NDSSL Technical Report 13-062

May 07, 2013

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

Authors: Maksudul Alam
Maleq Khan
Madhav V. Marathe

Acknowledgements: We thank our external collaborators and members of the Network Dynamics and Simulation Science Laboratory (NDSSL) for their suggestions and comments. This work has been partially supported by DTRA CNIMS Contract HDTRA1-11-D-0016-0001, NSF PetaApps Grant OCI-0904844, NSF NetSE Grant CNS-1011769, and NSF SDCI Grant OCI-1032677.

Network Dynamics and Simulation Science Laboratory
Virginia Bioinformatics Institute
Virginia Polytechnic Institute and State University

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

Maksudul Alam*†, Maleq Khan†, and Madhav V. Marathe*†
*Department of Computer Science
†Network Dynamics and Simulation Science Laboratory
Virginia Tech, Blacksburg, VA 24061, USA
{maksud, maleq, mmarathe}@vbi.vt.edu

ABSTRACT

Various random networks are being widely used in modeling and analyzing complex systems. As the complex systems are growing larger, generation of random networks with billions of nodes or larger became a necessity. Generation of such massive networks requires efficient and parallel algorithms. 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. Our algorithms can generate scale-free networks with 50 billion edges in 123 seconds using 768 processors.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; G.2.2 [**Discrete Mathematics**]: Graph Theory

General Terms

Theory, Algorithm

1. INTRODUCTION

The huge improvement of hardware technology as well as the developments in software and algorithms have enabled the detailed study of complex systems. At the same time, advancement of various technologies is also causing a rapid growth of the complex systems. These complex systems, such as the Internet [11, 25], biological networks [13], social networks [19, 17], and various infrastructure networks [4, 18, 7] are sometimes modeled by various random graphs for the purpose of studying their behavior. The study of these complex systems have significantly increased the interest in various random graph models [5]. As the complex systems are growing larger and larger, it requires generations of massive random networks efficiently. Analyzing a very large complex system using a smaller model may not produce accurate results. As the interactions in a larger network lead to complex collective behavior [20], a smaller network may not exhibit the same behavior, even if both networks are generated using the same model. In [20], 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]. Chassin and Posse [7] showed how Barabasi-Albert model can be used for evaluating the North American electric grid with high reliability.

Many random graph models have been developed in the past. Among them, the first and well-studied model is the Erdos-Renyi model [10]. However, the Erdos-Renyi 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 [26], Barabasi-Albert [3, 1], Chung-Lu [22], exponential random graph [12, 24], and HOT[6] models, have been proposed.

Barabasi 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. [7]

Watts and Strogatz [26] described small-world networks, which also lead to relatively homogenous topology. [7] This model transforms a regular one-dimensional lattice (with vertex degree of four or higher) by rewiring each edge, with

*We thank our external collaborators and members of the Network Dynamics and Simulation Science Laboratory (NDSSL) for their suggestions and comments. This work has been partially supported by DTRA CNIMS Contract HDTRA1-11-D-0016-0001, NSF PetaApps Grant OCI-0904844, NSF NetSE Grant CNS-1011769, and NSF SDCI Grant OCI-1032677.

.

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. [27]

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." Toward meeting this goal, recently some efficient sequential algorithms have been developed for some random graph models: Erdos-Renyi [5], small world [5], Preferential Attachment [5, 23], Chung-Lu [22].

However, these efficient sequential algorithms are able to generate networks with millions of nodes quickly, but generating networks with billions of nodes can take an undesirably longer time. Further, a large memory requirement may even prohibit the generations of such large networks using these sequential algorithms. Thus, distributed-memory parallel algorithms are now desirable in dealing with these problems. Note shared-memory parallel algorithms also suffer from the memory restriction as these algorithms use memory of a single compute machine. Further, current shared-memory architectures are limited to only a few parallel processors whereas distributed-memory parallel systems are available with hundreds or thousands of processors. Until very recently researcher did not pay any attention in developing parallel algorithms for generating random graphs. Parallelization of these sequential algorithms for distributed-memory systems poses two main challenges as follows. Firstly, the dependencies among the edges, especially in the preferential-attachment model, impede independent operations of the processors. Secondly, 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 Erdos-Renyi 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 is given in [23].

For the preferential attachment model, the only previously known distributed-memory parallel algorithm is given by Yoo and Henderson [27]. This algorithm has several shortcomings: 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 the running the algorithm repeatedly, which significantly limits the usefulness of this algorithm. Several other studies was done on generating a large Barabasi-Albert model in parallel. In article [21] the authors describe how an evolving network can be generated in parallel. Dorogovtsev et al. [9] 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.

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. In this paper we study the problem of generating a massive scale-free network based on the preferential attachment (PA) model using a parallel algorithm.

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 the serial version of the algorithm. In Section 3.2 we provide our parallel solution 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 their seminal paper [3] Barabasi 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 follow. 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: $F_k(t)$ for $1 \leq k \leq x$ with $F_k(t) < t$; that is, $F_k(t)$ denotes the kth node which t is connected to. Thus each phase adds x new edges $(t, F_1(t)), (t, F_2(t)), \ldots, (t, F_x(t))$ to the network, which exhibits the evolving nature of the model. For each of the x new edges, nodes $F_1(t), F_2(t), \ldots, F_x(t)$ 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. Barabasi 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 $F_1(t)$. 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 [15].

As it turns out none of the above algorithms lead to an efficient parallelization. Another algorithm called *copy model* proposed in [16] 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 Barabasi-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

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

When $p = \frac{1}{2}$, we have $\Pr\{F(t) = i\} = \frac{d_i}{\sum_j d_j}$.

Thus, copy model is more general than BA model. In [16], 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 an MPI-based parallel system. 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 discuss the general case $x \geq 1$ in Section 3.3.

Let P be the number of processors. The set of nodes V are 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 depends on how V is partitioned. We study two partitioning approaches. In the first approach consecutive nodes are assigned to a processor whereas in the second approach nodes are assigned in a round robin fashion. The details of these partitioning schemes are given later in Section 3.5.

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 even 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 $t \in V_i$ **do**
3: $k \leftarrow$ a uniform random node in $[1, t-1]$
4: $c \leftarrow$ a uniform random number in $[0, 1]$
5: **if** $c < p$ (i.e., with probability p) **then**
6: $F(t) \leftarrow k$
7: **else**
8: $F(t) \leftarrow$ NILL // to be set later to $F(k)$
9: send message $\langle request, t, k \rangle$ to P_j, where $k \in V_j$

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

11: Upon receipt of message $\langle request, t', k' \rangle$ from P'_j: note that $k' \in V_i$
12: **if** $F(k') \neq$ NILL **then**
13: send message $\langle resolved, t', F(k') \rangle$ to P'_j
14: **else**
15: store t' in queue $Q_{k'}$

16: Upon receipt of message $\langle resolved, t, v \rangle$:
17: $F(t) \leftarrow v$
18: **for** each $t' \in Q_t$ **do**
19: send message $\langle resolved, t', v \rangle$ to P_j where $t' \in V_j$

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 is as 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 $(t, F_1(t)), (t, F_2(t)), \ldots, (t, F_x(t))$, and make sure such edges are distinct and do not create any parallel edges. For this general case, the set of nodes $\{F_1(t), F_2(t), \ldots, F_x(t)\}$ 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 $F_e(t)$ 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.

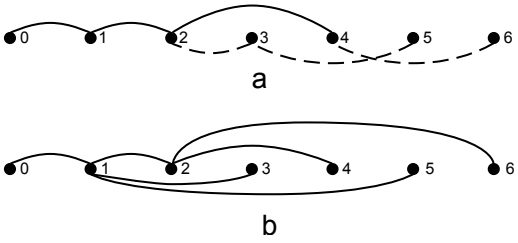


Figure 1: A network with 7 nodes generated by Algorithm 3.1: a) an intermediate instance of the network in the middle of the execution of the algorithm, b) the final network. Solid lines show final resolved edges, and dashed lines show waiting of the nodes. For example, for node $t = 4$, k is chosen 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 $F(5) = F(3) = F(2) = 1$.

For the eth 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 $(t, F_\ell(k))$, 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 $F_\ell(k)$ (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 $(t, F_e(k))$ and $(t, F_{e'}(k'))$. 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 $F_e(k)$ and $F_{e'}(k')$. If the e-th edge of k and e'-th edge of k' both connects to the same node u, then $F_e(k) = F_{e'}(k') = u$. Hence, t may create a duplicate edge (t, u) which could not be detected early. To deal with such duplicate edges, after we receive a resolved message $< resolved, t, e, v >$, we first check the adjacency list of t to see if the 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

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: $F_e(t) \leftarrow k$
9: **else**
10: go to line 4
11: **else**
12: $l \leftarrow$ a uniform random number in $[1, x]$
13: $F_e(t) \leftarrow$ NILL // to be set later to $F_l(k)$
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** $F_{l'}(k') \neq$ 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: $F_e(t) \leftarrow v$
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 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 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 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.

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, 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_x is independent. Notice if $v_i \in D_t$, D_{v_i} is a subsequence and a suffix of D_t. Also it is easy to see 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 Theorem 3.3.

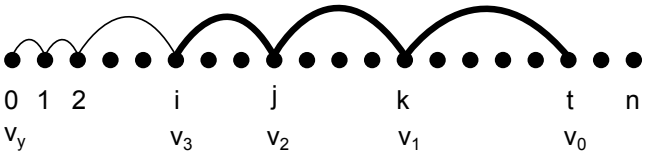


Figure 2: Selection chain and dependency chain. The entire chain, which is marked by the solid lines, is a selection chain $\langle v_0, v_1, v_2, v_3, 2, 1, 0\rangle$, and the subchain marked by the thick solid lines is a dependency chain $\langle v_0, v_1, v_2, v_3\rangle$.

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. Hence, for $1 \leq i < t$, we have

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

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

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

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

By subtracting Eqn. 3 from Eqn. 5,

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

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

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 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 denote the random node selected for node i. Now observe 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,

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

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, 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 $A_{i_k}, 2 \le k \le \ell$, 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 $E[L_t] \le \log n$ 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

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

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

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

By Lemma 3.2, the random variables $X_t(i)$, for $1 \le i < t$, are mutually independent. Applying the Chernoff bound on independent Poisson trials, we have

$$\Pr\left\{\sum_t X_t(i) \ge (1+\delta)\mu\right\} \le \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu$$

In the Chernoff bound, we set $\delta = \frac{5 \log n}{\mu} - 1$. Since $\mu \le \log n$, we have $\delta > 0$. Then,

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

Thus, with probability at least $1 - \frac{1}{n^3}$, the length of 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 satisfy 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 a equal number of nodes in each partition, i.e., $|V_i| = \lceil \frac{n}{P} \rceil$ for all i. This uniform partitioning satisfies Criterion A and C above; however, it is clear 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 Step 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, 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

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

The second last line follows from (Equation 2.36 in page 41 of [14]).

Thus, using another constant $b = 1 + c$, the total computation load at Processor i is

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

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.

$$\begin{aligned}
n_0 &= 0 \\
n_P &= n - 1 \\
(n_{i+1} - n_i)(H_{n-1} + b) - (n_{i+1}H_{n_{i+1}} - n_iH_{n_i}) &= \tfrac{c'n}{P} \quad (10)
\end{aligned}$$

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 very good load balancing and performance of the algorithm.

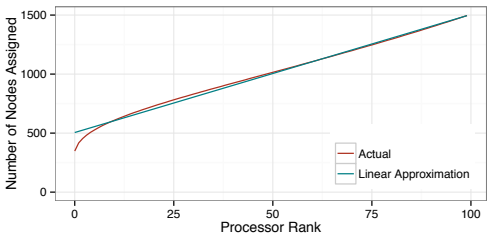


Figure 3: Distribution of the nodes among processors for actual solutions of Equation 10 and its linear approximation.

As in the LCP scheme the number of nodes is increasing linearly with i, the rank of the processor, 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: In our algorithm, 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 single message by buffering them instead of sending them individually. Each processor 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 large number of messages at a time limiting our ability to generate 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.

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 be arisen.

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 need 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 of the resolved messages cannot wait any longer. Otherwise, it can cause a circular waiting among the processors leading to deadlock situation.

4. EXPERIMENTAL RESULTS

In this section, we evaluate our algorithm and its performance by experimental analysis. We demonstrate the accuracy of our algorithm by showing that our algorithm produces networks with power law degree distribution as desired. We present the strong and weak scaling of our algorithm and show that our algorithm scales very well with the number of processors. We also present experimental results evaluating the impact of load balancing on the performance of our algorithm.

4.1 Experimental Setup

4.1.1 *Hardware*

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 processor (16 cores per node) and 64GB of 1600MHz DDR3 RAM. The nodes are interconnected by QLogic QDR InfiniBand interconnects.

4.1.2 *Software*

For the MPI based implementation of our algorithm, we used the MPICH2 implementation (version 1.7) optimized for QLogic InfiniBand cards.

4.2 Degree Distribution

The degree distribution of the network generated by our algorithm is shown in Figure 4 in a $\log-\log$ scale. We use $n = 10^9$ nodes each generating $x = 4$ new edges with a total of 4×10^9 edges. The distribution is heavy tailed, which is a distinct feature of the power-law networks. The exponent γ of the power-law degree distribution is measured to be 2.7. This supports the fact that for the finite average degree of a scale-free network the exponent should be $2 < \gamma < \infty$ [8].

4.3 Strong Scaling

We discuss the scalability of our algorithm with a fixed problem size. We use the run-time of the serial version of the algorithm as the base-line for calculating speedup. As the serial algorithm cannot generate more than 6×10^9 edges, we choose $n = 10^9$ nodes each contributing $x = 6$ new edges as our fixed problem size. We have used the

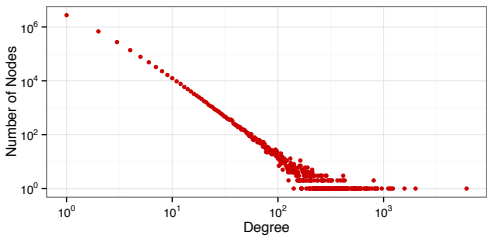


Figure 4: The degree distribution of the BA Network. In $\log - \log$ scale the degree distribution is a straight line.

uniform consecutive (UCP), linear consecutive (LCP), and round-robin partitioning (RRP) version of our algorithm. We varied the number of processors from 1 to 736 for this experiment. The strong scaling is shown in Figure 5. Parallelization of network algorithms are notoriously hard. We have already shown the problem is heavily sequential in nature due to dependencies. Given the difficulties of the problem, we get a very good speed-up. Further the speed-up factor increases linearly with the number of processors. As the computational loads are well balanced in RRP and LCP they provide much better performance than UCP.

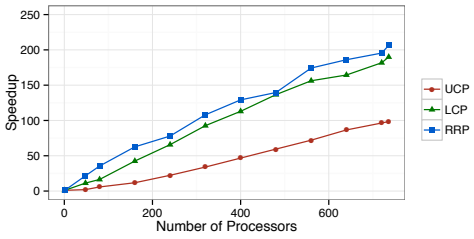


Figure 5: The strong scaling of parallel Barabasi-Albert algorithm.

4.4 Weak Scaling

The weak scaling shows how an algorithm scales when the input size per processors is constant. Figure 6 shows the weak scaling of our algorithm. The number of edges of the generated graphs in increased with the rate of 10^7 edges per processor. We vary the number of processors from 16 to 736 as a multiple of 16. Our algorithm has very good weak scaling, as run-time remains almost constant.

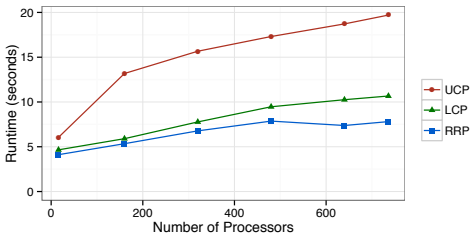


Figure 6: Weak scaling of parallel BA algorithm.

4.4.1 *Generating Large Networks*

Our main objective for designing this algorithm is to generate very large networks. Using our algorithm we are able to generate a network with 50 billion edges. Using 768 processors, the generation of this network takes only 123 seconds.

4.5 Partitioning and Load Balancing

We have discussed some aspects of a partitioning scheme earlier. Partitioning has significant effects on the performance of the system. Load balancing is also an important performance measure. In this experiment we examine the underlying relationship between the partitioning scheme and load balancing. In our algorithm, we measure the computational load in terms of the number of nodes per processor, the number of outgoing messages from a processor, and the number of incoming messages to a processor. In this section we show three partitioning schemes and how it affects the load and performance of the system.

For all of the experiments we used $n = 10^8$ nodes and $x = 10$ edges per node. We use the three partitioning algorithms here: Uniform Consecutive Partitioning (UCP), Linear Consecutive Partitioning (LCP), and Round Robin Partitioning (RRP). The results are shown in Figure 7.

4.5.1 *Node Distribution*

The node distribution is shown in Figure 7(a). For both UCP and RRP, nodes are distributed uniformly among the processors. For LCP, nodes are distributed linearly.

4.5.2 *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 processor $i+1$ to $P-1$. In the copy model, about 1/2 of the edges connect to near-end and require no message sending. And the remaining half of the edges require sending messages to other processors. Hence, the expected number of request messages is proportional to the number of nodes, as seen from the Figure 7(b).

Again, it is clear that lower ranked processors receives more messages than the higher ranked processors. This is due to the fact that lower ranked processors receive messages from a greater number of processors than higher ranked processors as shown in Lemma 3.4. This is shown in Figure 7(c).

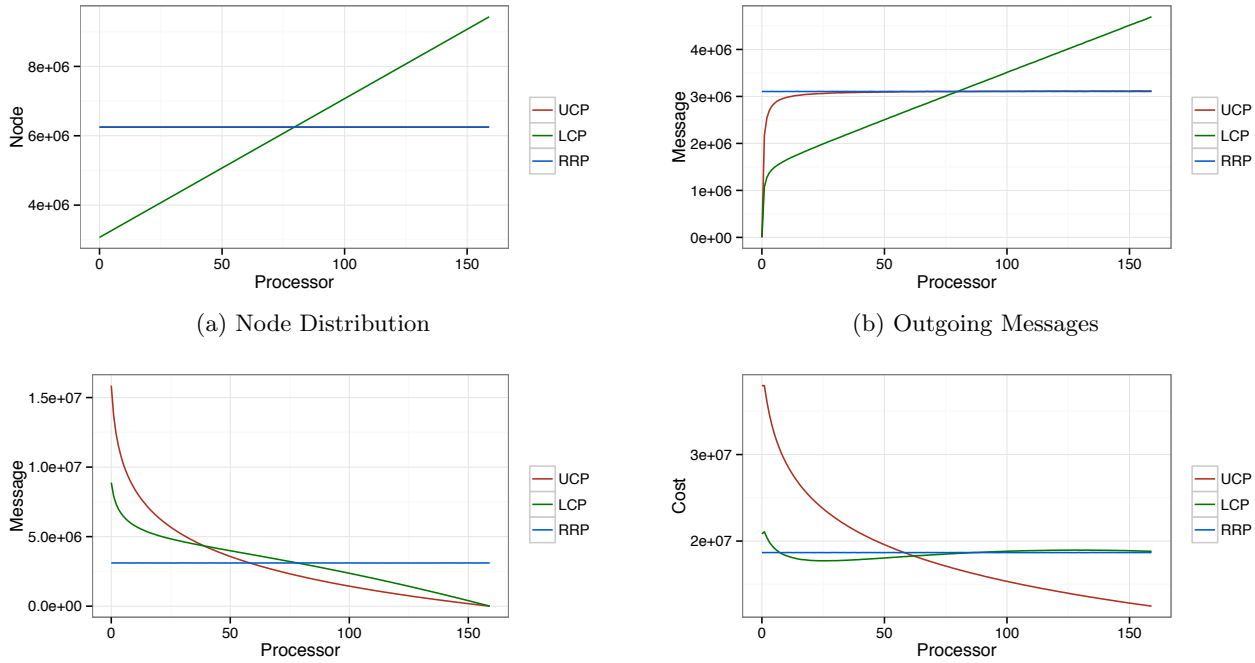
In RRP, both the number of request and response messages are similar, hence it results in a uniform number of messages per processor.

4.5.3 *Cost Distribution*

We have defined the cost in Equation 10. Here, we plot the cost function for all three partitioning algorithm. As evident from the Figure 7(d), RRP produces uniform cost in all the processors. UCP does not result in uniform cost. The tuned LCP produces almost uniform cost among all the processors. This verifies our analysis of the partitioning algorithm.

5. CONCLUSION

We have developed a parallel algorithm to generate massive scale-free networks using the preferential attachment model. We have analyzed the dependency nature of the problem in detail. Such analysis led us to the development of



(a) Node Distribution

(b) Outgoing Messages

(c) Incoming Messages

(d) Cost

Figure 7: Node and Message distribution for various partitioning schemes

an efficient parallel algorithm for the problem. We have discussed various node partitioning scheme and its effect on the algorithm. Our algorithm produces networks which strictly follow power-law distribution. The linear scalability of our algorithm enables us to produce 50 billion nodes in just 123 seconds. Power-law graphs are very crucial for the understanding of many real-world networks, and hence we believe our algorithm will be very useful in analyzing and studying massive scale real-world networks.

References

[1] Reka Albert, Hawoong Jeong, and Albert-Laszlo Barabasi. 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 2006 International Conference on Parallel Processing*, ICPP '06, pages 539–550, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2636-5. doi: 10.1109/ICPP.2006.57. URL `http://dx.doi.org/10.1109/ICPP.2006.57`.

[3] Albert-Laszlo Barabasi 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):1412ï£¡1427, 1999.

[7] 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.

[8] Sergey N. Dorogovtsev and Jose F. F. Mendes. Evolution of networks. In *Adv. Phys*, pages 1079–1187, 2002.

[9] 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`.

[10] 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.

[11] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29(4): 251–262, August 1999.

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

[13] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proc. Natl. Acad. of Sci. USA*, 99(12):7821–7826, June 2002.

[14] 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.

[15] 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 (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[16] 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 '00, pages 57–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0850-2. URL `http://dl.acm.org/citation.cfm?id=795666.796570`.

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

[18] V. Latora and M. Marchiori. Vulnerability and protection of infrastructure networks. *Phys. Rev. E*, 71(1), 2005.

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

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

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

[22] 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.

[23] 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 '11, pages 331–342, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0528-0. doi: 10.1145/1951365.1951406. URL `http://doi.acm.org/10.1145/1951365.1951406`.

[24] 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):173ï£¡191, May 2007.

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

[26] 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`.

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

APPENDIX

A. PARTITIONING SCHEME

Here, we describe in detail how the partitioning scheme is done in our paper. For a given n nodes and p processors, we create p partitions $V_0, V_1, \ldots V_{p-1}$. Processors are also labeled as $0, 1, \ldots p-1$. Partition V_i is processed by processor i. For each scheme, we need to know i) the size of the partition, ii) the set of nodes in the partition, and iii) given a node u, find the partition number to which the node belongs. We show how we calculate them for the following schemes.

A.1 Uniform Consecutive Partitioning

It is the easiest of the partitioning scheme. Here all the nodes are partitioned sequentially and equally.

Partition size: Each partition has an equal number of nodes, $B = \left\lceil \frac{n}{p} \right\rceil$.

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

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

A.2 Linear Consecutive Partitioning

In this partitioning scheme, the number of nodes per partition is modeled with *arithmetic progression* $(a, a+d, a+2d, \ldots, a+(p-1)d)$.

Partition size: The number of nodes in partition V_i is given as $B_i = a+id$, where a and d are partitioning variables.

Partition range: Partition i has the nodes from $\sum_{j=0}^{i-1}(a+jd) = i\frac{(2a+(i-1)d)}{2}$ to $\sum_{j=0}^{i}(a+jd) = (i+1)\frac{(2a+id)}{2} - 1$.

Finding processor rank: Given a node u we need to find the partition V_i it belongs to. According to the scheme, the node u satisfies the following inequality:

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

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

Taking only the left part of the equation 11:

$$\begin{aligned} \frac{i\,(2a+(i-1)d)}{2} &\leq u \\ di^2 + (2a-d)i - 2u &\leq 0 \end{aligned} \qquad (12)$$

This inequality has two solutions: $i_1 = \frac{-(2a-d)-\sqrt{(2a-d)^2+8du}}{2d}$ and $i_2 = \frac{-(2a-d)+\sqrt{(2a-d)^2+8du}}{2d}$. Though both i_1 and i_2

satisfies equation 12, only i_2 satisfies the right side of equation 11. Also note $(2a-d)^2 + 8du \geq 0$ as $d, u \geq 0$, hence $i_1 \leq 0$. Thus, u belongs to the partition number $i = \lfloor i_2 \rfloor = \left\lfloor \frac{-(2a-d)+\sqrt{(2a-d)^2+8du}}{2d} \right\rfloor$.

Determining partition parameters: The two partition parameters a and d are closely related with the number of nodes n and the number of processors p. According to the scheme, the following equation is satisfied:

$$\begin{aligned} \sum_{j=0}^{p-1}(a+jd) &= n \\ \frac{p\,(2a+(p-1)d)}{2} &= n \\ a &= \frac{n}{p} - \frac{(p-1)d}{2} \end{aligned} \tag{13}$$

Thus given a parameter value d, we can calculate a using equation 13.

A.3 Round Robin Partitioning

Partition size: As the nodes are distributed in round robin fashion to all the partitions, the number of nodes is equal for all as given by $B = \frac{n}{p}$.

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: It is very easy to determine the partition number i for node u as: $i = u \mod p$.

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 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 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

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

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,

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