(H_{n-1} - H_k)$, where H_k is the kth harmonic number.*

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

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

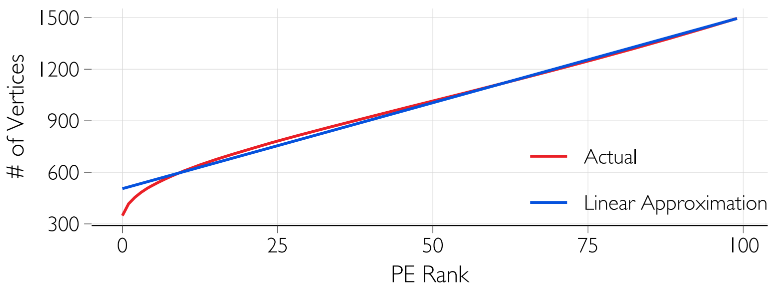


Fig. 11. Distribution of the vertices among PEs for actual solutions of Equation (15) and its linear approximation.

□

Next, we calculate the computational load for each PE with an arbitrary number of vertices assigned to the PEs. To do so, we make the following simplifying assumptions: (i) Sending a message takes the same computation time as receiving a message, and (ii) $p = \frac{1}{2}$ (the same analysis will follow for arbitrary p by simply multiplying each term with $2(1-p)$). The number of vertices in PE $\mathcal{P}_i$ is $n_{i+1} - n_i$. Then computation cost for load of type A and B is $c(n_{i+1} - n_i)$ for some constant c. Following Lemma 4.1, the expected load for type C in PE $\mathcal{P}_i$ is

$$\begin{aligned}\sum_{k=n_i}^{n_{i+1}-1} (H_{n-1} - H_k) &= (n_{i+1} - n_i)H_{n-1} - \sum_{k=n_i}^{n_{i+1}-1} (H_k) \\ &= (n_{i+1} - n_i)H_{n-1} - (n_{i+1}H_{n_{i+1}} - n_iH_{n_i}) + (n_{i+1} - n_i) \\ &= (n_{i+1} - n_i)(H_{n-1} + 1) - (n_{i+1}H_{n_{i+1}} - n_iH_{n_i}). \end{aligned} \tag{14}$$

The second to the last line follows from Equation (2.36) on page 41 of Reference [28]. Thus, using another constant $b = 1 + c$, the total computational load at PE $\mathcal{P}_i$ is

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

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

$$\begin{aligned} n_0 &= 0, \\ n_P &= n - 1, \\ c(P_i) = (n_{i+1} - n_i)(H_{n-1} + b) - (n_{i+1}H_{n_{i+1}} - n_iH_{n_i}) &= \frac{c'n}{P}. \end{aligned} \tag{15}$$

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

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

To overcome these difficulties, guided by experimental results, we approximate the solution of the above system of equations with a linear function and call the resultant partitioning scheme *linear consecutive partitioning* (LCP). Figure 11 shows the distribution of the vertices among PEs for actual solutions of Equation (15) and linear approximation. As we will see later in Section 5, our approximate scheme LCP provides a very good load balancing and performance of the algorithm.

As in the LCP scheme, the number of vertices is increasing linearly with i (the ranks of the PEs), the number of vertices in PE $\mathcal{P}_i$ follows the arithmetic progression $a, a + d, a + 2d, \ldots, a + (P - 1)d$, that is, the number of vertices in PE $\mathcal{P}_i$ is $B_i = a + id$, where d is the slope of the line for linear approximation as shown in Figure 11. Slope d can be approximated easily by sampling two points on the actual line. Partition V_i has the vertices from $\sum_{j=0}^{i-1}(a + jd) = i\frac{(2a+(i-1)d)}{2}$ to $\sum_{j=0}^{i}(a + jd) - 1 = (i + 1)\frac{(2a+id)}{2} - 1$. Finding the rank of the PE for vertex u is more complicated in this scheme. Given a vertex u, we need to find the PE $\mathcal{P}_i$ such that $u \in V_i$. Vertex u satisfies the following inequality:

$$\sum_{j=0}^{i-1}(a + jd) \leq u < \sum_{j=0}^{i}(a + jd),$$
$$\frac{i\,(2a + (i - 1)d)}{2} \leq u < \frac{(i + 1)\,(2a + id)}{2}. \tag{16}$$

Solving the inequality Equation (16), we have

$$i = \left\lfloor \frac{-(2a - d) + \sqrt{(2a - d)^2 + 8du}}{2d} \right\rfloor. \tag{17}$$

Determining partition parameters *a* and *d*. The parameters a and d are determined using the number of vertices n and the number of PEs P. Parameter d is the slope of the straight line $y = a + dx$, where y represents the number of vertices in the PE with rank $x = i$. We calculate d by finding two points on this straight line. Putting $i = 0$ and $i = P - 1$ in Equation (15), we can compute n_1 and n_{P-1}. Then, the number of vertices in the first PE is $n_1 - n_0 = n_1$ and the number of vertices in the last PE is $n_P - n_{P-1} = n - 1 - n_{P-1}$. Hence, we have

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

Now, we have

$$\sum_{j=0}^{P-1}(a + jd) = n,$$
$$\frac{P\,(2a + (P - 1)d)}{2} = n,$$
$$a = \frac{n}{P} - \frac{(P - 1)d}{2}. \tag{18}$$

Message Buffering. The PEs exchange two types of messages: request messages and resolve messages. For each vertex t, a PE may need to send one request message and receive one resolve message. If PE $\mathcal{P}_i$ has multiple messages destined to the same PE, say PE $\mathcal{P}_j$, then PE $\mathcal{P}_i$ can combine them into a single message by buffering them instead of sending them individually. Each PE can do so by maintaining $P - 1$ buffers, one for each of the other PEs. If the messages are not combined, then for large n there can be a large number of outstanding messages in the system, and the system may not be able to deal with such a large number of messages at a time, limiting our ability to generate a large network. Further message buffering reduces overhead of packet headers and thus improves efficiency.

4.2 Round-robin Partitioning (RRP)

In this scheme, vertices are distributed in a round-robin fashion among all PEs. Partition V_i contains the vertices $\langle i, i + p, i + 2p, \ldots, i + kp\rangle$ such that $i + kp \leq n < i + (k + 1)p$; that is, $V_i = \{j|j$

mod $P = i\}$. In other words, vertex i is assigned to set $V_{i \bmod p}$. Similar to SCP, in this RRP scheme the number of vertices in the sets is almost equal. The number of vertices in a set is either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$. The difference between the number of vertices in two sets is at most 1.

From Lemma 4.1, it is clear that the expected number of received messages decreases monotonically with increasing vertex labels. Round-robin partitioning on such a monotonic distribution typically performs better. For the round-robin vertex partitioning scheme, the computational load among PEs is well-balanced as shown in Lemma 4.2.

Lemma 4.2. *The difference between the computational load for any two PEs is at most $O(\log n)$, while the total computational load is $\Omega(n)$.*

Proof. The expected number of request messages received for vertex k is $(H_{n-1} - H_k)$ (see Lemma 4.1). Other loads for any vertex are constant. Then the total load for vertex k is $CL(k) = (H_{n-1} - H_k) + b$, for some constant b. Thus, the total load for PE $\mathcal{P}_i$ with partition $V_i = \{j | j \bmod P = i\}$ is $PL(i) = \sum_{k \in V_i} (H_{n-1} - H_k + b)$.

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

$$PL(0) - PL(P-1) = \sum_{k \in V_0} (H_{n-1} - H_k + b) - \sum_{k \in V_{P-1}} (H_{n-1} - H_k + b) \tag{19}$$

$$\leq (H_{n-1} + b)(|V_0| - |V_{P-1}|) - \sum_{k \in V_0} H_k + \sum_{k \in V_{P-1}} H_k. \tag{20}$$

If n is a multiple of P, then we have

$$|V_0| - |V_{P-1}| = 0, \tag{21}$$

$$\sum_{k \in V_{P-1}} H_k < \sum_{k \in V_0} H_k + H_n, \tag{22}$$

$$\text{and thus, } PL(0) - PL(P-1) < H_n = O(\log n). \tag{23}$$

Otherwise,

$$|V_0| - |V_{P-1}| = 1, \tag{24}$$

$$\sum_{k \in V_{P-1}} H_k \leq \sum_{k \in V_0} H_k, \tag{25}$$

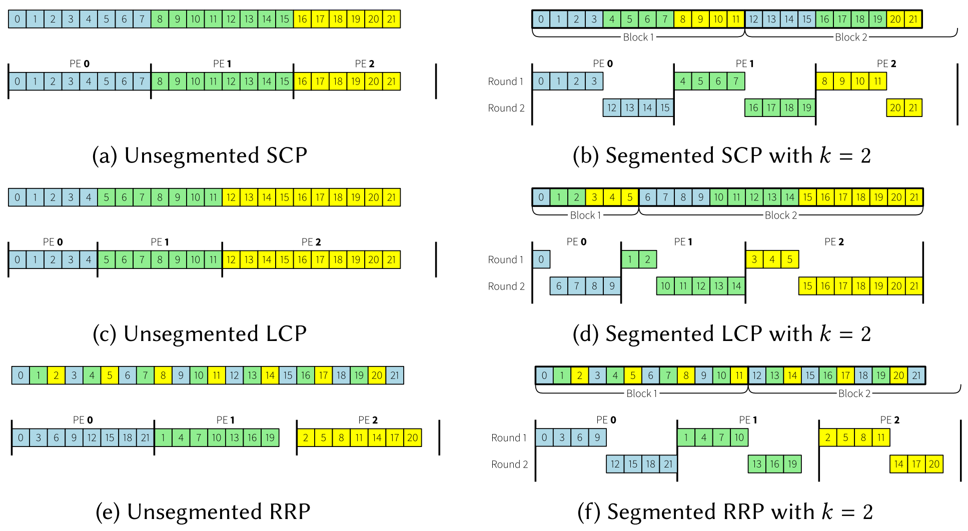
$$\text{and thus, } PL(0) - PL(P-1) \leq H_{n-1} + b = O(\log n). \tag{26}$$

□

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

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

However, in the RRP scheme, deadlock can occur if the messages are not buffered carefully. The request messages can be buffered as it is done in SCP or LCP. The resolved message can also be buffered, but it needs to be done in a special way to avoid deadlock. To avoid deadlock, resolved



(a) Unsegmented SCP (b) Segmented SCP with $k = 2$

(c) Unsegmented LCP (d) Segmented LCP with $k = 2$

(e) Unsegmented RRP (f) Segmented RRP with $k = 2$

Fig. 12. Segmented partitioning with $P = 3$ PEs.

messages must be sent out from the buffer (even if the buffer is not full yet) after processing every group of received messages (when buffering is used, messages are sent and received in groups). Sending the resolved messages cannot wait any longer. Otherwise, it can cause circular waiting among the PEs leading to a deadlock situation.

4.3 Segmented Partitioning

So far, we have studied partitioning schemes where the entire set of vertices are partitioned into P subsets and each PE works on a partition. In this section, we present another fine-grained partitioning technique called *Segmented Partitioning*.

In the segmented partitioning technique, first the entire set of vertices is partitioned into k consecutive subsets $S_1, S_2, S_3, \ldots, S_k$ called *segments* (similar to the consecutive partitioning). From the copy model definition, clearly vertices on a segment S_i may only depend on vertices on segment S_j where $i \geq j$ but not vice versa. Let $B_i = |S_i|$ denote the number of elements (also called the segment size) in segment S_i where $1 \leq i \leq k$. Next, the parallel algorithm is executed in k rounds where round i executes the parallel algorithm for all the vertices in segment S_i. In round i, the B_i vertices in segment S_i are further partitioned into P subsets $V_0(S_i), V_1(S_i), \ldots V_{P-1}(S_i)$ (using the previous schemes) and executed in parallel using the P PEs. After a round is completed, every edge originating from the vertices in the segment is completely determined. We used segmented partitioning technique for SCP, LCP, and RRP schemes. The technique is illustrated in Figure 12.

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

However, as segment size keeps increasing beyond some point, we start losing any advantages because of the synchronization issues that need to be performed in each round. Therefore, there is an optimal value of k. We experimentally varied k to determine the optimal value for each partitioning scheme.

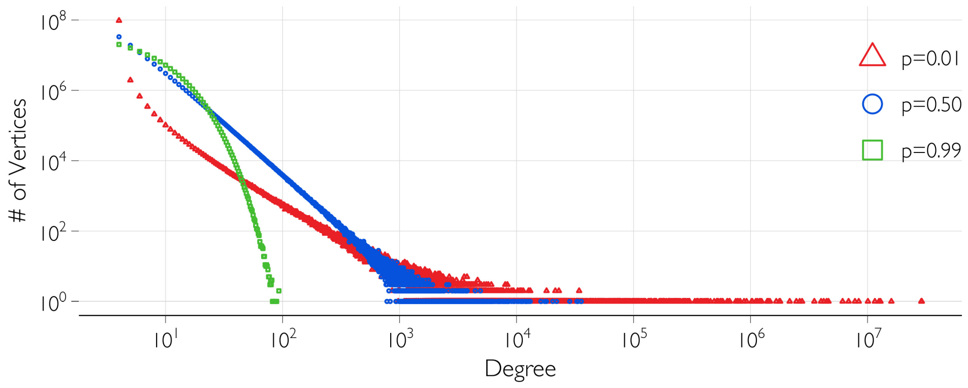


Fig. 13. The degree distribution (in log - log scale) of the network generated by our parallel algorithms. The network is generated with $n = 10^9$ and $x = 4$.

5 EXPERIMENTAL RESULTS

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

Experimental Setup. We used a high-performance computing cluster of 4,600 IBM Power System AC922 nodes. Each node consists of two IBM POWER9 processors and 512 GB of DDR4 memory. The POWER9 processor is built around IBM's SIMD Multi-core (SMC) having 22 SMCs per processor. The nodes are connected to a dual-rail EDR InfiniBand network in a Non-blocking Fat Tree topology. For the MPI-based implementation of our algorithms, we used IBM Spectrum MPI from IBM XL Compilers.

In the experimental evaluation, we used each core of the IBM POWER9 processor (SMC) as a PE and used up to 1,000 PEs. Each of the algorithms we considered generates the network in main memory, and the run time does not include the time required to write the graph to disk.

5.1 Validating Scale-free Property with Degree Distribution

The degree distribution of the graph generated by our parallel algorithm is shown in Figure 13 in a log - log scale. We used $n = 1$B vertices and $x = 4$ that generates a network with 4B edges. As shown in the figure, the copy model produces power-law degree distributions for various values of p. When $p = 0.5$, the degree distribution is the same as the BA model. As the figure shows, the distribution is heavy tailed, which is a distinct feature of the real-world power-law networks. The exponent γ of this power-law degree distribution is measured to be 2.7, which supports the fact that for a finite average degree of a scale-free network, the exponent γ satisfies $2 < \gamma < \infty$ [19]. When p is very close to 0, the network is mainly built on copy edges, therefore, there is a higher level of bias toward the higher degree vertices as evident from the longer tail. However, when p is close to 1, the network mainly consists of direct edges, and we do not see long tails, a salient property of many real world networks. The above results show that copy model is more general and capable of generating many interesting degree distributions. Further, it shows that our algorithms produce scale-free networks very accurately.

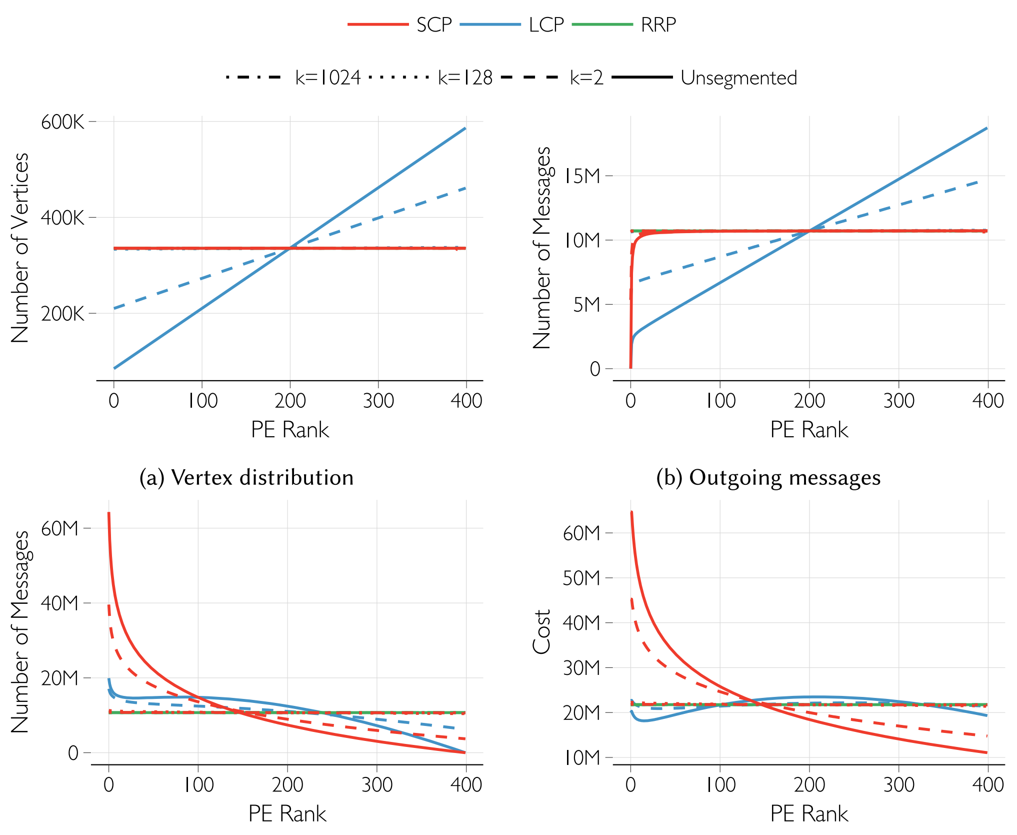


Fig. 14. Vertex and message distribution for the partitioning schemes ($n = 2^{27}$, $x = 64$, $p = 0.5$). Note that part of the lines for RRP and SCP are overlapping in the figures.

5.2 Performance of Partitioning Schemes and Segment Sizes

Vertex partitioning has significant effects on load balancing and performance of the algorithm. In Section 4, we discussed three partitioning schemes SCP, LCP, and RRP, and theoretically analyzed them. In this section, we experimentally study these schemes and their effect on the performance of the algorithm. In these experiments, we used $n = 2^{27}$ vertices, $x = 64$ edges per vertex, $p = 0.5$, and 400 PEs, which are sufficient to demonstrate the behavior and differences of the partitioning schemes. For each of the three schemes, we measure the computational load in the PEs by the number of vertices per PE, the number of outgoing messages from the PEs, and the number of incoming messages to the PEs. The results are shown in Figure 14.

Vertex Distribution. The vertex distribution is shown in Figure 14(a). For SCP and RRP, vertices are distributed uniformly among the PEs, and each PE has about 336K vertices. For LCP, the number of vertices in the PEs are increasing linearly with the rank of the PEs.

Message Distribution. In a consecutive partitioning (SCP and LCP), PE $\mathcal{P}_i$ sends outgoing request messages to PEs $\mathcal{P}_0$ to $\mathcal{P}_{i-1}$ and receives incoming messages from PEs $\mathcal{P}_{i+1}$ to $\mathcal{P}_{P-1}$. For each vertex, a PE sends a request message with probability at most $1 - p$ (see Equation (2)). Thus, the expected number of request messages sent by a PE is proportional to the number of vertices in the PE, as shown in Figure 14(b). Note that in the SCP and LCP schemes, PE $\mathcal{P}_0$ does not need to send any request messages at all.

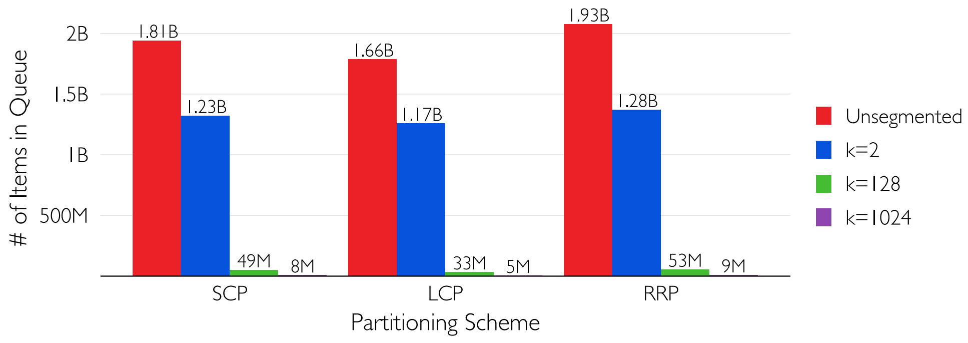


Fig. 15. Size of waiting queue for different segment sizes ($n = 2^{27}$, $x = 64$, $p = 0.5$).

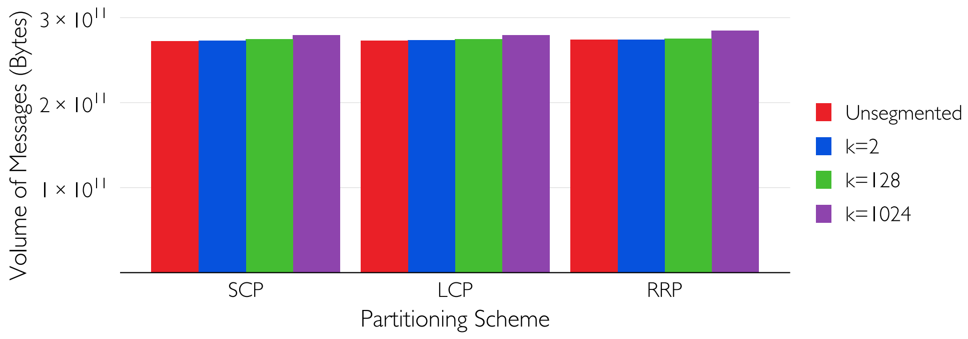


Fig. 16. Total amount of messages (in bytes) communicated among PEs ($n = 2^{27}$, $x = 64$, $p = 0.5$).

Figure 14(c) shows the number of incoming request messages for each PE. It is clear that a lower ranked PE receives more messages than a higher ranked PE in consecutive partitioning (SCP and LCP) as suggested by Lemma 4.1. In the RRP scheme, both incoming and outgoing messages are evenly distributed among the PEs.

Total Load Distribution. Besides sending and receiving messages, for each vertex, a PE can incur a constant other computational cost. Thus, for analysis purposes, we measure the total computational load of a PE as the sum of the number of vertices in the PE and the number of incoming and outgoing messages. Figure 14(d) shows the total load for the three partitioning schemes. The RRP scheme distributes the load almost perfectly among the PEs. Load balancing in the LCP scheme is also quite good. However, the SCP scheme distributes the load very poorly. These experimental results verify our theoretical analysis given in Section 4.

Size of the Waiting Queue. With the segmented partitioning scheme, the total size of the waiting queues is reduced with increasing segment size as shown in Figure 15. Therefore, segmented partitions yield better performance in our algorithm.

Message Volume. Figure 16 shows the total volume of the messages to be exchanged among the PEs for different partitioning schemes. Note that in the copy model, the expected number of copy edges is $(1 - p)$ fractions of all the edges and the PEs need message exchange to resolve copy edges that do not reside in the same PE. Therefore, the total volume of the messages does not differ by much in the different configurations of partitioning schemes as shown in the figure.

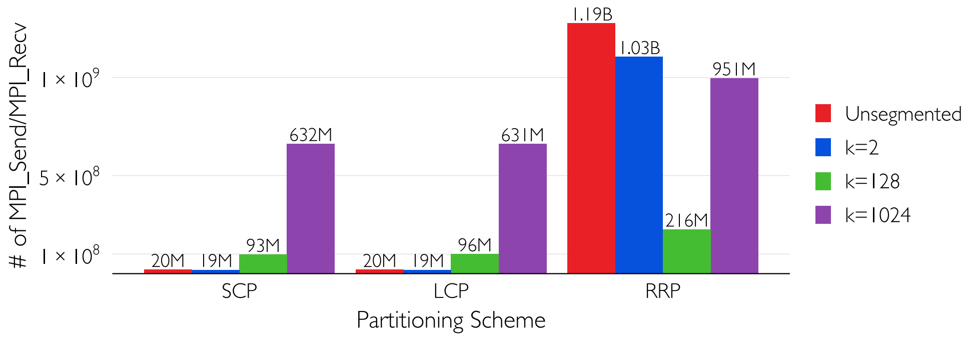


Fig. 17. Total number of `MPI_Send` and `MPI_Recv` calls ($n = 2^{27}$, $x = 64$, $p = 0.5$).

Number of MPI Messages. Figure 17 shows the total number of `MPI_Send` and `MPI_Recv` calls among different PEs for different partitioning schemes. The number of MPI calls depends on message buffering as discussed in Section 4. In this experiment, we used a message buffer of size 1,024. Note that although the message volume is at the similar level for all of the algorithms (Figure 16) consecutive partitions (SCP and LCP) tend to send a smaller number of MPI messages. Although it might indicate a more efficient message transfer, however, in reality, it exhibits less frequent sync among the PEs leading to increased idle time. Also note that in RRP, to avoid deadlock scenario, we sent out response messages as soon as a batch of request messages is received and processed. For this reason, RRP exhibits higher number of MPI message compared to the consecutive schemes.

As the number of stages is increased for each partitioning scheme, the number of MPI messages tend to decrease before increasing as most evident in the RRP scheme. This can be attributed to the flushing of message buffers and dependency chains. In the RRP scheme, we need to flush message buffers for each batch of messages a PE processes to avoid circular waiting. When the number of segments is smaller, the segment size and the lengths of dependency chains in each segment are larger. It requires more flushing of message buffers due to dependency chains and hence the number of MPI messages becomes large. However, if the number of segments is larger, then the segment size and the length of dependency chains are smaller. In this case, the number of MPI messages is dominated by the number of segments. This suggests that there might be an optimal segment size that reduces the MPI message communication. Next, we experimentally evaluate the best segment size that works in practice.

Optimal Segment Size. Although an increasing segment size reduces the size of waiting queue, it also reduces concurrency. Therefore, if the segment size is increased beyond some limit, then the performance would start to decrease. To figure out the best segment size, we ran a wide set of experiments varying the number of vertices ($2^{26}, 2^{27}, 2^{28}$), number of edges per vertex (16, 32, 64, 128), PEs (100, 200, 400, 800), and different segment sizes for a fixed $p = \frac{1}{2}$. We measure the performance of each experimental run by the million edges produced per second per PE. The performance results are presented in Figure 18. Each straight line connecting the points represents the same configuration of n, x, and P but varying segment size. Our experimental results suggest that the optimal segment size S can be calculated as $S = C_S \times \frac{nx}{P \log P}$ where C_S is a constant that depends on the processing capabilities of the computing cluster. In our cluster, the value of $C_S = \frac{1}{12{,}000}$. As shown in the figure, the best performance peaks among all the experimental configuration runs tend to maximize around the calculated step size. Our further experiments suggest that the value of S works for most of the other cases quite well.

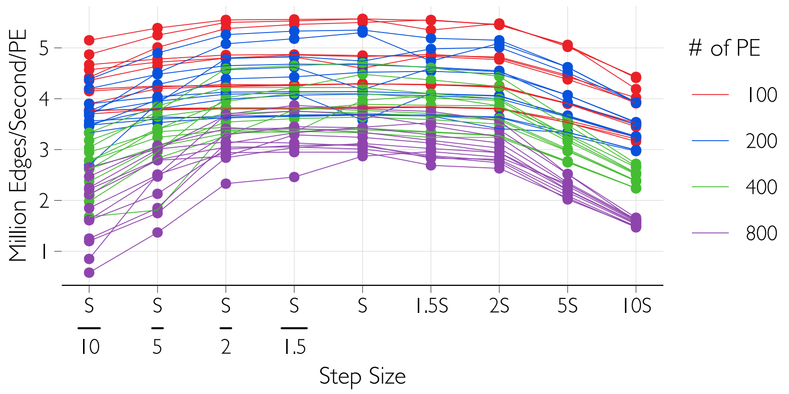


Fig. 18. Performance of our algorithms for different segment sizes.

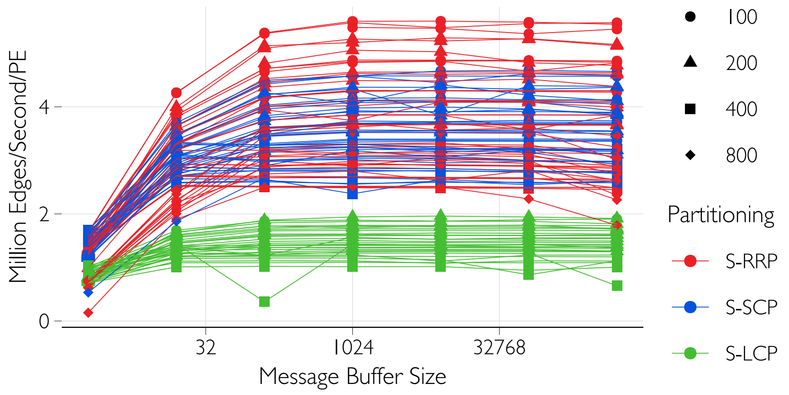


Fig. 19. Performance of our algorithms for different message buffer sizes.

Optimal Message Buffer Size. Next, we experimentally evaluate the best message buffer size. To figure out the best message buffer size, we ran a wide set of experiments varying the number of vertices (2^{26}, 2^{27}, 2^{28}), number of edges per vertex (16, 32, 64, 128), PEs (100, 200, 400, 800), segment size of $S = C_S \times \frac{nx}{P \log P}$, and message buffer sizes (2, 16, 128, 1,024, 8,192, 65,536, 524,288) for a fixed $p = \frac{1}{2}$. We also measure the performance of each experimental run by the million edges produced per second per PE. The performance results are presented in Figure 19, which shows that as the buffer size is increased the performance increases. However, beyond message buffer size of 1,024, there is no discernible gain observed. Therefore, a message buffer size of 1,024 is being used in the production large case runs.

Effect of p on Performance. If p is reduced, then most of the edges produced consists of copy edges, therefore requiring more message exchanges. As p is increased toward 1, most edges consist of direct edges. Therefore communication is reduced. This is shown in Figure 20.

5.3 Parallel Execution and Scalability

Strong Scaling. Strong scaling of a parallel algorithm shows its performance with an increasing number of PEs keeping the problem size fixed. Figure 21 shows speedup factors of our algorithms with segmented and unsegmented techniques using SCP, linear consecutive LCP, and RRP partitioning schemes, as the number of PEs increases with problem size $n = 100M$ and $x = 60$. Speedup

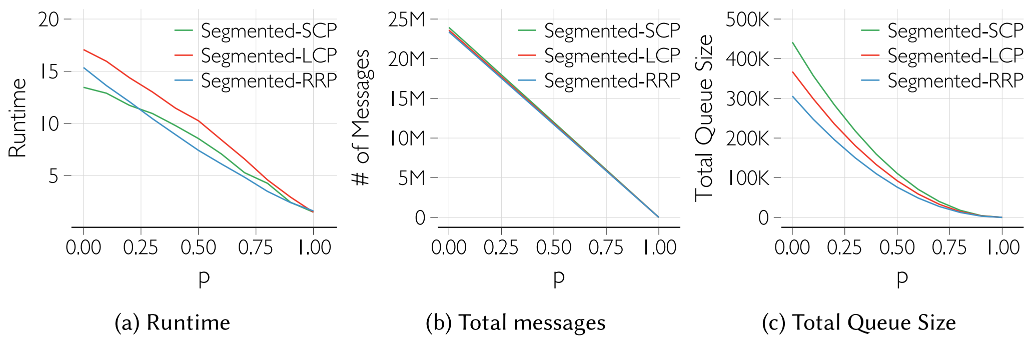


Fig. 20. Performance of segmented partitioning on p.

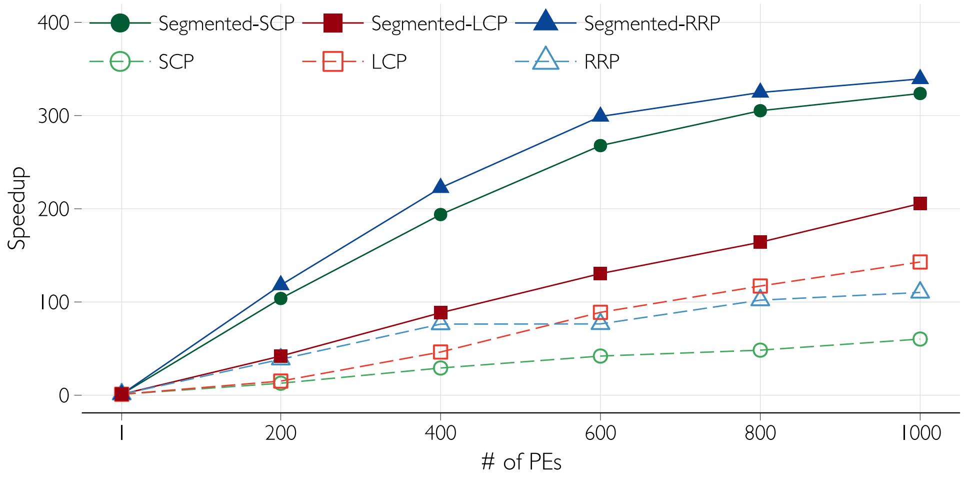


Fig. 21. The strong scaling of our parallel algorithms for the problem size n = 100M and x = 60.

factors are measured as T_s/T_p, where T_s and T_p are the running time of a sequential algorithm and the parallel algorithm, respectively. We have implemented the sequential version of our algorithm in C++. This sequential implementation outperforms the best available implementation of the BA model given in the NetworkX graph algorithm library [29]. As the sequential algorithm cannot generate more than 6B edges due to memory limitations, we choose n = 100M and x = 60. We varied the number of PEs from 1 to 1,024 for this experiment.

Parallelization of network algorithms is notoriously hard. Furthermore, we have observed that the problem of generating a scale-free random network is quite sequential in nature due to the dependencies among the edges. As Figure 21 shows, the speedups of our algorithms are increasing almost linearly with the number of PEs. Given the sequential nature of the problem, our algorithms show very good speedup. Further, the speedup of segmented versions performs better. Note that both Segmented-SCP and Segmented-RRP are performing the best, due to better load balancing and reduced queue size.

Weak Scaling. Weak scaling measures the performance of a parallel algorithm when the input size per PE remains constant. For this experiment, we varied the number of PEs from 1 to 1,000. With the number of PEs, the input size is also increased proportionally: for P PEs, a network with

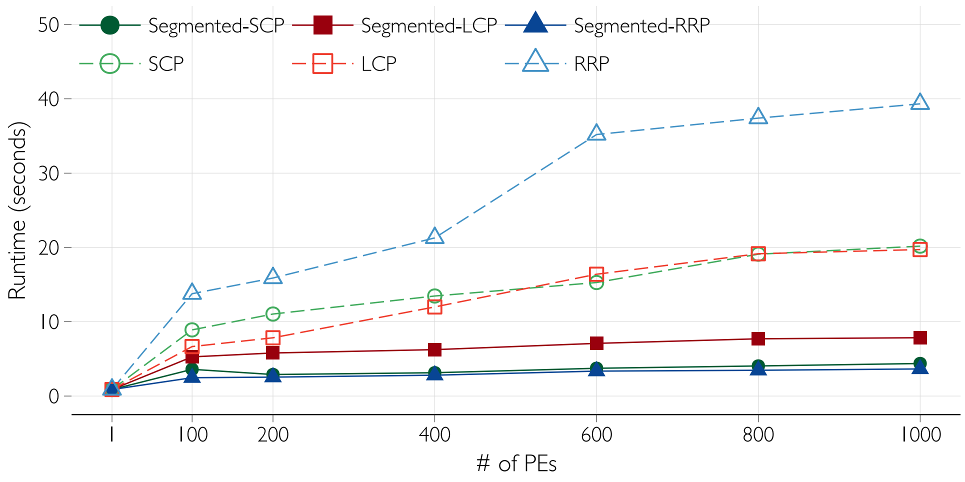


Fig. 22. Weak scaling of our parallel PA algorithm.

10^7P edges is generated. Figure 22 shows the weak scaling of our algorithms with the increasing number of PEs.

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

Generating Large Networks. Our main goal for designing this algorithm is to generate very large random networks. Using our algorithm with the Segmented RRP scheme, we are able to generate a network with one trillion (2^{40}) edges, with $n = 2^{34}$ and $x = 64$. Using 1,000 PEs, the generation of the network takes 84.79 seconds with duplicate edges and 324.60 seconds without any duplicate edges.

Comparison with KaGen. We also compare the performance of our algorithm to the best available distributed memory algorithm to generate Barabási–Albert networks named KaGen [26, 43]. KaGen is available as an open source library from https://github.com/sebalamm/KaGen. We compiled KaGen in our same computing cluster with the most optimization level (`-O3`) of the `C++` compiler. KaGen generates graphs with duplicate edges. To generate the same graph with one trillion edges ($n = 2^{34}$ and $x = 64$), KaGen took 73.83 seconds using 1,000 PEs in our cluster. In comparison, our algorithm with duplicate edges take 84.79 seconds in the same cluster. However, our algorithm is more general and can produce a more general class of power–law graphs. Note that the BA generator of KaGen is reported to generate 10^{15} edges within 3,600 seconds using 16,384 [43].

6 RELATED WORK

Although the concepts of random networks have been used and well studied over the last several decades, efficient algorithms to generate the networks were not available until recently. The first efficient sequential algorithm to generate Erdős-Rényi and Barabási-Albert networks was proposed in Reference [14]. A distributed memory-based algorithm to generate preferential attachment networks was proposed in Reference [46]. However, their algorithm was not exact, rather an

Table 2. Runtime Performance Recent Power-law Network Generators

Implementation	Edges	Generator Model	Duplicate Edges	PEs	Runtime Second	Million Edges/ PE/Second
This article	10^{12}	Copy	No	1,000	324.60	3.39
This article	10^{12}	Copy	Yes	1,000	84.79	12.97
KaGen [26, 43]	10^{12}	BA	Yes	1,000	73.83	14.89
Sanders et al. [43]	10^{15}	BA	Yes	16,384	3,600	16.95
Alam et al. [4]	2.5×10^{11}	Chung-Lu	No	1,024	12	20.35
Kepner et al. [30]	1.15×10^{12}	Kronecker	Yes	41,472	1	27.65
MP-BA [38]	6.6×10^{10}	BA	Yes	1 GPU	2,675	24.87/GPU
Alam et al. [5, 6]	2×10^{9}	Copy	No	1 GPU	2.3	869.57/GPU
Alam et al. [7]	1.6×10^{10}	Copy	No	4 GPUs	7	571.43/GPU

approximate algorithm and required manually adjusting several control parameters. The first exact distributed-memory-based parallel algorithm using the copy model was proposed in Reference [3]. Another distributed-memory-based parallel algorithm using the Barabási-Albert model was proposed in References [38, 43]. However, instead of using pseudo-random number generators, they used hash functions to generate the networks. A shared-memory-based parallel algorithm using the copy model was proposed in Reference [9].

Several other theoretical studies were done on the preferential attachment-based models. Machta and Machta [37] described how an evolving network can be generated in parallel. Dorogovtsev et al. [20] proposed a model that can generate graphs with fat-tailed degree distributions. In this model, starting with some random graphs, edges are randomly rewired according to some preferential choices. There exist other popular network models to generate networks with power law degree distribution. R-MAT [17] and stochastic Kronecker graph (SKG) [35] models can generate networks with power–law degree distribution using matrix multiplication. Due to its simpler parallel implementation, the Graph500 group [1] choose the SKG model in their supercomputer benchmark. Recently, a massively parallel network generators based on the Kronecker model was presented in Reference [30]. Highly scalable generators for Erdős-Rényi, 2D/3D random geometric graphs, 2D/3D Delaunay graphs, and hyperbolic random graphs are described in Reference [26]. The corresponding software library release also includes an implementation of the algorithm described in Reference [43]. An efficient and scalable algorithmic method to generate Chung–Lu, block two–level Erdős–Rényi (BTER), and stochastic blockmodels were also presented in Reference [4]. Their algorithm can generate power–law networks with a given expected power–law degree distribution. Recently there is a trend of using Graphics Processing Unit (GPU) for graph problems. A GPU-based preferential attachment-based algorithm using the copy model was proposed in References [5, 6]. A multi-GPU implementation of the preferential attachment-based algorithm using the copy model with the hash functions was presented in Reference [7].

A summary of runtime performances of parallel algorithms to generate power-law networks is presented in Table 2. All the corresponding numbers are collected from the corresponding paper. Although the underlying machines and architectures are different among these implementations, the numbers present a broad depiction of performance of these implementations. For comparative analysis among these implementations, we define the number of edges generated by each PE per second as our metrics. In this article, we used a generalized copy model to generate the power–law networks that still have dependencies and communications among PEs. Using hash functions instead of using pseudo-random generators can eliminate the communications and dependencies and hence yields better performance [43]. The Chung–Lu model also performs well to generate a power–law degree distribution although it does not preserve the network structure defined by the Barabási–Albert or the copy models [4]. Kronecker products are also very effective to generate power–law networks that require pre-computed matrices [30]. The GPU-based algorithms offer a significant improvement on performance, even with the most constraint copy model without any relaxation on a single GPU [5, 6]. The multi-GPU algorithm scales well with larger networks by leveraging hash functions [7].

7 CONCLUSION

We developed a parallel algorithm to generate massive scale-free networks using the preferential attachment model. We analyzed the dependency nature of the problem in detail, which led to the development of an efficient parallel algorithm for the problem. Various vertex partitioning schemes and their effect on the algorithm were discussed as well. Our algorithm produces networks that strictly follow power-law distribution. The linear scalability of our algorithm enables us to produce one trillion edges in just six minutes. It will be interesting to develop scalable parallel algorithms for other classes of random networks in the future.

REFERENCES

[1] 2016. Graph 500. Retrieved from https://graph500.org/.

[2] Maksudul Alam. 2016. *HPC-based Parallel Algorithms for Generating Random Networks and Some Other Network Analysis Problems*. Ph.D. Dissertation. Virginia Tech.

[3] Maksudul Alam, Maleq Khan, and Madhav V. Marathe. 2013. Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*. ACM Press, 1–12. DOI: https://doi.org/10.1145/2503210.2503291

[4] Maksudul Alam, Maleq Khan, Anil Vullikanti, and Madhav Marathe. 2016. An efficient and scalable algorithmic method for generating large-scale random graphs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*. IEEE. DOI: https://doi.org/10.1109/sc.2016.31

[5] Maksudul Alam and Kalyan S. Perumalla. 2017. *Generating Billion-Edge Scale-free Networks in Seconds: Performance Study of a Novel GPU-based Preferential Attachment Model.* Technical Report ORNL/TM-2017/486. Oak Ridge National Laboratory. DOI: https://doi.org/10.2172/1399438

[6] Maksudul Alam and Kalyan S. Perumalla. 2017. GPU-based parallel algorithm for generating massive scale-free networks using the preferential attachment model. In *Proceedings of the IEEE International Conference on Big Data (BigData'17)*. IEEE, 3302–3311. DOI: https://doi.org/10.1109/bigdata.2017.8258315

[7] Maksudul Alam, Kalyan S. Perumalla, and Peter Sanders. 2019. Novel parallel algorithms for fast multi-GPU-based generation of massive scale-free networks. *Data Sci. Eng.* 4, 1 (2019), 61–75. DOI: https://doi.org/10.1007/s41019-019-0088-6

[8] Réka Albert, Hawoong Jeong, and Albert-László Barabási. 2000. Error and attack tolerance of complex networks. *Nature* 406, 6794 (2000), 378–382. DOI: https://doi.org/10.1038/35019019

[9] Keyvan Azadbakht, Nikolaos Bezirgiannis, Frank S. de Boer, and Sadegh Aliakbary. 2016. A high-level and scalable approach for generating scale-free graphs using active objects. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC'16)*. ACM Press. DOI: https://doi.org/10.1145/2851613.2851722

[10] David A. Bader and Kamesh Madduri. 2006. Parallel algorithms for evaluating centrality indices in real-world networks. In *Proceedings of the International Conference on Parallel Processing*. 539–547. DOI: https://doi.org/10.1109/ICPP.2006.57

[11] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *Science* 286, 5439 (1999), 509–12. DOI: https://doi.org/10.1126/science.286.5439.509

[12] Albert-László Barabási and Zoltán N. Oltvai. 2004. Network biology: Understanding the cell's functional organization. *Nature Rev. Genet.* 5 (Jan. 2004), 101. http://dx.doi.org/10.1038/nrg1272.

[13] Christopher L. Barrett, Stephen G. Eubank, Anil Kumar S. Vullikanti, and Madhav V. Marathe. 2004. Understanding large-scale social and infrastructure networks: A simulation based approach. *SIAM News* 37, 4 (2004), 1–5. Retrieved from https://www.siam.org/pdf/news/227.pdf.

[14] Vladimir Batagelj and Ulrik Brandes. 2005. Efficient generation of large random networks. *Phys. Rev. E* 71, 3 Pt. 2A (2005), 036113. DOI: https://doi.org/10.1103/PhysRevE.71.036113

[15] Gunnar Blom, Lars Holst, and Dennis Sandell. 1994. *Problems and Snapshots from the World of Probability*. Springer, New York. Retrieved from https://www.ebook.de/de/product/3716793/gunnar_blom_lars_holst_dennis_sandell_problems_and_snapshots_from_the_world_of_probability.html.

[16] Jean M. Carlson and John Doyle. 1999. Highly optimized tolerance: A mechanism for power laws in designed systems. *Phys. Rev. E* 60, 2 (1999), 1412. DOI: https://doi.org/10.1103/PhysRevE.60.1412

[17] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining*. 442–446. DOI: https://doi.org/10.1137/1.9781611972740.43

[18] David P. Chassin and Christian Posse. 2005. Evaluating north american electric grid reliability using the barabási-albert network model. *Physica A* 355, 2–4 (2005), 667–677. DOI: https://doi.org/10.1016/j.physa.2005.02.051

[19] Sergey N. Dorogovtsev and José Fernando Ferreira Mendes. 2002. Evolution of networks. In *Advances in Physics*, Vol. 51. MIT Press, 1079–1187. DOI: https://doi.org/10.1080/00018730110112519

[20] Sergey N. Dorogovtsev, José Fernando Ferreira Mendes, and Alexander N. Samukhin. 2003. Principles of statistical mechanics of uncorrelated random networks. *Nuclear Phys. B* 666, 3 (2003), 396–416. DOI: https://doi.org/10.1016/S0550-3213(03)00504-2

[21] Paul Erdős and Alfréd Rényi. 1960. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.* 5 (1960), 17–61. http://www.math-inst.hu/ p_erdos/1967-11.pdf.

[22] Paul Erdős and Alfréd Rényi. 1961. On a classical problem of probability theory. *Publ. Math. Inst. Hung. Acad. Sci.* Ser. A 6 (1961), 215–220.

[23] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, Vol. 29. ACM Press, 251–262. DOI: https://doi.org/10.1145/316188.316229

[24] Marco Ferrante and Nadia Frigo. 2012. On the expected number of different records in a random sample. *Arxiv Preprint Arxiv:1209.4592*.

[25] Ove Frank and David Strauss. 1986. Markov graphs. *J. Amer. Statist. Assoc.* 81, 395 (1986), 832. DOI: https://doi.org/10.2307/2289017

[26] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. 2017. Communication-free massively distributed graph generation. Retrieved from http://arxiv.org/abs/1710.07565.

[27] Michelle Girvan and Mark E. J. Newman. 2002. Community structure in social and biological networks. *Proc. Natl. Acad. Sci. U.S.A.* 99, 12 (2002), 7821–7826. DOI: https://doi.org/10.1073/pnas.122653799

[28] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1989. *Concrete Mathematics: A Foundation for Computer Science*. Vol. 2. Addison-Wesley Longman Publishing. DOI: https://doi.org/10.2307/3619021

[29] Aric Hagberg, Daniel Schult, and Pieter Swart. 2008. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the Python in Science Conference*. 11–15. Retrieved from http://www.scopus.com/inward/record.url?eid=2-s2.0-67249148362&partnerID=tZOtx3y1.

[30] Jeremy Kepner, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Tim Davis, Vijay Gadepally, Michael Houle, Matthew Hubbell, Hayden Jananthan, Michael Jones, Anna Klein, Peter Michaleas, Roger Pearce, Lauren Milechin, Julie Mullen, Andrew Prout, Antonio Rosa, Geoffrey Sanders, Charles Yee, and Albert Reuther. 2018. Design, generation, and validation of extreme scale power-law graphs. Retrieved from http://arxiv.org/abs/1803.01281.

[31] Jon M. Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S. Tomkins. 1999. The web as a graph: Measurements, models, and methods. In *Proceedings of the Annual International Conference on Computing and Combinatorics*. Springer-Verlag, Berlin, 1–17. http://dl.acm.org/citation.cfm?id=1765751.1765753.

[32] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tomkins, and Eli Upfal. 2000. Stochastic models for the web graph. In *Proceedings of the Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc, 57–65. DOI: https://doi.org/10.1109/SFCS.2000.892065

[33] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is twitter, a social network or a news media? In *Proceedings of the International World Wide Web Conference Committee*. 1–10. DOI: https://doi.org/10.1145/1772690.1772751

[34] Vito Latora and Massimo Marchiori. 2004. Vulnerability and protection of critical infrastructures. *Phys. Rev. E* 71, 1 (2004), 4. DOI: https://doi.org/10.1103/PhysRevE.71.015103

[35] Jure Leskovec. 2010. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.* 11 (2010), 985–1042. Retrieved from http://www.jmlr.org/papers/v11/leskovec10a.html.

[36] Jure Leskovec and Eric Horvitz. 2008. Planetary-scale views on a large instant-messaging network. In *Proceedings of the International Conference on World Wide Web*. ACM Press, 915. DOI: https://doi.org/10.1145/1367497.1367620

[37] Benjamin Machta and Jonathan Machta. 2005. Parallel dynamics and computational complexity of network growth models. *Phys. Rev. E* 71, 2 (2005), 26704. DOI: https://doi.org/10.1103/PhysRevE.71.026704

[38] Ulrich Meyer and Manuel Penschuck. 2016. Generating massive scale-free networks under resource constraints. In *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments (ALENEX'16)*. Society for Industrial and Applied Mathematics. DOI: https://doi.org/10.1137/1.9781611974317.4

[39] Joel C. Miller and Aric Hagberg. 2011. Efficient generation of networks with given expected degrees. In *Proceedings of the International Workshop on Algorithms and Models for the Web-Graph*, Vol. 6732 LNCS. 115–126. DOI: https://doi.org/10.1007/978-3-642-21286-4_10

[40] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. 2011. Fast random graph generation. In *Proceedings of the International Conference on Extending Database Technology*. 331. DOI: https://doi.org/10.1145/1951365.1951406

[41] Romualdo Pastor-Satorras and Alessandro Vespignani. 2001. Epidemic spreading in scale-free networks. *Phys. Rev. Lett.* 86 (Apr 2001), 3200–3203. Issue 14. DOI: https://doi.org/10.1103/PhysRevLett.86.3200

[42] Garry Robins, Pip Pattison, Yuval Kalish, and Dean Lusher. 2007. An introduction to exponential random graph (p*) models for social networks. *Social Netw.* 29, 2 (2007), 173–191. DOI: https://doi.org/10.1016/j.socnet.2006.08.002

[43] Peter Sanders and Christian Schulz. 2016. Scalable generation of scale-free graphs. *Inform. Process. Lett.* 116, 7 (July 2016), 489–491. DOI: https://doi.org/10.1016/j.ipl.2016.02.004

[44] Georgos Siganos, Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 2003. Power laws and the as-level internet topology. *IEEE/ACM Trans. Netw.* 11, 4 (2003), 514–524. DOI: https://doi.org/10.1109/TNET.2003.815300

[45] Duncan J. Watts and Steven H. Strogatz. 1998. Collective dynamics of "small-world" networks. *Nature* 393, 6684 (1998), 440–442. DOI: https://doi.org/10.1038/30918

[46] Andy Yoo and Keith Henderson. 2010. Parallel generation of massive scale-free graphs. *Computing Research Repository.* Retrieved from http://arxiv.org/abs/1003.3684.

[47] Chenwei Zhang, Yi Bu, Ying Ding, and Jian Xu. 2018. Understanding scientific collaboration: Homophily, transitivity, and preferential attachment. *J. Assoc. Info. Sci. Technol.* 69, 1 (2018), 72–86.

Received June 2018; revised August 2019; accepted March 2020