Distributed-Memory Parallel Algorithms for Generating Massive Scale-free Networks Using 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, Virginia Tech, USA

Recently, there has been substantial interest in the study of various random networks as mathematical models of complex systems. As these 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 the sequential algorithms for generating random networks may not work due to the dependencies among the edges and the possibility of creating duplicate (parallel) edges. In this paper, we present MPI-based distributed memory parallel algorithms for generating random scale-free networks using the preferential-attachment model. Our algorithms scale very well to a large number of processors and provide almost linear speedups. The algorithms can generate scale-free networks with 400 billion edges in 5 minutes using 1024 processors.

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

Additional Key Words and Phrases: Network Science, Random Networks, Preferential Attachment, Distributed Algorithms

ACM Reference Format:
Maksudul Alam, Maleq Khan, Kalyan S. Perumalla, and Madhav Marathe. 2018. Distributed-Memory Parallel Algorithms for Generating Massive Scale-free Networks Using Preferential Attachment Model. *ACM Trans. Parallel Comput.* 1, 1, Article 01 (May 2018), 31 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

1.1 Motivation

Preferential attachment is a model that generates random scale-free networks, where a new vertex makes connections to some existing vertices that are chosen preferentially based on some of the properties of those vertices. For the preferential attachment model, the only previously known distributed-memory parallel algorithm is given by Yoo and Henderson [25]. Although useful, the algorithm has two weaknesses: (*i*) to deal with dependencies and the required complex synchronization, they came up with an approximation algorithm rather than an exact algorithm; and

Authors' addresses: Maksudul Alam, Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, TN, 37831, USA, alamm@ornl.gov; Maleq Khan, Texas A&M University-Kingsville, 700 University Blvd. Kingsville, TX, 78363, USA, maleq.khan@tamuk.edu; Kalyan S. Perumalla, Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, TN, 37831, USA, perumallaks@ornl.gov; Madhav Marathe, Virginia Tech, 1015 Life Science Circle, Blacksburg, VA, 24061, USA, mmarathe@vt.edu.

1539-9087/2018/5-ART01 $15.00
https://doi.org/0000001.0000001

(*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 [22] described how an evolving network can be generated in parallel. Dorogovtsev et al. [13] 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.

In this section, we study the problem of designing a distributed memory parallel algorithm for generating massive scale-free networks based on the preferential attachment (PA) model. To the best of our knowledge, our algorithms are the first distributed-memory parallel algorithms for generating random graphs while exactly following the preferential attachment model.

The rest of the section is organized as follows. Preliminaries, notations, and a description of the parallel computation model are given in Section 1.2. In Section 1.3, we describe the problem and algorithms. Some sequential algorithms are discussed in Section 1.3. In Section 2, we present our parallel algorithm for distributed memory architecture for the case where each vertex connects a single edge to the existing network. In Section 3, we extend the algorithm for the general case where each vertex contributes x edges to the existing network. Experimental results showing the performance of our parallel algorithms are presented in Section 5. Finally, we conclude in Section 7.

1.2 Preliminaries and Notations

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

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

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

1.3 Background: Preferential Attachment Model

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

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

Barabási-Albert Model. One way to generate a random PA network is to use the Barabási-Albert (BA) model. Many real-world networks have two important characteristics: (*i*) they are evolving in nature and (*ii*) the network tends to be scale free [8]. They provided a model, known as the Barabási-Albert (BA) model, where a new vertex is connected to an existing vertex that is chosen with probability directly proportional to its current degree.

The BA model works as follows. Starting with a small clique of $\hat{x}$ vertices, in every time step, a new vertex t is added to the network and connected to $x \leq \hat{x}$ randomly chosen existing vertices: $F_t(k)$ for $1 \leq k \leq x$ with $F_t(k) < t$; that is, $F_t(k)$ denotes the k-th vertex which t is connected to. 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, the probability $P_t(i)$ that vertex t is connected to vertex $i < t$ is given by $P_t(i) = \frac{d_i}{\sum_j d_j}$, where d_j represents the degree of vertex j.

The networks generated by the BA model are called the BA networks, which bear those two characteristics of a real-world network. 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 [8].

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 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 $O(m)$ [9]. This algorithm maintains a list of vertices such that each vertex i appears in this list exactly d_i times. The list can easily be updated dynamically by simply appending u and v to the list whenever a new edge (u, v) is added to the network. Now to find F_t, a vertex is chosen from the list uniformly at random. Since each vertex i occurs exactly d_i times in the list, we have $\Pr[F_t = i] = \frac{d_i}{\sum_j d_j}$. A sequential implementation of this algorithm is given in the graph algorithm library NetworkX [17].

Copy Model. As it turns out, the BA model does not easily lend itself to an efficient parallelization [3]. Another algorithm called the *copy model* [19, 20] preserves preferential attachment and power-law degree distribution. The copy model works as follows. Similar to the BA model, it starts with a small clique 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:

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

where l is a random outgoing connection from vertex k. Note that we also assume $F_k(l) = k$ 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 $p = \frac{1}{2}$.*

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); this event 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, and the outgoing edge to i is selected (out of x outgoing edges from v) in the second step (Equation 2); this event 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 [20]. 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 algorithm for generating preferential attachment networks and develop our parallel algorithm based on the copy model.

To summerize, Table 1 lists the symbols used in this paper.

Table 1. Symbols used in this paper

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 k-th edge from vertex t
$\mathbb{F}_t$	The set of outgoing ends of edges from vertex t

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, it requires that F_i is known for each $i < t$. As a result, any algorithm for preferential attachment seems to be highly sequential in nature: phase t cannot be executed until all previous phases are completed. However, a careful observation reveals that F_t can be partially, or sometimes completely, determined even before completing the previous phases. The copy model helps us exploit this observation in designing a parallel algorithm. However, it requires complex synchronizations and communications among the processors. 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.

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

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

```
/* Each processor $\mathcal{P}_i$ 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 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, processor $\mathcal{P}_i$ 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 $t' \in V_j$
```

2.1 Parallel Algorithm

The basic principle behind our parallel algorithm is as follows. Recall the sequential algorithm for the copy model. Each processors $\mathcal{P}_i$ 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, determination of F_t needs to wait until F_k is known. If $k \in V_j$ where $i \neq j$, processor $\mathcal{P}_i$ sends a *request* message to processor $\mathcal{P}_j$ to find F_k. Note that at the time when processor $\mathcal{P}_j$ receives this message, F_k can still be unknown. If so, $\mathcal{P}_j$ keeps this message in a queue 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 processor $\mathcal{P}_i$ is given in Algorithm 1. An example instance of the execution of this algorithm with seven vertices is depicted in Fig. 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 Fig. 1, computation of F_5 is waiting for F_3, which in turn is waiting

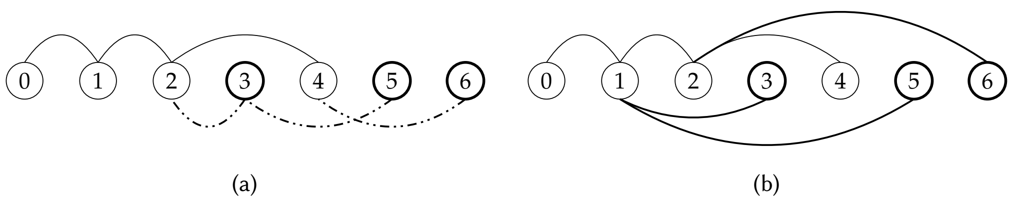


Fig. 1. A network with 7 vertices generated by Algorithm 1: a) an intermediate instance of the network in the middle of the execution of the algorithm, b) the final network. Solid lines show final resolved edges, and dashed lines show waiting of the vertices. For example, for vertex $t = 4$, k is chosen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

for F_2, and thus we have chain of dependency $\langle 5, 3, 2\rangle$. If the lengths of these chains are large, the waiting period for some vertices can be quite long, leading to poor performance of the parallel algorithm. Fortunately, the length of a dependency chain is small, and the performance of the algorithm is hardly affected by such waiting.

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 processor may need to wait for $O(\log n)$ steps, the processor hardly remains idle as it has other vertices to work with.

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

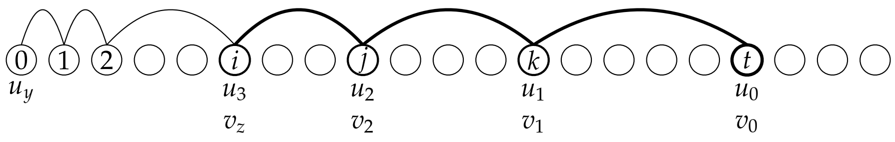


Fig. 2. Selection chain and dependency chain. The entire chain, which is marked by the solid lines, is a selection chain $\langle t, k, j, i, 2, 1, 0\rangle$, and the sub-chain marked by the thick solid lines is a dependency chain $\langle t, k, j, i\rangle$.

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

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

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

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, $P_{t+1}(i) = \frac{1}{i}$. 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 $i_1 < i_2 < \ldots < i_\ell$. 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

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

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

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

Let R_i be a random variable that denotes the random vertex selected for vertex i. Now observe that the occurrence of event $i_1 \in S_{i_2}$ can be fully determined by the variables in $\{R_j \mid i_1 < j \le i_2\}$; that is, event $i_1 \in S_{i_2}$ does not depend on any random variables other than the variables in $\{R_j \mid i_1 < j \le i_2\}$. Similarly, the events $i_2, \ldots, i_\ell \in S_t$ do not depend on any random variables other than the variables in $\{R_j \mid i_2 < j \le t\}$. Since the random variables R_is are chosen independently at random and the sets $\{R_j \mid i_1 < j \le i_2\}$ and $\{R_j \mid i_2 < j \le t\}$ 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}, $2 \le k \le \ell$, we have $\Pr\left[\bigcap_{k=2}^{\ell} A_{i_k}\right] = \prod_{k=2}^{\ell} \Pr\left[A_{i_k}\right]$. Then using Equation 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 $E[L_t] \le \log n$ and $L_{\max} = O(\log n)$ w.h.p., where n is the number of vertices.*

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

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

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

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

By Lemma 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,

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

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 pseudo-code of the algorithm is given in Algorithm 2. The basic structure of the algorithm for the general case is the same as that of the special case $x = 1$. We focus our discussion only on the modifications required and the differences between the two cases. The main difference is that, for each vertex t, instead of computing one edge (t, F_t), we need to compute x edges $(t, F_t(1)), (t, F_t(2)), \ldots, (t, F_t(x))$, and make sure such edges are distinct and do not create any parallel edges. For this general case, the set of vertices $\{F_t(1), F_t(2), \ldots, F_t(x)\}$ is denoted by $\mathbb{F}_t$.

3.1 Parallel Algorithm

The algorithm starts with an initial network, which is a clique of the first x vertices labeled $0, 1, 2, \ldots, x-1$. Each of the other vertices from x to $n-1$ generates x new edges. There are fundamentally two important issues that need to be handled for the general case: i) how we select $F_t(\ell)$ 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 redo the copy model. Such checking is done whenever a new edge is created.

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

```
/* Each processor $\mathcal{P}_i$ 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 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 4
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 ⟨request, $F_\ell(t), F_l(k)$⟩ to $\mathcal{P}_j$, where $k \in V_j$

/* Next, processor $\mathcal{P}_i$ receives messages sent to it and processes them as follows:   */
14 Upon receipt of message ⟨request, $F_{\ell'}(t'), F_{l'}(k')$⟩ from $\mathcal{P}_{j'}$:    // note that $k' \in V_i$
15 if $F_{l'}(k') \neq$ NULL then
16     send message ⟨resolved, $F_{\ell'}(t'), F_{l'}(k')$⟩ to $\mathcal{P}_{j'}$
17 else
18     store ⟨$F_{\ell'}(t'), F_{l'}(k')$⟩ in queue $Q_{k'}$

19 Upon receipt of message ⟨resolved, $F_t(\ell), v$⟩:
20 if $v \notin \mathbb{F}_t$ then
21     $F_t(\ell) \leftarrow v$
22     foreach ⟨$F_{\ell'}(t'), F_t(\ell)$⟩$\in Q_t$ do
23         send message ⟨resolved, $F_{\ell'}(t'), v$⟩ to $\mathcal{P}_j$ where $t' \in V_j$
24 else
25     $k \leftarrow$ a uniform random vertex in $[x, t-1]$
26     $l \leftarrow$ a uniform random number in $[1, x]$
27     re-send message ⟨request, $F_t(\ell), F_l(k)$⟩ to $\mathcal{P}_j$, where $k \in V_j$
```

For the ℓ-th edge of a vertex t, another vertex k is chosen from $[0, t-1]$ uniformly at random (Line 3). Edge (t, k) is created with probability p (Line 5). However, before creating such an edge (t, k) in Line 7, the existence of such an edge is checked immediately before creating them in Line 6. If the edge already exists at that time, the process is repeated again (Line 9). With the remaining $1 - p$ probability, t is connected to some vertex in $\mathbb{F}_k$; that is, we make an edge $(t, F_k(\ell))$, such that ℓ is chosen from $[1, x]$ uniformly at random. Similar to the special case $x = 1$, if k is in another processor, a request message is sent to that processor to find $F_k(\ell)$ (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 processor as t. Hence, request messages are sent to the processors containing k and k' to resolve $F_k(\ell)$ and $F_{k'}(\ell')$. If the ℓ-th 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 $\langle \texttt{resolved}, F_t(\ell), v \rangle$, the adjacency list of t is checked to find whether edge (t, v) already exists (Line 20). If the edge does not exist, it is created. Otherwise, new k and ℓ are selected (Line 25-26), and a new request message is sent (Line 27).

3.2 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 executes the copy model (irrespective of other outgoing edges from the same vertex) and follows the exact same procedures with the same probabilities as shown in the Lemma 2.1 and Lemma 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 $O\left(1 - \frac{x}{n^3}\right)$. As $x \leq n$, we can say that the length of the dependency chain is still $O(\log n)$ w.h.p.

Experimental Validation. We also experimentally evaluated the maximum length of dependency chain using our general algorithm (Algorithm 2). In this experiment, we varied the number of vertices n from $1K$ to $64M$. For each n, we also varied x from 1 to 128. For each possible combination of values of n and x, we calculated the maximum length of dependency chain by repeating the algorithm several times. Fig. 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.

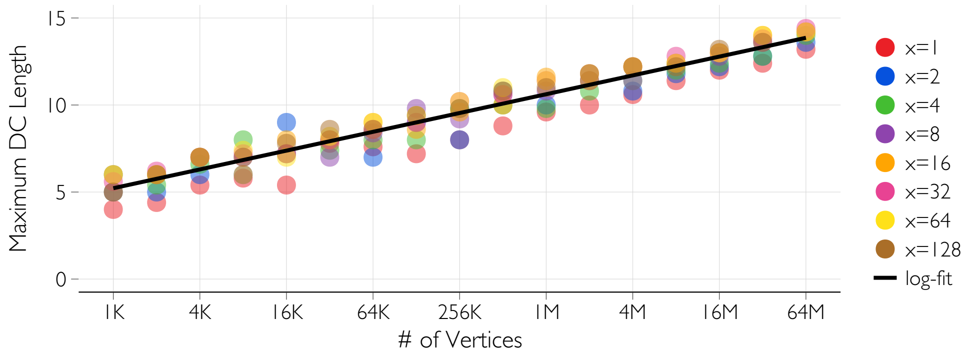


Fig. 3. Experimental result shows that the maximum length of dependency chain is $O(\log n)$. The horizontal axis (in log scale) represents the number of vertices and the vertical axis represents the length of the dependency chain. Filled circles show the maximum of dependency chain for each pair of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

3.3 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,

$$P_t(i) = \begin{cases} \frac{1}{t} + (1-p) \sum\limits_{k=x}^{t-1} \frac{1}{t} P_k(i) & i < x \\ \\ \frac{p}{t} + (1-p) \sum\limits_{k=i+1}^{t-1} \frac{1}{t} P_k(i) & i \geq x \end{cases} \tag{9}$$

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 9, we compute the probabilities numerically for $n = 10000$, $x = 4$, and $p = \{0.01, 0.5, 0.99\}$. The expected degree of a vertex t is given by:

$$E[d_i] = \begin{cases} \sum\limits_{k=i+1}^{n-1} P_k(i) & i < x \\ \\ x + \sum\limits_{k=i+1}^{n-1} P_k(i) & i \geq x \end{cases} \tag{10}$$

Fig. 4 shows the expected degree distribution by rounding off the expected degrees of each vertices to their nearest integer values. 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 shape of degree distributions. In the experimental section, we will show that our implementation produces the similar form of degree distribution for massive generated networks validating the implementation.

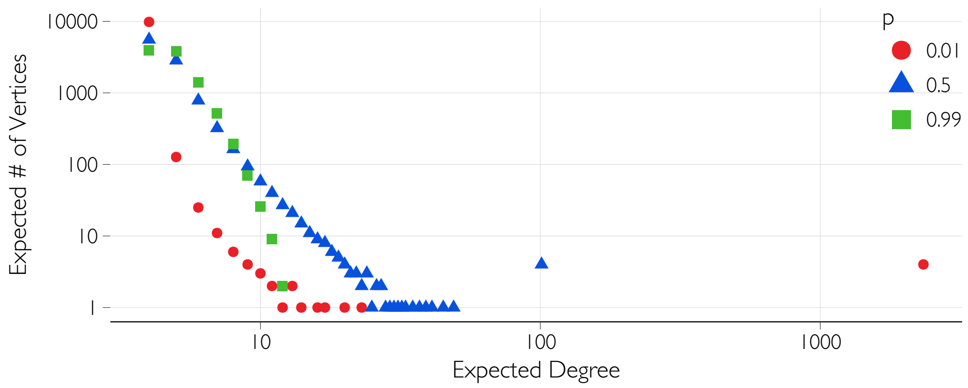


Fig. 4. Expected degree distribution of the algorithm for $n = 10000$, $x = 4$, and $p = \{0.01, 0.5, 0.99\}$. With different values of p different shapes of the degree distribution is achieved.

3.4 Bounding the Maximum Number of Regeneration of Edges

As noted earlier, with $x > 1$, there is a possibility of making duplicate edge during the edge creation process. As we are interested only to generate simple graphs with no self-loop or no parallel edges, we avoid the creation of duplicate edges by checking for potential duplicate edge and execute the copy model again if necessary. However, if the number of regeneration of edges is very large, 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 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 is no close form results on the expected number of trials required. In [14], the authors presented a formulation of the problem as follows. Let X_k denotes the number of trials to get k-th unique vertices given that $k-1$-th vertices are unique. Note that $X_1 = 1$, that is 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: $X = X_1 + X_2 + X_3 + \ldots + X_x$. Then, the expected number of trials is given by [14]:

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

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.

Due to the lack of close form solutions and intractable form of exact solution, we instead analyze the maximum number of trails 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 execution of the copy model, we collected the maximum number of trials needed for a vertex. In Fig. 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 a 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.

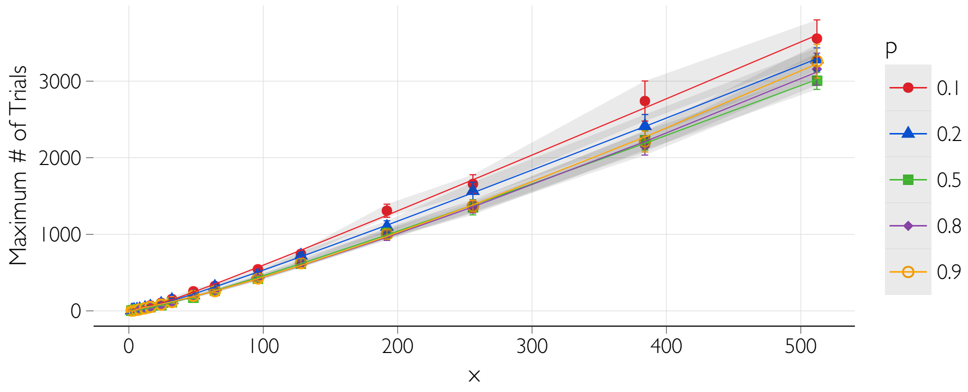


Fig. 5. Experimental results show that the maximum number of required trials is $O(x \log x)$

Although, the maximum number of trials is $O(x \log x)$, not all vertices require that many trials. In fact, as t becomes larger and larger than x the number of trials required becomes smaller. In Fig. 6 we show the number of trials required in each vertex for different values of x with $p = 0.5$.

Only the first 1000 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.

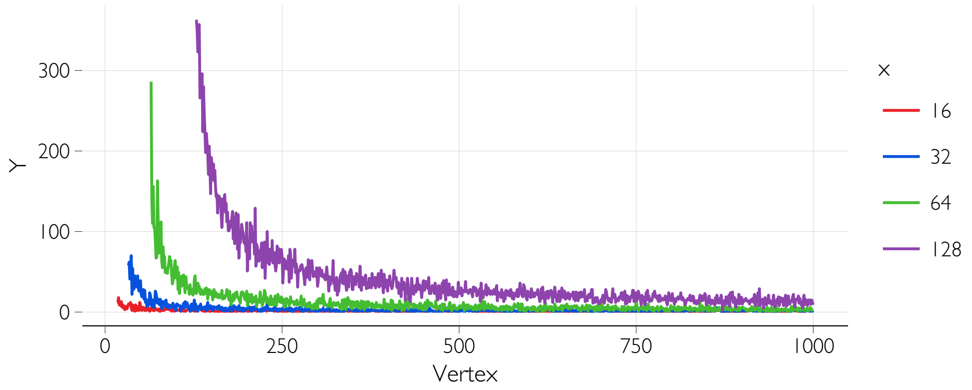


Fig. 6. Experimental results show that the number of required trials reduces as t becomes larger and larger than x

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 Fig. 7 we show the average percentage overhead per vertex for $n = 100K$ 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 is mostly of theoretical interest rather the most practical scenarios have large values of p. For almost all practical purposes the average overhead per vertex is very small.

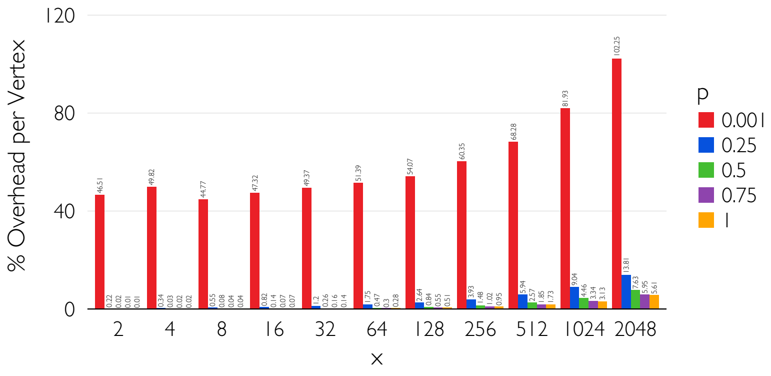


Fig. 7. Experimental results show that the maximum number of retrials is $O(x \log x)$

3.5 Analysis of Waiting Queue Size

In our parallel algorithm, after receiving a request message for an edge $F_l(k)$ (Line 14 of Algorithm 2), a processor sends a corresponding response message immediately if the edge $F_l(k)$ is already known. Otherwise, the request message is stored in a queue called the *waiting queue* (Line 18 of Algorithm 2). If a processor receives a large number of such request messages whose responses could not be sent immediately, the size of the waiting queues becomes large, leading to a large memory requirement and the parallel algorithm 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 stored in the waiting queue of a vertex is $O(x \log n)$ with high probability.*

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

Therefore, while creating a copy edge $(t, F_k(\ell))$, the event that the request message will be put in the waiting queue Q_k consists 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 $P_{k_\ell}(t)$ denotes the probability that any outgoing edge from vertex t makes a copy edge $(t, F_t(\ell))$ and the corresponding request message is put in the waiting queue Q_k. Therefore, we have:

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

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

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

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

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

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

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

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

$$\begin{aligned}
E\Big[|Q_k|\Big] = \sum_{t=k+1}^{n-1} E[Y_k(t)] &= \sum_{t=k+1}^{n-1} \sum_{\ell=1}^{x} E\left[X_{k_\ell}(t)\right] \\
&\leq \sum_{t=k+1}^{n-1} \sum_{\ell=1}^{x} (1-p)^2 \frac{1}{(t-1)} \\
&\leq (1-p)^2 x \sum_{t=k+1}^{n-1} \frac{1}{t-1} \\
&\leq (1-p)^2 x \sum_{i=k}^{n-2} \frac{1}{i} \\
&\leq (1-p)^2 x \left(H_{n-2} - H_i\right) \\
&\leq (1-p)^2 x H_n \\
&\leq (1-p)^2 x \log n. \qquad (13)
\end{aligned}$$

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

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

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

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

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

3.6 Experimental Validation of Waiting Queue Size

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

In Fig. 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. The horizontal axis is plotted in log scale. The figure demonstrates that the maximum size of a waiting queue is proportional to $\log n$.

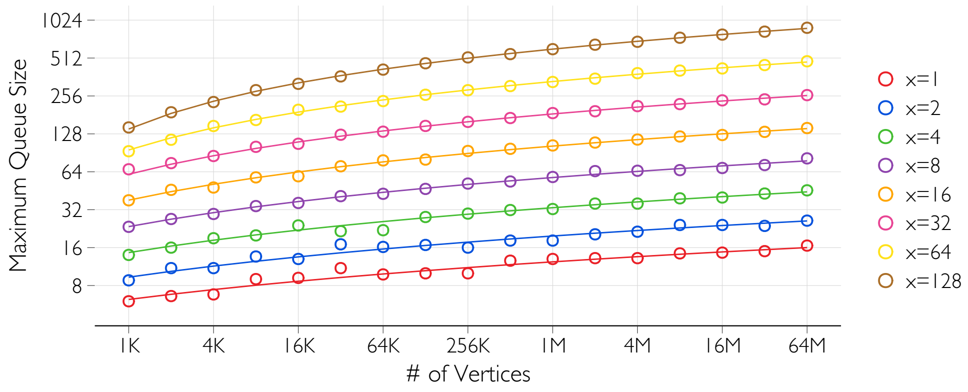


Fig. 8. The maximum size of the waiting queues changes logarithmically with n

In Fig. 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.

In Fig. 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 processor. Vertex partitioning has significant effects on the performance of the algorithm. In this section, we study several partitioning schemes and their effects on load balancing and the performance of the algorithm. In our algorithm, we measure the computational load in terms of the number of vertices per processor, the number of outgoing messages (request message) from a processor, and the number of incoming messages (response messages) to a processor.

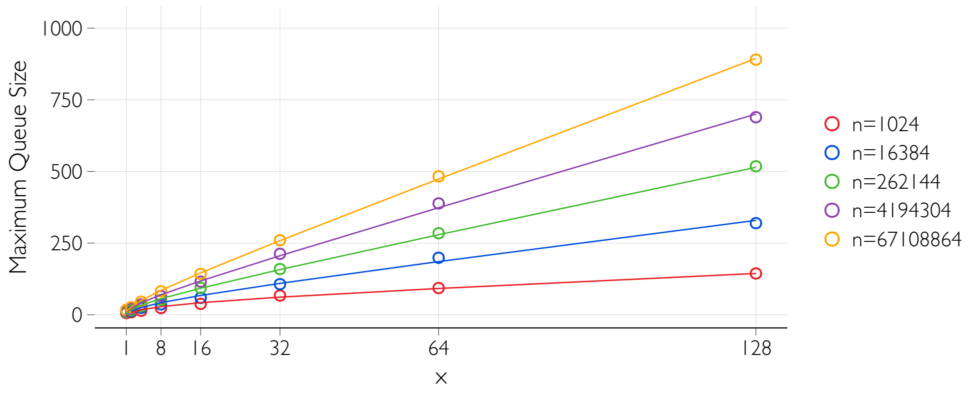


Fig. 9. The maximum size of the waiting queues changes linearly with x

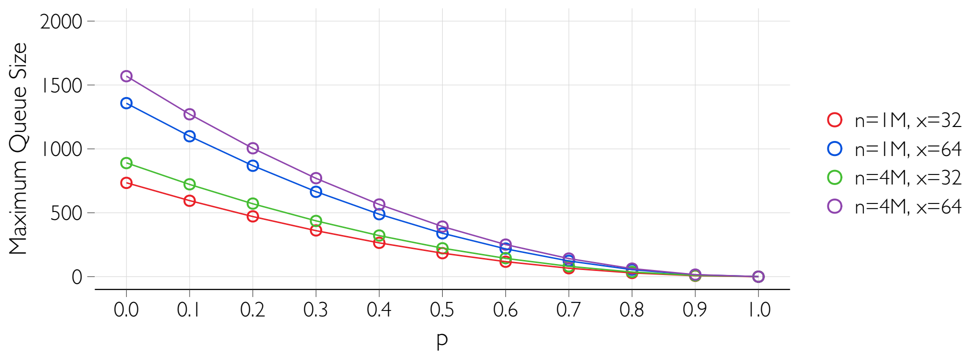


Fig. 10. The maximum size of the waiting queues changes with $(1 - p)^2$

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

A. For any given $k \in V$, finding the processor $\mathcal{P}_j$, where $k \in V_j$ (Line 8, Algorithm 1), can be done efficiently, preferably in constant time without communicating with the other processors.
B. The partitioning should lead to a good load balancing. The degrees of the vertices vary significantly, and a vertex with a larger degree causes more messages to work with. As a result, naïve partitioning may lead to poor load balancing.
C. As we discuss later, combining multiple messages (to the same destination) and using a `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 processors 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 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 processor from a vertex u is pretty straightforward in the SCP scheme. For a vertex $u \in V_i$, the rank of the processor $\mathcal{P}_i$ is given by $i = \left\lfloor \frac{u}{B} \right\rfloor$.

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 processor $\mathcal{P}_i$ involves the following three types of load:

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

The computational load for load type A and B above is directly proportional to the number of vertices in partition V_i. Computational load for load type C depends not only on the number of vertices in a processors but also on i, the rank of the processor. With simple consecutve vertex partitioning (SCP), a lower ranked processor receives more request messages than a higher ranked processor, because with $j < k$, $E[M_j] > E[M_k]$, where M_k is the number of request messages received for Vertex k (see Lemma 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 decided to assign F_k to F_t. The probability of such an event is $(1-p)\frac{1}{t}$. Then the expected number of messages received for Vertex k is given by

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

□

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

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

The second last line follows from Equation 2.36 in page 41 of [16]. Thus, using another constant $b = 1 + c$, the total computational load at Processor $\mathcal{P}_i$ is

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

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

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

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 and 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). Fig. 11 shows the distribution of the vertices among processors 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.

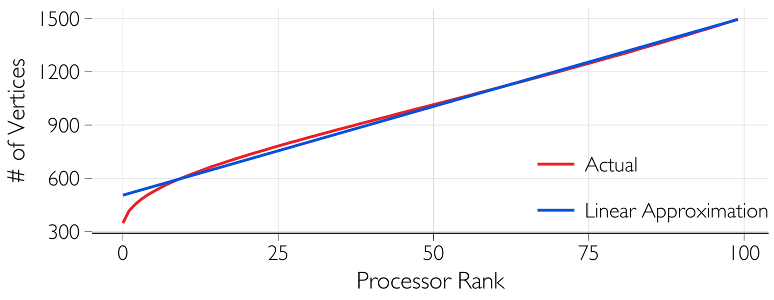


Fig. 11. Distribution of the vertices among processors for actual solutions of Equation 15 and its linear approximation.

As in the LCP scheme, the number of vertices is increasing linearly with i (the ranks of the processors), the number of vertices in Processor $\mathcal{P}_i$ follows the arithmetic progression a, $a + d$, $a + 2d, \ldots, a + (P - 1)d$, that is, the number of vertices in Processor $\mathcal{P}_i$ is $B_i = a + id$, where d is the slope of the line for linear approximation as shown in Fig. 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 processor for vetex u is more complicated in this scheme. Given a vertex u, we need to find the processor $\mathcal{P}_i$ such that $u \in V_i$.

Vertex u satisfies the following inequality:

$$\sum_{j=0}^{i-1}(a+jd) \leq u < \sum_{j=0}^{i}(a+jd)$$

$$\frac{i\,(2a+(i-1)d)}{2} \leq u < \frac{(i+1)\,(2a+id)}{2} \tag{16}$$

Solving the inequality 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 processors P. Parameter d is the slope of the straight line $y = a + dx$, where y represent the number of vertices in the processor with rank $x = i$. We calculate d by finding two points on this straight line. Putting $i = 0$ and $i = P-1$ in Equation 15, we can compute n_1 and n_{P-1}. Then, the number of vertices in the first processor is $n_1 - n_0 = n_1$ and the number of vertices in last processor is $n_P - n_{P-1} = n - 1 - n_{P-1}$. Hence, we have

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

Now, we have

$$\sum_{j=0}^{P-1}(a+jd) = n$$

$$\frac{P\,(2a+(P-1)d)}{2} = n$$

$$a = \frac{n}{P} - \frac{(P-1)d}{2} \tag{18}$$

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

4.2 Round-Robin Partitioning (RRP)

In this scheme, vertices are distributed in a round robin fashion among all processors. Partition V_i contains the vertices $\langle i, i+p, i+2p, \ldots, i+kp \rangle$ such that $i+kp \leq n < i+(k+1)p$; that is, $V_i = \{j | j \bmod P = i\}$. In other words, vertex i is assigned to set $V_{i \bmod p}$. Similar to SCP, in this RRP scheme also, the number of vertices in the sets is almost equal. The number of vertices in a set is either $\lceil n/p \rceil$ 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 processors are well-balanced as shown in Lemma 4.2.

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

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

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

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

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

If n is a multiple of P, 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 processor where the vertex belongs to can be computed in constant time. Finding the rank of processor $\mathcal{P}_i$ for a given vertex $u \in V_i$ is determined by $i = u \mod P$.

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

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

4.3 Segmented Partitioning

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

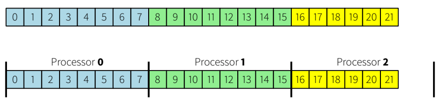
In the segmented partitioning technique, first the entire set of vertices are partitioned into k consecutive subsets $S_1, S_2, S_3, \ldots, S_k$ called *segments* (similar to the consecutive partitioning). From the copy model definition, clearly vertices on a segment S_i may only depend on vertices on segment S_j where $i \geq j$ but not vice versa. Let $B_i = |S_i|$ denotes 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 threads. After a round is completed, every edges 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 Fig. 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 average 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.

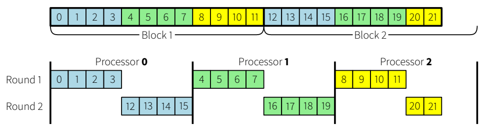
However, as segment size is kept increasing beyond some point, we start losing the advantages because of synchronization issues needed to perform in each round. Therefore, there is an optimal value of k. We experimentally varied k to determine the optimal value for each partitioning schemes.

5 EXPERIMENTAL RESULTS

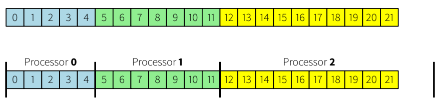
In this section, we evaluate the performance of our algorithms experimentally. The accuracy of our parallel algorithm is demonstrated by showing that the algorithm produces networks with power law degree distribution. Then we present the strong and weak scaling of the algorithms. These algorithms scale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



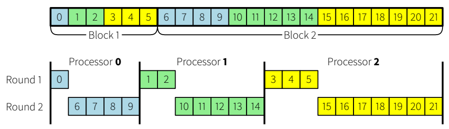
(a) Unsegmented SCP



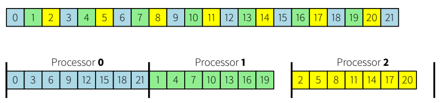
(b) Segmented SCP with $k = 2$



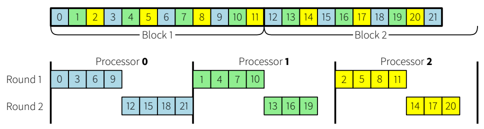
(c) Unsegmented LCP



(d) Segmented LCP with $k = 2$



(e) Unsegmented RRP



(f) Segmented RRP with $k = 2$

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

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

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

5.1 Validating Scale-Free Property with Degree Distribution

The degree distribution of the graph generated by our parallel algorithm is shown in Fig. 13 in a log – log scale. We used $n = 1B$ 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 $2 < \gamma < \infty$ [12]. When p is very close to 0, the network is mainly built on copy edges, therefore, there is a higher level of bias towards the higher degree vertices as evident from the longer tail. However, when p is close to 1, the network mainly consists of direct edges, and we don't 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 also shows that our algorithms produce scale-free networks very accurately.

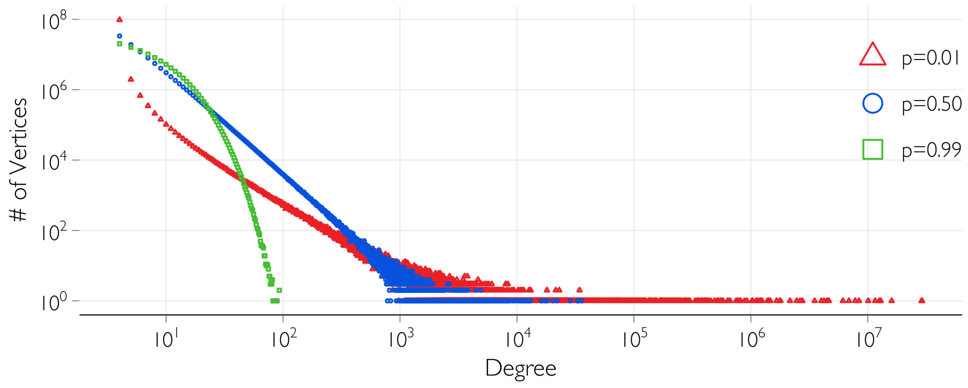
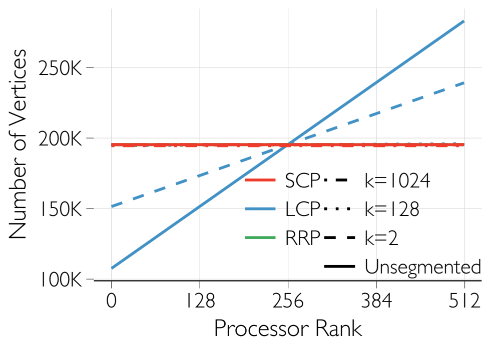


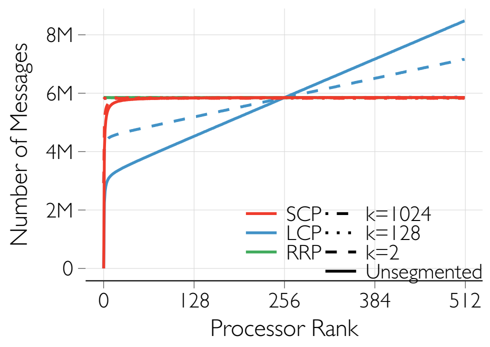
Fig. 13. The degree distribution (in log – log scale) of the network generated by our parallel algorithms. The network is generated with $n = 10^9$ and $x = 4$.

5.2 Performance of Partitioning and Load Balancing

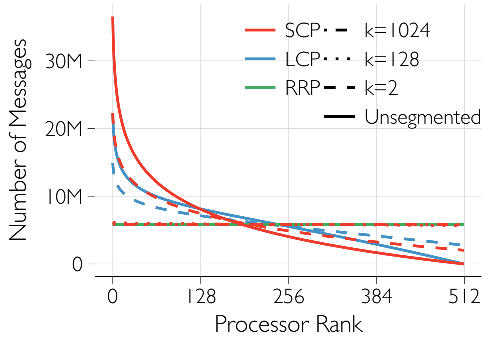
Vertex partitioning has significant effects on load balancing and performance of the algorithm. In Section 4, we have discussed three partitioning schemes SCP, LCP, and RRP, and theoretically analyzed them. In this section, we experimentally study these schemes and their effect on the performance of the algorithm. In these experiments, we use $n = 100M$ vertices, $x = 60$ edges per vertex, and 512 processors. 512 processors are sufficient to demonstrate the behavior and differences of the partitioning schemes. For each of the three schemes, we measure the computational load in the processors by the number of vertices per processor, the number of outgoing messages from the processors, and the number of incoming messages to the processors. The results are shown in Fig. 14.



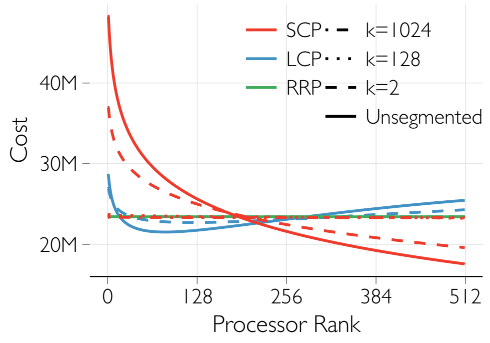
(a) Vertex distribution



(b) Outgoing messages



(c) Incoming messages



(d) Total computational load

Fig. 14. Vertex and message distribution for the partitioning schemes

Vertex Distribution. The vertex distribution is shown in Fig. 14(a). For SCP and RRP, vertices are distributed uniformly among the processors, and each processor has about 195K vertices. For LCP, the number of vertices in the processors are increasing linearly with the rank of the processors.

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

Fig. 14(c) shows the number of incoming request messages for each processor. It is clear that a lower ranked processor receives more messages than a higher ranked processor 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 processors.

Total Load Distribution. Besides sending and receiving messages, for each vertex, a processor can incur a constant other computational cost. Thus, for analysis purposes, we measure the total computational load of a processor as the sum of the number of vertices in the processor and the number of incoming and outgoing messages. Fig. 14(d) shows the total load for the three partitioning schemes. The RRP scheme distributes the load almost perfectly among the processors. Load balancing in the LCP scheme is also quite good. On the other hand, the SCP scheme distributes the load very poorly. These experimental results verify our theoretical analysis given in Section 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 Fig. 15. Therefore, segmented partitions yield better performance in our algorithm.

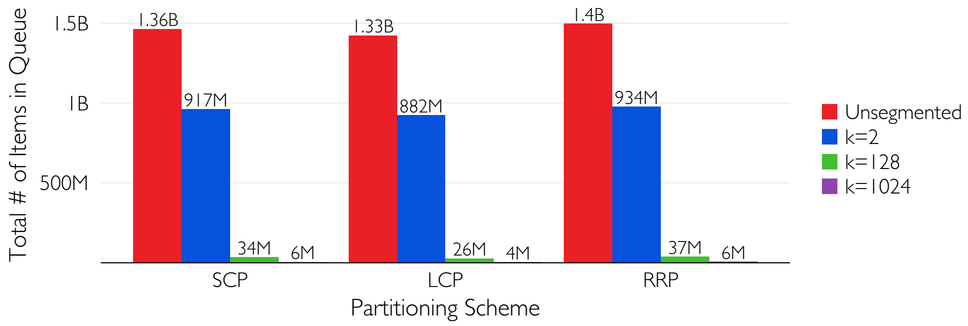


Fig. 15. Change of run time based on segment size

Effect of 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, the performance would start to decrease. This is demonstrated in Fig. 16.

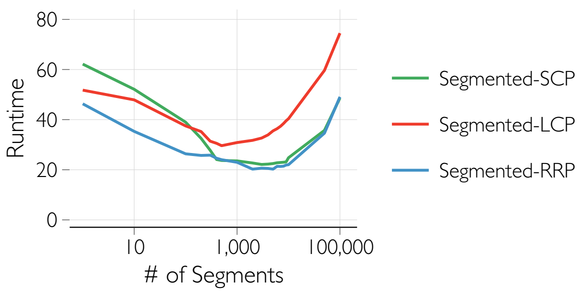


Fig. 16. Change of run time based on segment size

Effect of p on Performance. If p is reduced, most of the edges produced consists of copy edges, therefore requiring more message exchanges. As p is increased towards 1, most edges consist of direct edges. Therefore communication is reduced. This is shown in Fig. 17.

5.3 Parallel Execution and Scalability

Strong Scaling. Strong scaling of a parallel algorithm shows its performance with an increasing number of processors keeping the problem size fixed. Fig. 18 shows speedup factors of our algorithms with segmented and unsegmented techniques using simple consecutive (SCP), linear consecutive (LCP), and round-robin partitioning (RRP) partitioning schemes, as the number of processors

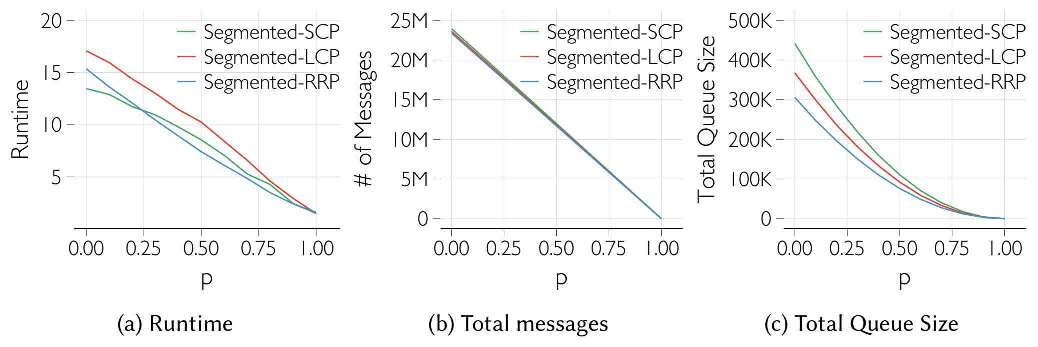


Fig. 17. Effect of segmented partitioning on segment size

increases with problem size $n = 100\mathbf{M}$ and $x = 60$. Speedup factors are measured as T_s/T_p, where T_s and T_p are the running time of a sequential algorithm and the parallel algorithm, respectively. We have implemented the sequential version of our algorithm in C++. This sequential implementation outperforms the best available implementation of the BA model given in the NetworkX graph algorithm library [17]. As the sequential algorithm cannot generate more than $6\mathbf{B}$ edges due to memory limitations, we choose $n = 100\mathbf{M}$ and $x = 60$. We varied the number of processors from 1 to 1024 for this experiment.

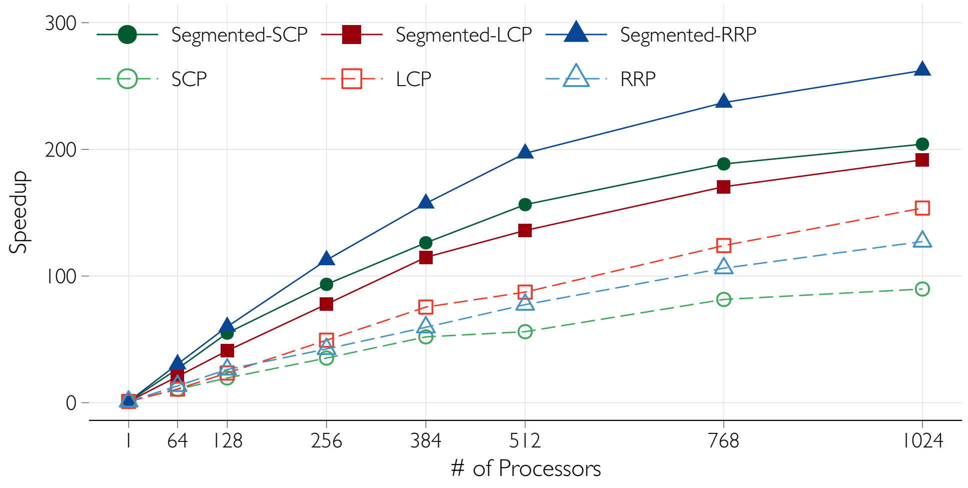


Fig. 18. The strong scaling of our parallel algorithms for the problem size $n = 100M$ and $x = 60$.

Parallelization of network algorithms is notoriously hard. Furthermore, we have observed that the problem of generating a scale-free random network is quite sequential in nature due to the dependencies among the edges. As Fig. 18 shows, the speedups of our algorithms are increasing almost linearly with the number of processors. Given the sequential nature of the problem, our algorithms show very good speedup. Further, the speedup of segmented versions performs better. Note that both B-SCP and B-RRP is 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 processor remains constant. For this experiment, we varied the number of processors from 16 to 1024. With the number of processors, the input size is also increased proportionally: for P processors, a network with $10^7 P$ edges is generated. Fig. 19 shows the weak scaling of our algorithms with the increasing number of processors.

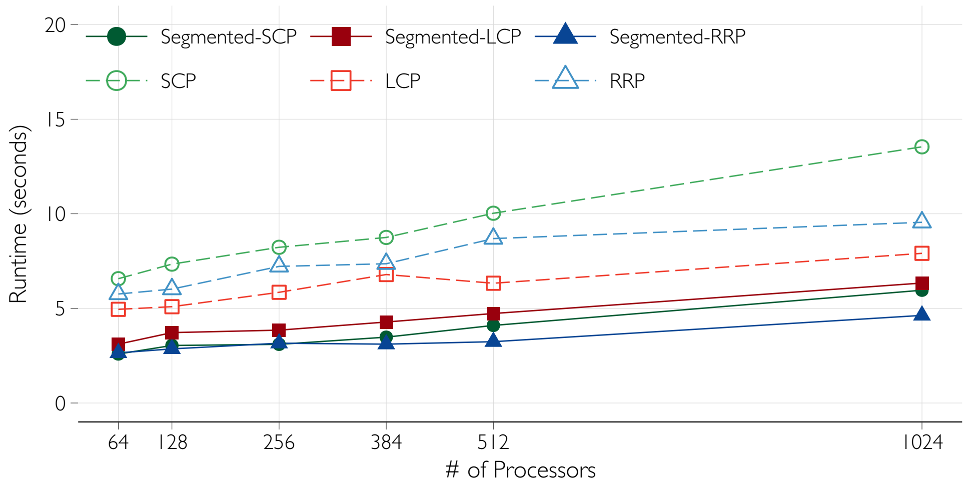


Fig. 19. Weak scaling of our parallel PA algorithm.

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

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

6 RELATED WORKS

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 [9]. A distributed memory–based algorithm to generate preferential attachment networks was proposed in [25]. However, their algorithm was not exact, rather an approximate algorithm and required manually adjusting several control parameters. The first exact distributed memory–based parallel algorithm using the copy model was proposed in [3]. Another distributed memory–based parallel algorithm using the Barabási–Albert model was proposed in [23, 24]. 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 [7].

Several other theoretical studies were done on the preferential attachment-based models. Machta and Machta [22] described how an evolving network can be generated in parallel. Dorogovtsev et al. [13] 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 exists other popular network models to generate networks with power law degree distribution. R-MAT [11] and stochastic Kronecher graph (SKG) [21] models can generate networks with power–law degree distribution using matrix multiplication. Due to it's simpler parallel implementation, the Graph500 group [1] choose the SKG model in their supercomputer benchmark. Recently, a massively parallel network generators based on the Kronecher model was presented in [18]. Highly scalable generators for Erdős-Rényi, 2D/3D random geometric graphs, 2D/3D Delaunay graphs, and hyperbolic random graphs are described in [15]. The corresponding software library release also includes an implementation of the algorithm described in [24]. An efficient and scalable algorithmic method to generate Chung–Lu, block two–level Erdős–Rényi (BTER), and stochastic blockmodels were also presented in [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 on the GPU using the copy model was proposed in [5, 6].

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 depictions of performance of these implementations. For comparative analysis among these implementations, we define the number of edges generated by each processor per second as our metrics. In this paper, we used a generalized copy model to generate the power–law networks that still have dependencies and communications among processors. A relaxation of using hash functions instead of using pseudo-random generators can eliminate the communications and dependencies and hence yields better performance [24]. The Chung–Lu model also performs well to generate a power–law degree distribution although it does not preserve the network structure define by the Barabási–Albert or the Copy models [4]. Kronecher products are also very effective to generate power–law networks that require pre-computed matrices to generate the networks [18]. 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]. It remains to be seen how the algorithm scales with larger networks with multiple GPUs.

Table 2. Runtime performance recent power-law network generators

Implementation	Edges	Processors	Runtime (sec.)	Million Edges/Proc./Sec.
Alam et al. [3] (this paper)	4×10^{11}	1024	300	4.44
Sanders et al. [10]	10^{15}	16384	3600	16.95
Alam et. al. [4]	2.5×10^{11}	1024	12	20.35
Kepner et. al. [18]	1.15×10^{12}	41472	1	27.65
Alam et. al. [5, 6]	2×10^{9}	1 GPU	2.3	869.57 (Per GPU)

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 that 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 which strictly follow power-law distribution. The linear scalability of our algorithm enables us to produce 400 billion edges in just five minutes. It will be interesting to develop scalable parallel algorithms for other classes of random networks in the future.

ACKNOWLEDGMENTS

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

This paper 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.

REFERENCES

[1] 2010. Graph 500. http://www.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 *International Conference for High Performance Computing, Networking, Storage and Analysis.* ACM Press, 1–12. 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 Press, Article 32, 12 pages. 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. 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 *IEEE International Conference on Big Data (Big Data).* 3302–3311. https://doi.org/10.1109/BigData.2017.8258315

[7] 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 *Proc. of the Annual ACM Symposium on Applied Computing.* https://doi.org/10.1145/2851613.2851722

[8] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–12. https://doi.org/10.1126/science.286.5439.509

[9] Vladimir Batagelj and Ulrik Brandes. 2005. Efficient Generation of Large Random Networks. *Physical Review E* 71, 3 Pt 2A (2005), 036113. https://doi.org/10.1103/PhysRevE.71.036113

[10] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A Large Time-Aware Web Graph. *ACM SIGIR Forum* 42, 2 (2008), 33. https://doi.org/10.1145/1480506.1480511

[11] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SIAM International Conference on Data Mining.* 442–446. https://doi.org/10.1137/1.9781611972740.43

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

[13] Sergey N. Dorogovtsev, José Fernando Ferreira Mendes, and Alexander N. Samukhin. 2003. Principles of Statistical Mechanics of Uncorrelated Random Networks. *Nuclear Physics B* 666, 3 (2003), 396–416. https://doi.org/10.1016/S0550-3213(03)00504-2

[14] Marco Ferrante and Nadia Frigo. 2012. On the expected number of different records in a random sample. *arXiv preprint arXiv:1209.4592* (2012).

[15] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. 2017. Communication-free Massively Distributed Graph Generation. *CoRR* abs/1710.07565 (2017). arXiv:1710.07565 http://arxiv.org/abs/1710.07565

[16] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1989. *Concrete Mathematics: A Foundation for Computer Science.* Vol. 2. Addison-Wesley Longman Publishing Co., Inc. xiii + 625 pages. https://doi.org/10.2307/3619021

[17] Aric Hagberg, Daniel Schult, and Pieter Swart. 2008. Exploring Network Structure, Dynamics, and Function Using NetworkX. In *Python in Science Conference*. 11–15. http://www.scopus.com/inward/record.url?eid=2-s2.0-67249148362&partnerID=tZOtx3y1

[18] 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. *CoRR* abs/1803.01281 (2018). arXiv:1803.01281 http://arxiv.org/abs/1803.01281

[19] Jon M. Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S. Tomkins. 1999. The Web as a Graph: Measurements, Models, and Methods. In *Annual International Conference on Computing and Combinatorics*. Springer-Verlag, Berlin, Heidelberg, 1–17. http://dl.acm.org/citation.cfm?id=1765751.1765753

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

[21] Jure Leskovec. 2010. Kronecker Graphs: An Approach to Modeling Networks. *Journal of Machine Learning Research* 11 (2010), 985–1042. http://www.jmlr.org/papers/v11/leskovec10a.html

[22] Benjamin Machta and Jonathan Machta. 2005. Parallel Dynamics and Computational Complexity of Network Growth Models. *Physical Review E* 71, 2 (2005), 26704. https://doi.org/10.1103/PhysRevE.71.026704

[23] Ulrich Meyer and Manuel Penschuck. 2016. Generating massive scale-free networks under resource constraints. In *Proc. of the Workshop on Algorithm Engineering and Experiments (ALENEX)*.

[24] Peter Sanders and Christian Schulz. 2016. Scalable generation of scale-free graphs. *Info. Proc. Letters* (2016). https://doi.org/10.1016/j.ipl.2016.02.004

[25] Andy Yoo and Keith Henderson. 2010. Parallel Generation of Massive Scale-Free Graphs. *Computing Research Repository* abs/1003.3 (2010), 1–13. http://arxiv.org/abs/1003.3684

Received May 2018