Distributed-Memory Parallel Algorithms for Generating Massive Scale-free Networks Using Preferential Attachment Model

Maksudul Alam*† maksud@vbi.vt.edu
Maleq Khan† maleq@vbi.vt.edu
Madhav V. Marathe*† mmarathe@vbi.vt.edu

10 August, 2013

Abstract

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. Naive 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 50 billion edges in 123 seconds using 768 processors.

Keywords. scale-free networks; Big Data; high performance computing; preferential attachment; random networks; parallel algorithms; copy model

1 Introduction

Advances in hardware technology as well as the developments in software and algorithms have enabled the detailed study of complex networks. Complex networks such as the Internet [12, 26], biological networks [14], social networks [18, 20], and various infrastructure networks [4, 8, 19] are abstracted as random graphs for the purposes of obtaining rigorous mathematical results; see e.g. [8]. The study of these complex systems have significantly increased the interest in various random graph models [5]. As some of the complex networks grow, it has become necessary to correspondingly generate massive random networks efficiently. As discussed in [21], a smaller network may not exhibit the same behavior, even if both networks are generated using the same model. In [21], by experimental analysis, it was shown that the structure of larger networks is fundamentally different from small networks, and many patterns emerge only in massive datasets. 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 [2].

Many random graph models have been developed in the past. Among them, the first and well-studied model is the Erdös–Rényi model [11]. However, the Erdös–Rényi model does not exhibit the characteristics observed in many real-world complex systems [5]. As a result, many other random graph models, such as small-world [27], Barabási–Albert [1, 3], Chung-Lu [23], exponential random graph [13, 25], R-MAT [7], and HOT [6] models, have been proposed.

Barabási and Albert [3] discovered a class of inhomogeneous networks, called scale-free networks, characterized by a power-law degree distribution $P(k) \propto k^{-\gamma}$, where k represents the degree of a node. While highly connected nodes are improbable in exponential networks, they do occur with statistically significant probability in scale-free networks. Furthermore, the work of Albert et al. [1] suggests these highly interconnected nodes appear to play an important role in the behavior of scale-free systems, particularly with respect to their resilience [8].

*Department of Computer Science, Virginia Tech
†Network Dynamics and Simulation Science Laboratory, Virginia Bioinformatics Institute

Watts and Strogatz [27] described small-world networks, which also lead to relatively homogenous topology [8]. This model transforms a regular one-dimensional lattice (with vertex degree of four or higher) by rewiring each edge, with certain probability, to a randomly chosen vertex. It has been found that, even with the small rewiring probability, the average shortest-path length of the resulting graphs is of the order of random graphs, and generates graphs with fat-tailed degree distributions [28].

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. Batagelj and Brandes [5] justifiably said "To our surprise we have found that the algorithms used for these generators in software such as BRITE, GT-ITM, JUNG, or LEDA are rather inefficient. ...superlinear algorithms are sometimes tolerable in the analysis of networks with tens of thousands of nodes, but they are clearly unacceptable for generating large numbers of such graphs." As a step towards meeting this goal, recently efficient sequential algorithms have been developed to generate certain classes of random graphs: Erdös–Rényi [5], small world [5], Preferential Attachment [5, 24], and Chung-Lu [23].

However, although efficient sequential algorithms are able to generate networks with millions of nodes quickly, generating networks with billions of nodes 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.

The design of parallel distributed memory algorithms poses two main challenges in the context of generating random graphs. Firstly, the dependencies among the edges, especially in the preferential-attachment model, impede independent operations of the processors. Second, different processors can create duplicate edges, which must be avoided. Dealing with both of these problems requires complex synchronization and communications among the processors, and thus gaining satisfactory speedup by the parallelization becomes a challenging problem. Even for the Erdös–Rényi model where the existence of edges are independent of each other, parallelization of a non-naive efficient algorithm, such as the algorithm by Batagelj and Brandes [5], is a non-trivial problem. A parallelization of Batagelj and Brandes's algorithm was recently proposed in [24].

For the preferential attachment model, the only previously known distributed-memory parallel algorithm is given by Yoo and Henderson [28]. Although useful, the algorithm has two weaknesses: (*i*) to deal the dependencies and the required complex synchronization, they came up with 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 evolving and growth model. Machta and Machta [22] described how an evolving network can be generated in parallel. Dorogovtsev et al. [10] proposed a model that can generate graphs with fat-tailed degree distributions. In this model, starting with some random graph, edges are randomly rewired according to some preferential choices.

In this paper, 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 following the preferential attachment model exactly.

The rest of the paper is organized as follows. Preliminaries, notations and a description of the parallel computation model is given in Section 2. In Section 3, we describe the problem and algorithms. Some sequential algorithms are discussed in Section 3.1. In Section 3.2, we present our parallel algorithm for distributed memory architecture for the case where each node connects a single edge to the existing network. In Section 3.3, we extend the algorithm for a general case where each node contributes x edges to the existing network. Experimental results showing the performance of our parallel algorithms are presented in Section 4. Finally, we conclude in Section 5.

2 Preliminaries and Notations

In the rest of the paper, we use the following notations. We denote a network $G(V, E)$, where V and E are the sets of vertices (nodes) and edges, respectively, with $m = |E|$ edges and $n = |V|$ vertices labeled as $0, 1, 2, \ldots, n-1$. We use the terms *node* and *vertex* interchangeably. 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 file system and they read-write data files from the same external memory. 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., 2B stands for two billion.

3 Preferential Attachment Model

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 node is added to the network and connected to some existing nodes that are chosen preferentially based on some properties of the nodes. In the most common application, preference is given to nodes with larger degrees: the higher the degree of a node, 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.

3.1 Sequential Algorithms for Preferential Attachment Model

One way to generate a random PA network is to use the Barabási–Albert (BA) model. In [3], Barabási and Albert showed many real-world networks have two important characteristics: (i) they are evolving in nature and (ii) the network tends to be scale free. They provided a model, known as the Barabási–Albert (BA) model, where a new node is connected to an existing node that is chosen with probability directly proportional to its current degree. The networks generated by BA model are called 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 node has degree d is given by $\Pr\{d\} \sim d^{-\gamma}$, where γ is a positive constant.

The BA model works as follows. Starting with a small clique of $\hat{x}$ nodes, in each phase t, a new node t is added to the network and connected to $x \leq \hat{x}$ randomly chosen existing nodes: $F_t(k)$ for $1 \leq k \leq x$ with $F_t(k) < t$; that is, $F_t(k)$ denotes the kth node 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, nodes $F_t(1), F_t(2), \ldots, F_t(x)$ are randomly selected based on the degrees of the nodes in the current network. In particular, the probability $P_t(i)$ that node t is connected to node $i < t$ is given by $P_t(i) = \frac{d_i}{\sum_j d_j}$, where d_j represents the degree of node j. Barabási and Albert showed this preferential attachment method of selecting nodes results in a power-law degree distribution [3].

In the following discussion, 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. A naive implementation of the above algorithm can take $\Omega(n^2)$ time. One naive approach is to maintain a list of the degrees of the nodes, and in each phase 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 [5] give an efficient algorithm with running time $O(m)$. This algorithm maintains a list of nodes such that each node 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 node is chosen from the list uniformly at random. Since each node i occurs exactly d_i times in the list, we have $\Pr\{F_t = i\} = \frac{d_i}{\sum_j d_j}$. Notice for the case $x > 1$, this algorithm may produce duplicate edges. To avoid duplicate edges efficiently, the algorithm requires each node to maintain separate lists of neighbors. A sequential implementation of this algorithm is given in the graph algorithm library NetworkX [16].

As it turns out none of the above algorithms lead to an efficient parallelization. Another algorithm, called *copy model*, proposed in [17] also leads to preferential attachment and power law degree distribution. The algorithm works as follows. In each phase t,

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

Step 2: then F_t is determined as follows:

$$F_t = k \text{ with prob. } p \quad (1)$$

$$= F_k \text{ with prob. } (1-p) \quad (2)$$

It can be easily shown that $\Pr\{F_t = i\} = \frac{d_i}{\sum_j d_j}$ when $p = \frac{1}{2}$. Thus when $p = \frac{1}{2}$, this algorithm follows the Barabási–Albert model as shown below. F_t can be equal to i in two mutually exclusive ways: i) i is chosen in the first step and assigned to F_t in the second step (Eq. 1); this event occurs with probability $\frac{1}{t-1} \cdot p$; or ii) a neighbor of i, $v \in \{u | F_u = i\}$ is chosen in the first step, and F_v is assigned to F_t in the second step (Eq. 2); this event occurs with probability $\frac{d_i - 1}{t-1} \cdot (1-p)$. Thus we have

$$\begin{aligned} \Pr\{F_t = i\} &= \frac{1}{t-1} \cdot p + \frac{d_i - 1}{t-1} \cdot (1-p) \\ &= \frac{p + (d_i - 1)(1-p)}{\frac{1}{2}\sum_j d_j} \end{aligned}$$

When $p = \frac{1}{2}$, we have $\Pr\{F_t = i\} = \frac{d_i}{\sum_j d_j}$.

Thus, the copy model is more general than the BA model. In [17], it has been shown that the copy model produces networks with degree distribution following a power law $d^{-\gamma}$, where the value of the exponent γ depends on the choice of p. Further, it is easy to see the running time of the copy model is $O(m)$, and we found that copy model leads to more efficient parallel algorithms for generating preferential attachment networks. We develop our parallel algorithm based on the copy model.

3.2 Parallel Algorithm for Preferential Attachment Model with $x = 1$

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 sometime 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 the ease of discussion, we first present our algorithm for the case $x = 1$. We present the general case $x \geq 1$ in Section 3.3.

Let P be the number of processors. The set of nodes V is divided into P disjoint subsets $V_1, V_2, \ldots, V_P$; 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 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. We study three partitioning approaches. In the first two approaches, consecutive nodes are assigned to a processor whereas in the last approach, nodes are assigned in a round robin fashion. The details of these partitioning schemes are given later in Section 3.5. Some network analysts may prefer to generate networks on the fly and analyze it without performing disk I/O. Many network analysis algorithms require

partitioning the graph into equal number of edges per processor. Some algorithms require the consecutive nodes to be stored in the same processor. Our different partitioning schemes can be used to satisfy many such requirements.

The basic principle behind our parallel algorithm is as follows. Recall the sequential algorithm for the copy model. Each processors P_i can independently compute step 1 for each $t \in V_i$, as a random $k \in [1, t-1]$ is chosen with uniform probability (independent of the node 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 need to be waited until F_k is known. If $k \in V_j$ where $i \neq j$, processor i sends a *request* message to processor j to find F_k. Note that at the time when processor j receives this message, F_k can still be unknown. If so, P_j keeps this message in a queue until F_k is known. Once F_k is known, P_j sends back a *resolved* message to P_i. The basic method executed by a processor P_i is given in Algorithm 3.1. An example instance of the execution of this algorithm with seven nodes is depicted in Figure 1.

Algorithm 3.1 Parallel PA with $x = 1$

```
1: Each processor P_i executes the following in parallel:

2: for each t ∈ V_i do
3:     k ← a uniform random node in [1, t − 1]
4:     c ← a uniform random number in [0, 1]
5:     if c < p (i.e., with probability p) then
6:         F_t ← k
7:     else
8:         F_t ← NILL // to be set later to F_k
9:         send message ⟨request, t, k⟩ to P_j, where k ∈ V_j

10: Next, processor P_i receives messages sent to it and processes them as follows:

11: Upon receipt of message ⟨request, t′, k′⟩ from P′_j:          ▷ note that k′ ∈ V_i
12: if  F_k′ ≠ NILL then
13:     send message ⟨resolved, t′, F_k′⟩ to P′_j
14: else
15:     store t′ in queue Q_k′

16: Upon receipt of message ⟨resolved, t, v⟩:
17: F_t ← v
18: for each t′ ∈ Q_t do
19:     send message ⟨resolved, t′, v⟩ to P_j where t′ ∈ V_j
```

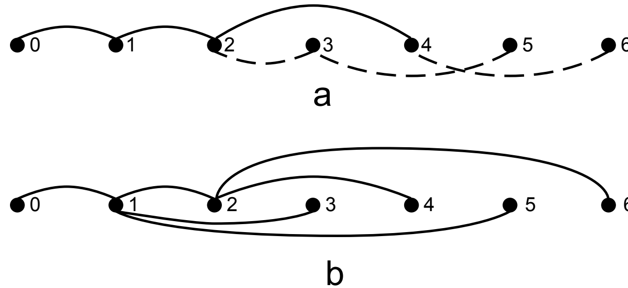


Figure 1: A network with 7 nodes generated by Algorithm 3.1: a) an intermediate instance of the network in the middle of the execution of the algorithm, b) the final network. Solid lines show final resolved edges, and dashed lines show waiting of the nodes. For example, for node $t = 4$, k is chosen to be 2, F_4 is chosen to be set to $k = 2$ (in Line 5-6), and thus edge $(4, 2)$ is finalized immediately. For node $t = 5$, k is 3 and F_5 is set to be F_3 (in Line 8); as a result, determination of F_5 is waited until F_3 is known. At the end, we have $F_5 = F_3 = F_2 = 1$.

3.3 Parallel Algorithm with $x \geq 1$

In Section 3.2, we presented the algorithm for the simpler case $x = 1$. In this section, we modify this algorithm for the general case where each node creates $x \geq 1$ edges. The pseudocode of the algorithm is given in Algorithm 3.2. The basic structure of the algorithm for the general case is the same as that of the special case $x = 1$. We mainly focus our discussion only on the modifications required and the differences between the two cases. The main difference is that, for each node 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 nodes $\{F_t(1), F_t(2), \ldots, F_t(x)\}$ is denoted by F_t.

The algorithm starts with an initial network, which is a clique of the first x nodes labeled $0, 1, 2, \ldots, x-1$. Each of the other nodes 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 $F_t(e)$ for node t where $1 \leq e \leq x$, and ii) how we avoid duplicate edge creation. Multiple edges for a node t are created by repeating the same procedure x times (Line 3), and duplicate edges are avoided by simply checking if such an edge already exists – such checking is done whenever a new edge is created.

For the e-th edge of a node t, another node k is chosen from $[x, t-1]$ uniformly at random (Line 4). Edge (t, k) is created with probability p (Line 6). However, before creating such an edge (t, k) in Line 8, the existence of such an edge is checked immediately before creating them in Line 7. If the edge already exists at that time, the process is repeated again (Line 10). With the remaining $1-p$ probability, t is connected to some node in 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. The only major change is that instead of one queue for each node, x queues are maintained for each node.

Duplicate edges can also be created during the execution of Line 23. For example, suppose node t creates two edges $(t, F_k(e))$ and $(t, F_{k'}(e'))$. 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 $F_k(e)$ and $F_{k'}(e')$. If the e-th edge of k and e'-th edge of k' both connects to the same node u, then $F_k(e) = F_{k'}(e') = u$. 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 $< resolved, t, e, v >$, the adjacency list of t is checked to find whether edge (t, v) already exists (Line 22). If the edge does not exist, it is created. Otherwise, new k and ℓ are selected (Line 27-28), and a new request message is sent (Line 29).

3.4 Dependency Chains

In our parallel algorithm, it is possible that computation of F_t for some node t can be waited until F_k for some other node 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 waited for F_3, which in turn is waited for F_2, and so on, and thus we have chain of dependency $\langle 5, 3, 2 \rangle$. If the length of these chains are very large, the waiting period for some nodes 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. In this section, we formally define a dependency chain and provide a rigorous analysis showing that maximum length of a dependency chain is at most $O(\log n)$ with high probability (w.h.p.). For a large n, $O(\log n)$ is very 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, 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 3.3. Thus, while for some nodes a processor may need to wait for $O(\log n)$ steps, the processor hardly remains idle as it has other nodes to work with.

For the purpose of analysis, first we introduce another chain named selection chain. In the first step (Line 3 of Algorithm 3.1), for each node t, another node $k \in [1, t-1]$ is selected. In turn for node k, another node in $[1, k-1]$ is selected. We can think such a selection process creates a chain called *selection chain*. Formally, we define a selection chain S_t starting at node t to be a sequence of nodes $\langle u_0, u_1, u_2, \ldots, u_i, \ldots u_x \rangle$ such that $u_0 = t, u_x = 1$, and u_{i+1} is selected for node u_i for $0 \leq i < x$. Notice that a selection chain must end at node 1. The length of a selection chain S_t denoted by $|S_t|$ is the number of nodes in S_t.

Algorithm 3.2 Parallel PA with $x \geq 1$

1: Each processor P_i executes the following in parallel:

2: **for** each $t \in V_i$ **do**
3:     **for** $e = 0$ to $x - 1$ **do**
4:         $k \leftarrow$ a uniform random node in $[x, t-1]$
5:         $c \leftarrow$ a uniform random number in $[0, 1]$
6:         **if** $c < p$ (i.e., with probability p) **then**
7:             **if** $k \notin F_t$ **then**
8:                 $F_t(e) \leftarrow k$
9:             **else**
10:                 go to line 4
11:         **else**
12:             $l \leftarrow$ a uniform random number in $[1, x]$
13:             $F_t(e) \leftarrow$ NILL // to be set later to $F_k(l)$
14:             send message $\langle request, t, e, k, l \rangle$ to P_j, where $k \in V_j$

15: Next, processor P_i receives messages sent to it and processes them as follows:

16: Upon receipt of message $\langle request, t', e', k', l' \rangle$ from P'_j: ▷ note that $k' \in V_i$
17: **if** $F_{k'}(l') \neq$ NILL **then**
18:     send message $\langle resolved, t', e', F_{k'} \rangle$ to P'_j
19: **else**
20:     store $\langle t', e' \rangle$ in queue $Q_{k',l'}$

21: Upon receipt of message $\langle resolved, t, e, v \rangle$:
22: **if** $v \notin F_t$ **then**
23:     $F_t(e) \leftarrow v$
24:     **for** each $\langle t', e' \rangle \in Q_{t,e}$ **do**
25:         send message $\langle resolved, t', e', v \rangle$ to P_j where $t' \in V_j$
26: **else**
27:     $k \leftarrow$ a uniform random node in $[x, t-1]$
28:     $l \leftarrow$ a uniform random number in $[1, x]$
29:     re-send message $\langle request, t, e, k, l \rangle$ to P_j, where $k \in V_j$

In the next step (see Eqn. 2 and Line 4-8 of Algorithm 3.1), 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 node t depends on node k. In such a case, we say node t is dependent on k; otherwise, we say node t is independent. In turn, node k can depend on some other node, and eventually such successive dependencies can form a dependency chain. Formally, a *dependency chain* D_t starting at node t is a sequence of nodes $\langle v_0, v_1, v_2, \ldots, v_i, \ldots v_y \rangle$ such that $v_0 = t$, v_i depends on v_{i+1} for $0 \leq i < y$, 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 $|D_t| \leq |S_t|$. 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 3.3. The following lemmas, Lemma 3.1 and 3.2, are needed to prove Theorem 3.3.

Lemma 3.1. *Let $P_t(i)$ be the probability that node i is in selection chain S_t starting at node t. Then for any $1 \leq i < t$, $P_t(i) = \frac{1}{i}$.*

Proof. Node i can be in S_t in two ways: a) node i is selected for t (in Line 3 of Algorithm 3.1); the probability of such an event is $\frac{1}{(t-1)}$; b) node k is selected for t, where $i < k < t$, with probability $\frac{1}{(t-1)}$, and i is in S_k.

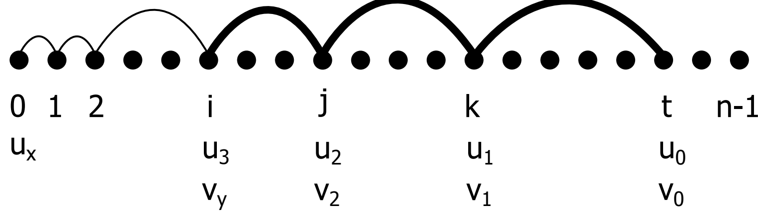


Figure 2: Selection chain and dependency chain. The entire chain, which is marked by the solid lines, is a selection chain $\langle t, k, j, i, 2, 1, 0\rangle$, and the sub-chain marked by the thick solid lines is a dependency chain $\langle t, k, j, i\rangle$.

Hence, for $1 \le i < t$, we have

$$P_t(i) = \frac{1}{t-1} + \sum_{k=i+1}^{t-1} \frac{1}{t-1} \Pr\{i \in C_k\} \tag{3}$$

$$\Rightarrow \quad (t-1)P_t(i) = 1 + \sum_{k=i+1}^{t-1} P_k(i) \tag{4}$$

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

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

By subtracting Eqn. 3 from Eqn. 5,

$$tP_{t+1}(i) - (t-1)P_t(i) = P_t(i) \tag{6}$$

$$\Rightarrow P_{t+1}(i) = P_t(i) \tag{7}$$

From Eqn. 7 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 node $i+1$; that is, $P_{i+1}(i) = \frac{1}{i}$. Hence, for any $t > i$, we have

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

□

Lemma 3.2. *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 $i_1 < i_2 < \ldots < i_k$. To prove the lemma, it is necessary and sufficient to show that for any ℓ with $2 \le \ell < t$,

$$\Pr\left\{\bigcap_{k=1}^{\ell} A_{i_k}\right\} = \prod_{k=1}^{\ell} \Pr\{A_{i_k}\}. \tag{8}$$

We know

$$\Pr\left\{\bigcap_{k=1}^{\ell} A_{i_k}\right\} = \Pr\left\{A_{i_1} \mid \bigcap_{k=2}^{\ell} A_{i_k}\right\} \cdot \Pr\left\{\bigcap_{k=2}^{\ell} A_{i_k}\right\}$$

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

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

Let R_i be a random variable that denotes the random node selected for node 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,

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

By Lemma 3.1, we have $\Pr\{i_1 \in S_{i_2}\} = \frac{1}{i_1} = \Pr\{i_1 \in S_t\} = \Pr\{A_{i_1}\}$ and thus,

$$\Pr\left\{\bigcap_{k=1}^{\ell} A_{i_k}\right\} = \Pr\{A_{i_1}\} \cdot \Pr\left\{\bigcap_{k=2}^{\ell} A_{i_k}\right\}. \tag{9}$$

Next, by using Eqn. 9 and applying induction on ℓ, we prove Eqn. 8. The base case, $\ell = 2$, follows immediately from Eqn. 9:

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

By induction hypothesis, for $\ell - 1$ events $A_{i_k}, 2 \leq k \leq \ell$, we have $\Pr\left\{\bigcap_{k=2}^{\ell} A_{i_k}\right\} = \prod_{k=2}^{\ell} \Pr\{A_{i_k}\}$. Then using Eqn. 9 for case $2 < \ell < t$, we have

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

□

Theorem 3.3. *Let L_t be the length of the dependency chain starting at node 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 nodes.*

Proof. Let S_t and D_t be the selection chain and dependency chain starting at node 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

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

By Lemma 3.2, 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 $\delta = \frac{5\log n}{\mu} - 1$. Since $\mu \leq \log n$, we have $\delta > 0$. Then,

$$\begin{aligned}
\Pr\{L \geq 5\log n\} &= \Pr\{L \geq (1+\delta)\mu\} \\
&\leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \\
&\leq \left(\frac{e}{1+\delta}\right)^{\mu(1+\delta)} \\
&\leq \left(\frac{e\mu}{5\log n}\right)^{5\log n} \\
&\leq \left(\frac{e\log n}{5\log n}\right)^{5\log n} \\
&\leq \frac{1}{n^3}
\end{aligned}$$

Thus, with probability at least $1 - \frac{1}{n^3}$, length of the dependency chain is $O(\log n)$. Using union bound, it holds simultaneously for all n nodes with probability at least $1 - \frac{1}{n^2}$. Hence, we can say, the length of the dependency chain is $O(\log n)$ w.h.p. □

3.5 Partitioning and Load Balancing

Recall the formal definition of partitioning of the set of nodes $V = \{0, 1, \ldots, n-1\}$ into P partitions $V_0, V_1, \ldots, V_{P-1}$ as described at the beginning of Section 3.2. A good load balancing is achieved by properly partitioning the set of nodes V and assigning each partition to one processor. Node 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 nodes per processor, the number of outgoing messages (request message) from a processor, and the number of incoming messages (response messages) to a processor.

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

A. For any given $k \in V$, finding of j, where $k \in V_j$ (Line 9, Algorithm 3.1), 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 nodes vary significantly, and a node with a larger degree causes more messages to work with. As a result, naive partitioning may lead to poor load balancing.

C. As we discuss later, combining multiple messages (to the same destination) and using one 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 following nodes partition schemes.

3.5.1 Consecutive Node Partitioning (CP)

In this partitioning scheme, the nodes are assigned to the processors sequentially. Partition V_i starts at nodes n_i and ends at $n_{i+1} - 1$, where $n_0 = 0$ and $n_P = n$. That is, $V_i = \{n_i, n_i + 1, \ldots, n_{i+1} - 1\}$ for all i.

With the consecutive node partitioning, the only decision to be made is the number of nodes to be assigned to each partition V_i. The simplest way to do so is uniform partitioning (UCP) where there are an equal number of nodes in each partition, i.e., $|V_i| = \left\lceil \frac{n}{P} \right\rceil$ for all i. This uniform partitioning satisfies Criterion A and C above; however, it is clear that such partitioning can lead to poor load balancing. The computation in each processor i involves the following three types of load:

A. generating random numbers and some other processing for each node $t \in V_i$,

B. sending request messages for the nodes in V_i and receiving their replies, and

C. receiving request messages from other processors and sending their replies.

The computation load for load type A and B above is directly proportional to the number of nodes in partition V_i. Computation load for load type C depends not only on the number of nodes in a processors but also on i, the rank of the processor. With uniform consecutve node partitioning (UCP), 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 Node k (see Lemma 3.4).

Lemma 3.4. *Let M_k be the number of request messages received for node k. Then $E[M_k] = (1-p)(H_{n-1} - H_k)$, where H_k is the kth harmonic number.*

Proof. Node k receives a request message from node $t > k$ 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 Node 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 computation load for each processor with an arbitrary number of nodes 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 nodes in Processor 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 3.4, the expected load for type C in Processor i is

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

The second last line follows from Eqn. 2.36 in page 41 of [15]. Thus, using another constant $b = 1 + c$, the total computation load at Processor i is

$$(n_{i+1} - n_i)(H_{n-1} + b) - (n_{i+1}H_{n_{i+1}} - n_i H_{n_i})$$

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.

$$\begin{gathered} n_0 = 0 \\ n_P = n - 1 \\ (n_{i+1} - n_i)(H_{n-1} + b) - (n_{i+1}H_{n_{i+1}} - n_i H_{n_i}) = \frac{c'n}{P} \end{gathered} \tag{10}$$

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 node partitioning (LCP). Figure 3 shows the distribution of the nodes among processors for actual solutions of Equation 10 and linear approximation. As we will see later in Section 4, our approximate scheme LCP provides a very good load balancing and performance of the algorithm.

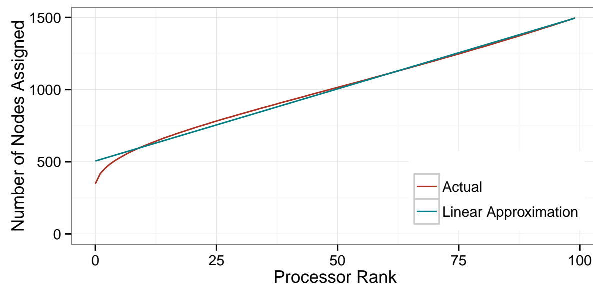


Figure 3: Distribution of the nodes among processors for actual solutions of Equation 10 and its linear approximation.

As in the LCP scheme, the number of nodes is increasing linearly with i (the ranks of the processors), the number of nodes in Processor i follows arithmetic progression a, $a+d$, $a+2d, \ldots$, $a+(i-1)d$, $\ldots$, that is, the number of nodes in Processor i is $a+(i-1)d$, where d is the slope of the line for linear approximation as shown in Figure 3. Slope d can be approximated easily by sampling two points on the actual line. How the parameter a and processor rank form a given node (Criterion A) can be computed is given in Appendix A.2.

Message Buffering: The processors exchange two types of messages: request messages and resolve messages. For each node t, a processor may need to send one request message and receive one resolve message. If a Processor i has multiple messages destined to the same processor, say Processor j, Processor 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 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.

3.5.2 Round-Robin Node Partitioning (RRP)

In this scheme, nodes are distributed in a round robin fashion among all processors. Partition V_i contains the nodes $\langle i, i+p, i+2p, \ldots, i+kp\rangle$ such that $i+kp \leq n < i+(k+1)p$; that is, $V_i = \{j | j \mod P = i\}$. In other words, node i is assigned to partition $V_{i \mod p}$. Similar to UCP, in this RRP scheme also, the number of nodes in the partitions is almost equal. The number of nodes in a partition is either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$. The difference between the number of nodes in two partitions is at most 1.

From Lemma 3.4, it is clear that the expected number of received messages decreases monotonically with increasing node labels. Round robin partition on such monotonic distribution typically performs better. For the round robin node partitioning scheme, the computation load among processors are well-balanced as shown analytically in Appendix A.3. The difference between the computation load for any two processors is at most $O(\log n)$, while the total computation load is $\Omega(n)$. RRP Scheme also satisfies Criterion A: given a node, finding the processor where the node belongs to can be computed in constant time (see Appendix A.3).

Message buffering: For consecutive node partitioning (both UCP and LCP), message buffering (combining messages) does not require any special care to avoid deadlock. In UCP and LCP, since Processor i may wait only for Processor 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 UCP 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 Experimental Results

In this section, we evaluate the performance of our algorithms. Our parallel algorithms' accuracy 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.

4.1 Experimental Setup

We used a high performance computing cluster of 48 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 the MPICH2 (version 1.7), which optimized for QLogic InfiniBand cards.

In the experiments, we used up to 768 processors. We varied n from 10^7 to 10^9 and x from 4 to 10. Each of the algorithms we considered generates the network in the main memory, and the runtime does not include the time required to write the graph into the disk.

4.2 Degree Distribution

The degree distribution of the graph generated by our parallel algorithm is shown in Figure 4 in a $\log - \log$ scale. We used $n = 10^9$ nodes each adding $x = 4$ new edges with a total of 4×10^9 edges.

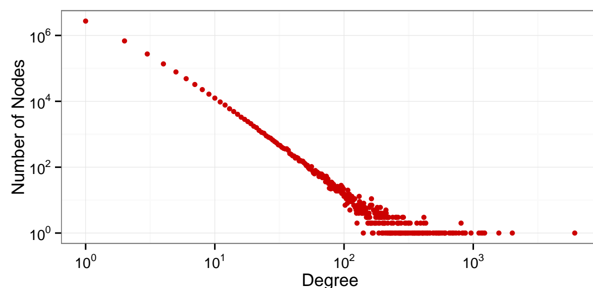


Figure 4: The degree distribution (in $\log - \log$ scale) of the network generated by our parallel algorithms. The network is generated with $n = 10^9$ and $x = 4$.

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 $2 < \gamma < \infty$ [9]. The above results show that our algorithms produce scale-free networks very accurately.

4.3 Strong Scaling

Strong scaling of a parallel algorithm shows its performance with the increasing number of processors keeping the problem size fixed. Figure 5 shows speedup factors of our algorithms with partitioning schemes uniform consecutive (UCP), linear consecutive (LCP), and round-robin partitioning (RRP) as the number of processors increases with problem size $n = 1B$ and $x = 6$. 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 BA model given in NetworkX graph algorithm library [16]. As the sequential algorithm cannot generate more than 6×10^9 edges due to memory limitation, we choose $n = 10^9$ and $x = 6$. We varied the number of processors from 1 to 768 for this experiment.

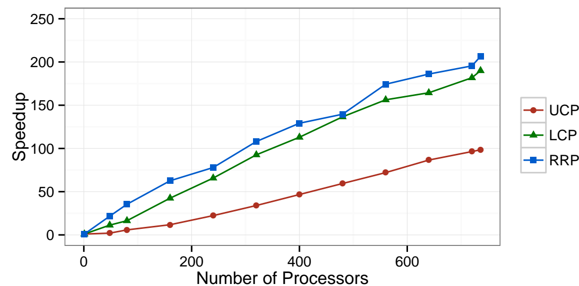


Figure 5: The strong scaling of our parallel algorithms for the problem size $n = 10^9$ and $x = 6$.

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 5 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 both LCP and RRP is better than UCP, due to better load-balancing as discussed in Section 4.6.

4.4 Weak Scaling

The weak scaling measures the performance of a parallel algorithm when the input size per processor remains constant. For this experiment, we varied the number of processors from 16 to 768. With the number of processors, the input size is also increased proportionally: for P processors, a network with $10^7 P$ edges is generated. Figure 6 shows the weak scaling of our algorithms with the increasing number of procesors.

In a perfect weak scaling case, the runtime 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 runtime. Our algorithm with load balancing schemes LCP and RRP show very good weak scaling, almost constant runtime. Again, due to poor load balancing in UCP scheme, we have worse weak scaling.

4.5 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 50 billion edges, with $n = 1B$ and $x = 50$. Using 768 processors, the generation of this network takes only 123 seconds.

4.6 Partitioning and Load Balancing

Node partitioning has significant effects on load balancing and performance of the algorithm. In Section 3.5, we have discussed three partitioning schemes UCP, 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 $n = 10^8$ nodes, $x = 10$ edges per node, and 160 processors. 160 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 nodes per processor, the number of outgoing messages from the processors, and the number of incoming messages to the processors. The results are shown in Figure 7.

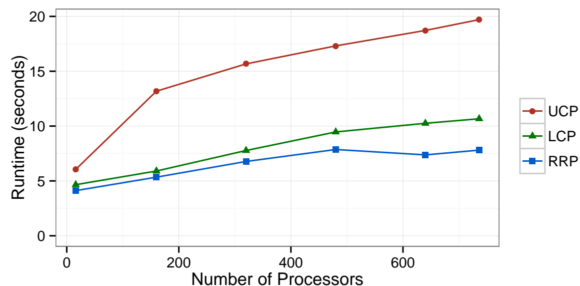
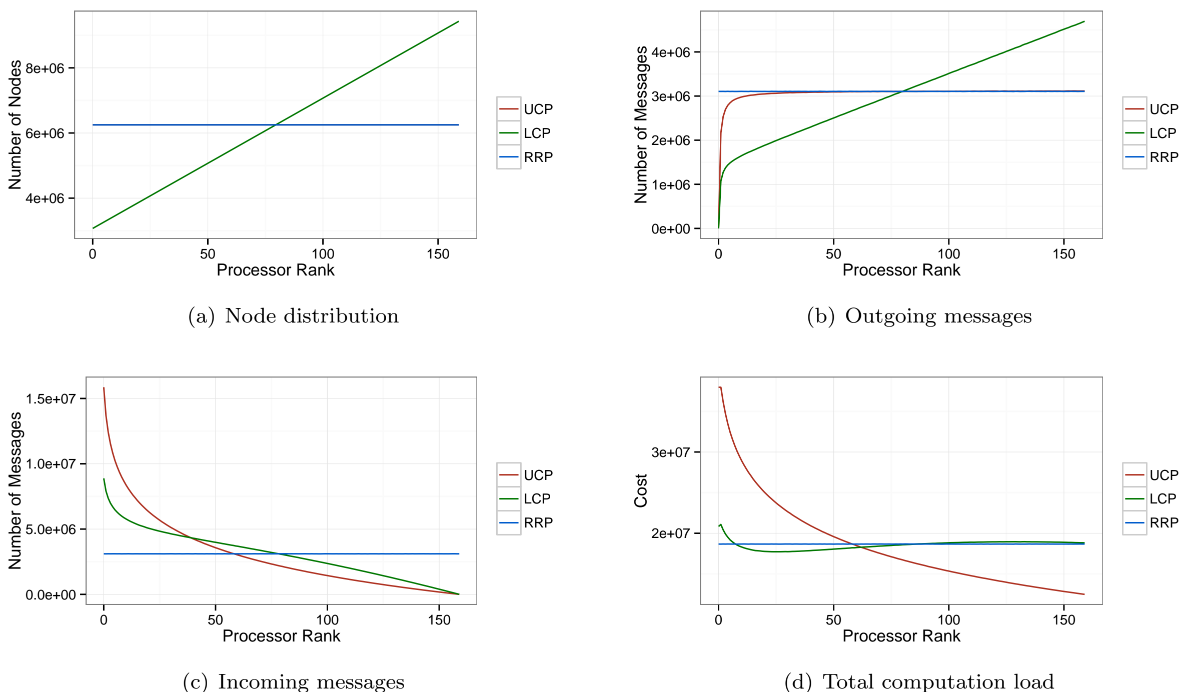


Figure 6: Weak scaling of our parallel PA algorithm.



(a) Node distribution

(b) Outgoing messages

(c) Incoming messages

(d) Total computation load

Figure 7: Node and message distribution for the three partitioning schemes: UCP, LCP, RRP

4.6.1 Node Distribution

The node distribution is shown in Figure 7(a). For UCP and RRP, nodes are distributed uniformly among the processors, and each processor has about 62,500 nodes. For LCP, the number of nodes in the processors are increasing linearly with the rank of the processors.

4.6.2 Message Distribution

In a consecutive partitioning (UCP and LCP), processor i sends outgoing request messages to processors 0 to $i-1$ and receives incoming messages from processors $i+1$ to $P-1$. For each node, a processor sends a request message with probability at most $1-p$ (see Eqn. 2). Thus, the expected number of request messages sent by a processor is proportional to the number of nodes in the processor, as shown in Figure 7(b). Note that in the UCP and LCP schemes, processor 0 does not need to send any request messages at all.

Figure 7(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 (UCP and LCP) as suggested by Lemma 3.4. In the RRP scheme, both incoming and outgoing messages are evenly

distributed among the processors.

4.6.3 Total Load Distribution

Besides sending and receiving messages, for each node, a processor can incur a constant other computation cost. Thus, for analysis purposes, we measure the total computation load of a processor as the sum of the number of nodes in the processor and the number of incoming and outgoing messages. Figure 7(d) shows the total load for the three partitioning schemes. Scheme RRP distributes the load almost perfectly among the processors. Load balancing in the LCP scheme is also quite good. On the other hand, UCP scheme distribute the load very poorly. These experimental results verify our theoretical analysis given in Section 3.5.

5 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 node 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 50 billion edges in just 123 seconds. It will be interesting to develop scalable parallel algorithms for other classes of random networks in the future.

References

[1] Reka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. *Nature*, 406(6794):378–382, July 2000. ISSN 1476-4687. doi: 10.1038/35019019. URL http://dx.doi.org/10.1038/35019019.

[2] David A. Bader and Kamesh Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 539–550, Washington, DC, USA, 2006. ISBN 0-7695-2636-5. doi: 10.1109/ICPP.2006.57. URL http://dx.doi.org/10.1109/ICPP.2006.57.

[3] Albert-László Barabási and Reka Albert. Emergence of scaling in random networks. *Science (New York, N.Y.)*, 286(5439):509–512, October 1999. ISSN 1095-9203. URL http://view.ncbi.nlm.nih.gov/pubmed/10521342.

[4] C. Barrett, S. Eubank, V.S.A. Kumar, and M. Marathe. The mathematics of networks understanding large-scale social and infrastructure networks: A simulation-based approach. *SIAM News*, 37(4), May 2004.

[5] Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):36113, 2005. doi: 10.1103/PhysRevE.71.036113.

[6] J. M. Carlson and J. Doyle. Highly optimized tolerance: a mechanism for power laws in designed systems. *Physics Review E*, 60(2):1412–1427, 1999.

[7] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Fourth SIAM International Conference on Data Mining*, April 2004.

[8] David P. Chassin and Christian Posse. Evaluating North American electric grid reliability using the Barabasi-Albert network model. *Physica A: Statistical Mechanics and its Applications*, 355(2-4):667 – 677, 2005. ISSN 0378-4371.

[9] Sergey N. Dorogovtsev and Jose F. F. Mendes. Evolution of networks. In *Advances in Physics*, pages 1079–1187, 2002.

[10] Sergey N. Dorogovtsev, Jose F. F. Mendes, and A. N. Samukhin. Principles of statistical mechanics of random networks. December 2002. URL http://arxiv.org/abs/cond-mat/0204111.

[11] Paul Erdös and Alfréd Rényi. On the evolution of random graphs. In *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, pages 17–61, 1960.

[12] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Computer Communication Review*, 29(4):251–262, August 1999.

[13] Ove Frank and David Strauss. Markov graphs. *Journal of the American Statistical Association*, 81 (395):832–842, 1986.

[14] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(12):7821–7826, June 2002.

[15] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994. ISBN 0201558025.

[16] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy)*, pages 11–15, August 2008.

[17] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Stochastic models for the web graph. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, page 57, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0850-2. URL http://dl.acm.org/citation.cfm?id=795666.796570.

[18] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 591–600, 2010.

[19] V. Latora and M. Marchiori. Vulnerability and protection of infrastructure networks. *Physical Review E*, 71(1), 2005.

[20] Jure Leskovec and Eric Horvitz. Planetary-scale views on a large instant-messaging network. In *Proceedings of the 17th International Conference on World Wide Web (WWW)*, 2008.

[21] Jurij Leskovec. *Dynamics of large networks*. PhD thesis, Pittsburgh, PA, USA, 2008. AAI3340652.

[22] Benjamin Machta and Jonathan Machta. Parallel dynamics and computational complexity of network growth models. *Physical Review E*, 71:026704, Feb 2005. doi: 10.1103/PhysRevE.71.026704. URL http://link.aps.org/doi/10.1103/PhysRevE.71.026704.

[23] Joel Miller and Aric Hagberg. Efficient generation of networks with given expected degrees. In *Proceedings of Algorithms and Models for the Web-Graph (WAW)*, pages 115–126, 2011.

[24] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT/ICDT)*, pages 331–342, 2011. ISBN 978-1-4503-0528-0. doi: 10.1145/1951365.1951406. URL http://doi.acm.org/10.1145/1951365.1951406.

[25] Garry Robins, Pip Pattison, Yuval Kalish, and Dean Lusher. An introduction to exponential random graph (p*) models for social networks social networks. *Social Networks*, 29(2):173–191, May 2007.

[26] Georgos Siganos, Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. Power laws and the as-level internet topology. *IEEE/ACM Transactions on Networking*, 11(4):514–524, Aug 2003.

[27] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393 (6684):440–442, June 1998. ISSN 0028-0836. doi: 10.1038/30918. URL http://dx.doi.org/10.1038/30918.

[28] Andy Yoo and Keith Henderson. Parallel Generation of Massive Scale-Free Graphs. March 2010. URL http://arxiv.org/abs/1003.3684.

A Partitioning Scheme

In this section, we describe some additional details of the partitioning schemes. For a given n nodes and P processors, we create P partitions $V_0, V_1, \ldots V_{P-1}$. The processors are labeled as $0, 1, \ldots P-1$. Partition V_i is processed by processor i. For each scheme, we need to know *i*) the sizes of the partitions, *ii*) the set of nodes in each partition, and *iii*) for a given a node u, find the partition number to which u belongs. Below we show how to calculate them for the three partitioning schemes.

A.1 Uniform Consecutive Partitioning (UCP)

UCP is the simplest among the partitioning schemes. Consecutive and equal number of nodes are assigned to the partitions.

Partition size: The sizes of the partitions are almost equal. Let $B = \lceil \frac{n}{P} \rceil$. Then, the size of a partition is either B or $B-1$.

Partition range: Partition i includes the nodes from iB to $(i+1)B-1$.

Finding processor rank: For a node $u \in V_i$, the rank of the processor i is given by $i = \lfloor \frac{u}{B} \rfloor$.

A.2 Linear Consecutive Partitioning (LCP)

In this scheme, consecutive nodes are assigned to the partitions, and the number of nodes in the partitions are modeled by arithmetic progression $a, a+d, a+2d, \ldots, a+(P-1)d$.

Partition size: The number of nodes in partition V_i is given by $B_i = a + id$, where a and d are partition parameters. At the end of this section, we discuss how a and d are determined.

Partition range: Partition i has the nodes 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 processor rank: Given a node u, we need to find the partition i such that $u \in V_i$. Node u satisfies the following inequality:

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

$$\frac{i\,(2a+(i-1)d)}{2} \leq u < \frac{(i+1)\,(2a+id)}{2} \tag{11}$$

Solving Inequality 11, we have

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

Determining partition parameters a and d: The parameters a and d are determined using the number of nodes 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 nodes 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 Eqn. 10, we can compute n_1 and n_{P-1}. Then, the number of nodes in the first processor is $n_1 - n_0 = n_1$ and the number of nodes 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

$$\begin{aligned}
& \sum_{j=0}^{P-1}(a+jd) = n \\
\Rightarrow\quad & \frac{P\,(2a+(P-1)d)}{2} = n \\
\Rightarrow\quad & a = \frac{n}{P} - \frac{(P-1)d}{2} \qquad (12)
\end{aligned}$$

A.3 Round Robin Partitioning

Partition size: As the nodes are distributed in round robin fashion to the partitions, the number of nodes is almost equal for all partitions. Similar to UCP scheme, the size of a partition is either $\lceil \frac{n}{P} \rceil$ or $\lceil \frac{n}{P} \rceil - 1$.

Partition range: For round robin partitioning, partition i has the nodes $i, i+P, i+2P, \ldots i+(B-1)P \leq n$.

Finding processor rank: For a given node $u \in V_i$, partition i is determined by $i = u \mod P$.

Computation load: The expected number of request messages received for node k is $(H_{n-1} - H_k)$ (see Lemma 3.4). Other loads for any node is constant. Then the total load for node k is $CL(k) = (H_{n-1} - H_k) + b$, for some constant b. Thus, the total load for Processor 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

$$\begin{aligned}
& PL(0) - PL(P-1) \\
= \quad & \sum_{k \in V_0}(H_{n-1} - H_k + b) - \sum_{k \in V_{P-1}}(H_{n-1} - H_k + b) \\
\leq \quad & (H_{n-1} + b)(|V_0| - |V_{P-1}|) - \sum_{k \in V_0} H_k + \sum_{k \in V_{P-1}} H_k
\end{aligned}$$

If n is a multiple of P, we have

$$\begin{aligned}
& |V_0| - |V_{P-1}| = 0, \\
& \sum_{k \in V_{P-1}} H_k < \sum_{k \in V_0} H_k + H_n, \\
\text{and thus,} \quad & PL(0) - PL(P-1) < H_n = O(\log n).
\end{aligned}$$

Otherwise,

$$\begin{aligned}
& |V_0| - |V_{P-1}| = 1, \\
& \sum_{k \in V_{P-1}} H_k \leq \sum_{k \in V_0} H_k, \\
\text{and thus,} \quad & PL(0) - PL(P-1) \leq H_{n-1} + b = O(\log n).
\end{aligned}$$