Parallel Algorithms for Generating Random Networks with Given Degree Sequences

Maksudul Alam*† maksud@vbi.vt.edu

Maleq Khan† maleq@vbi.vt.edu

Accepted: May 22, 2015

Abstract

Random networks are widely used for modeling and analyzing complex processes. Many mathematical models have been proposed to capture diverse real-world networks. One of the most important aspects of these models is degree distribution. Chung–Lu (CL) model is a random network model, which can produce networks with any given arbitrary degree distribution. The complex systems we deal with nowadays are growing larger and more diverse than ever. Generating random networks with any given degree distribution consisting of billions of nodes and edges or more has become a necessity, which requires efficient and parallel algorithms. We present an MPI-based distributed memory parallel algorithm for generating massive random networks using CL model, which takes $O\left(\frac{m+n}{P} + P\right)$ time with high probability and $O(n)$ space per processor, where n, m, and P are the number of nodes, edges and processors, respectively. The time efficiency is achieved by using a novel load-balancing algorithm. Our algorithms scale very well to a large number of processors and can generate massive power–law networks with one billion nodes and 250 billion edges in one minute using 1024 processors.

Keywords. Massive Networks, Parallel Algorithms, Network Generator

1 Introduction

The advancements of modern technologies are causing a rapid growth of complex systems. These systems, such as the Internet [22], biological networks [9], social networks [24, 25], and various infrastructure networks [6, 11] are sometimes modeled by 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 such as Erdős–Rényi (ER) [8], small-world [23], Barabási–Albert (BA) [1], Chung-Lu (CL) [7], HOT [4], exponential random graph (ERGM) [20], recursive matrix (R-MAT) [5], and stochastic Kronecker graph (SKG) [13, 14] models. Among those models, the SKG model has been included in Graph500 supercomputer benchmark [10] due to its simple parallel implementation. The CL model exhibits similar properties of the SKG model and further has the ability to generate a wider range of degree distributions [19]. To the best of our knowledge, there is no parallel algorithm for the CL model.

Analyzing a very large complex system requires generating massive random networks efficiently. As the interactions in a larger network lead to complex collective behavior, a smaller network may not exhibit the same behavior, even if both networks are generated using the same model. In [12], by experimental analysis, it was shown that the structure of larger networks is fundamentally different from small networks and many patterns emerge only in massive datasets. Demand for large random networks necessitates efficient algorithms to generate such networks. However, even efficient sequential algorithms for generating such graphs were nonexistent until recently. Sequential algorithms are sometimes acceptable in network analysis with tens of thousands of nodes, but they are not appropriate for generating large graphs [3]. Although, recently some efficient sequential algorithms have been developed [3, 5, 13, 16], these algorithms can generate networks

*Department of Computer Science, Virginia Tech

†Network Dynamics and Simulation Science Laboratory, Virginia Bioinformatics Institute

with only millions of nodes in a reasonable time. But, generating networks with billions of nodes can take an undesirably long time. Thus, efficient parallel algorithms that scale to large number of processors are desirable in dealing with these problems.

In this paper, we present a time-efficient MPI–based distributed memory parallel algorithm for generating random networks from a given sequence of expected degrees using the CL model. Please note that this algorithm can easily be adapted for shared-memory parallel systems. To the best of our knowledge, it is the first parallel algorithm for the CL model. The most challenging part of this algorithm is load-balancing. Partitioning the nodes with a balanced computational load is a non trivial problem. In a sequential setting, many algorithms for the load-balancing problem were studied [15, 17, 18]. Some of them are exact and some are approximate. These algorithms use many different techniques such as heuristic, iterative refinement, dynamic programming, and parametric search. All of these algorithms require at least $\Omega(n + P \log n)$ time, where n, P are the number of nodes and processors respectively. To the best of our knowledge, there is no parallel algorithm for this problem. In this paper, we present a novel and efficient parallel algorithm for computing the balanced partitions in $O\left(\frac{n}{P} + P\right)$ time. The parallel algorithm for load balancing can be of independent interest and probably could be used in many other problems. Using this load balancing algorithm, the parallel algorithm for the CL model takes an overall runtime of $O\left(\frac{n+m}{P} + P\right)$ with high probability (w.h.p.). The algorithm requires $O(n)$ space per processor. Our algorithm scales very well to a large number of processors and can generate a power-law network with one billion nodes and 250 billion edges in memory in less than a minute using 1024 processors.

The rest of the paper is organized as follows. In Section 2 we describe the problem and the efficient sequential algorithm. In Section 3, we present the parallel algorithm along with analysis of partitioning and load balancing. Experimental results showing the performance of our parallel algorithms are presented in Section 4. We conclude in Section 5.

2 Chung–Lu Model and Efficient Sequential Algorithm

Chung–Lu (CL) model [7] generates random networks from a given sequence of expected degrees. We are given n nodes and a set of non-negative weights $w = (w_0, \ldots w_{n-1})$ assuming $\max_i w_i^2 < S$, where $S = \sum_k w_k$ [7]. For every pair of nodes i and j, edge (i, j) is added to the graph with probability $p_{i,j} = \frac{w_i w_j}{S}$. If no self loop is allowed, i.e., $i \neq j$, the expected degree of node i is given by $\sum_j \frac{w_i w_j}{S} = w_i - \frac{w_i^2}{S}$. For massive graphs, where n is very large, the average degree converges to w_i, thus w_i represents the expected degree of node i [16].

The naïve algorithm of CL model for an undirected graph with n nodes takes each of the $\frac{n(n-1)}{2}$ possible node pairs $\{i, j\}$ and creates the edge with probability $p_{i,j}$, therefore requiring $O\left(n^2\right)$ time. An $O(n + m)$ algorithm was proposed in [16] to generate networks assuming w is sorted in non-increasing order, where m is the number of edges. It is easy to see that $O(n+m)$ is the best possible runtime to generate m edges. The algorithm is based on the edge skipping technique introduced in [3] for Erdős–Rényi model. Adaptation of that technique leads to the efficient sequential algorithm in [16]. The pseudocode of the algorithm is given in Algorithm 2.1, consisting of two procedures Serial–CL and Create–Edges. Note that we restructured Algorithm 2.1 by defining procedure Create–Edges to use it without any changes later in our parallel algorithm. Below we provide an overview and a brief description of the algorithm (for complete explanation and correctness see [16]).

The algorithm starts at Serial–CL, which computes the sum S and calls procedure Create–Edges(w, S, V), where V is the entire set of nodes. For each node $i \in V$, the algorithm selects some random nodes v from $[i + 1, n - 1]$, and creates the edges (i, v). A naïve way to select the nodes v from $[i + 1, n - 1]$ is: for each $j \in [i + 1, n - 1]$, select j independently with probability $p_{i,j} = \frac{w_i w_j}{S}$, leading to an algorithm with run time $O(n^2)$. Instead, the algorithm skips the nodes that are not selected by a random skip length δ as follows. For each $i \in V$ (Line 6), the algorithm starts with $j = i + 1$ and computes a random skip length $\delta \leftarrow \left\lfloor \frac{\log(r)}{\log(1-p)} \right\rfloor$, where r is a real number in $(0, 1)$ chosen uniformly at random and $p = p_{i,j} = \frac{w_i w_j}{S}$. Then node v is selected by skipping the next δ nodes (Line 14), and edge (i, v) is selected with probability $\frac{q}{p}$, where $q = p_{i,v} = \frac{w_i w_v}{S}$ (Line 16–19). Then from the next node $j + v$, this cycle of skipping and selecting edges is repeated (while loop in Line 8–20). As we always have $i < j$ and no edge (i, j) can be selected more than once, this algorithm

Algorithm 2.1 Sequential Chung–Lu Algorithm

```
1:  procedure SERIAL–CL(w)
2:      S ← ∑_k w_k
3:      E ← CREATE–EDGES(w, S, V)

4:  procedure CREATE–EDGES(w, S, V)
5:      E ← ∅
6:      for all i ∈ V do
7:          j ← i + 1, p ← min((w_i w_j)/S, 1)
8:          while j < n and p > 0 do
9:              if p ≠ 1 then
10:                 choose a random r ∈ (0, 1)
11:                 δ ← ⌊log(r)/log(1−p)⌋
12:             else
13:                 δ ← 0
14:             v ← j + δ                                  ▷ skip δ edges
15:             if v < n then
16:                 q ← min((w_i w_v)/S, 1)
17:                 choose a random r ∈ (0, 1)
18:                 if r < q/p then
19:                     E ← E ∪ {i, v}
20:                 p ← q,   j ← v + 1
21:     return E
```

does not create any self-loop or parallel edges. As the set of weights w is sorted in non-increasing order, for any node i, the probability $p_{i,j} = \frac{w_i w_j}{S}$ decreases monotonically with the increase of j. It is shown in [16] that for any i, j, edge (i, j) is included in E with probability exactly $\frac{w_i w_j}{S}$, as desired, and that the algorithm runs in $O(n + m)$ time.

3 Parallel Algorithm for the CL Model

Next we present our distributed memory parallel algorithm for the CL model. Although our algorithm generates undirected edges, for the ease of discussion we consider u as the *source node* and v as the *destination node* for any edge (u, v) generated by the procedure CREATE–EDGES. Let T_u be the task of generating the edges from source node u (Lines 6–20 in Algorithm 2.1). It is easy to see that for any pair of nodes (u, v), generating edges in task T_u does not depend on generating edges in task T_v, i.e., tasks T_u and T_v can be executed independently by two different processors. Now execution of procedure CREATE–EDGES(w, S, V) is equivalent to executing the set of tasks $\{T_u : u \in V\}$. Efficient parallelization of Algorithm 2.1 requires:

- Computing the sum $S = \sum_{k=0}^{n-1} w_k$ in parallel
- Dividing the task of executing CREATE–EDGES into independent subtasks
- Accurately estimating the computational cost for each task
- Balancing computational load among the processors

To compute the sum S efficiently, a parallel sum operation is performed on w using P processors, which takes $O\left(\frac{n}{P} + \log P\right)$ time. To divide the task of executing procedure CREATE–EDGES into independent subtasks, 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 $i \neq j$, $V_i \cap V_j = \emptyset$ and $\bigcup_i V_i = V$. Then V_i is assigned to processor P_i, and P_i execute the tasks $\{T_u : u \in V_i\}$; that is, P_i executes CREATE–EDGES(w, S, V_i).

Estimating and balancing computational loads accurately are the most challenging tasks. To achieve good speedup of the parallel algorithm, both tasks must also be done in parallel, which are non-trivial problems. A good load balancing is achieved by properly partitioning the set of nodes V such that the computational loads are equally distributed among the processors. We use two classes of partitioning schemes named consecutive partitioning (CP) and round-robin partitioning (RRP). In CP scheme consecutive nodes are assigned to each partition, whereas in RRP scheme nodes are assigned to the partitions in a round-robin fashion. The use of various partitioning schemes is not only interesting for understanding the performance of the algorithm, but also useful in analyzing the generated networks. It is sometimes desirable to generate networks on the fly and analyze it without performing disk I/O. Different partitioning schemes can be useful for different network analysis algorithms. Many network analysis algorithms require partitioning the graph into an equal number of nodes (or edges) per processor. Some algorithms also require the consecutive nodes to be stored in the same processor. Before discussing the partitioning schemes in detail, we describe some formulations that are applicable to all of these schemes.

Let e_u be the expected number of edges produced and c_u be the computational cost in task T_u for a *source node* u. For the sake of simplicity, we assign one unit of time to process a node or an edge. With $S = \sum_{v=0}^{n-1} w_v$, we have:

$$e_u = \sum_{v=u+1}^{n-1} p_{u,v} = \sum_{v=u+1}^{n-1} \frac{w_u w_v}{S} = \frac{w_u}{S} \sum_{v=u+1}^{n-1} w_v \tag{1}$$

$$c_u = e_u + 1 \tag{2}$$

For two nodes $u, v \in V$ such that $u < v$, we have $c_u \geq c_v$ as shown in Lemma 3.1.

Lemma 3.1. *For any two nodes $u, v \in V$ such that $u < v$, $c_u \geq c_v$.*

Proof. Proof omitted. The lemma follows immediately from Equation 2 and the fact that, the weights are sorted in non-increasing order. □

The expected number of edges generated by the tasks $\{T_u : u \in V_i\}$ is given by $m_i = \sum_{u \in V_i} e_u$. Note that the expected number of edges in the generated graph, i.e., the expected total number of edges generated by all processors is $m = |E| = \sum_{i=0}^{P-1} m_i = \sum_{u=0}^{n-1} e_u$. The computational cost for processor P_i is given by:

$$c(V_i) = \sum_{u \in V_i} c_u = \sum_{u \in V_i} (e_u + 1) = m_i + |V_i| \tag{3}$$

Therefore, the total cost for all processors is given by:

$$\sum_{i=0}^{P-1} c(V_i) = \sum_{i=0}^{P-1} (m_i + |V_i|) = m + n \tag{4}$$

3.1 Consecutive Partitioning (CP)

Let partition V_i starts at node n_i and ends at node $n_{i+1} - 1$, where $n_0 = 0$ and $n_P = n$, i.e., $V_i = \{n_i, n_i+1, \ldots, n_{i+1}-1\}$ for all i. We say n_i is the *lower boundary* of partition V_i. A naïve way for partitioning V is where each partition consists of an equal number of nodes, i.e., $|V_i| = \lceil \frac{n}{P} \rceil$ for all i. To keep the discussion neat, we simply use $\frac{n}{P}$. Although the number of nodes in each partition is equal, the computational cost among the processors is very imbalanced. Lemma 3.2 shows that for two consecutive partitions V_i and V_{i+1}, $c(V_i) > c(V_{i+1})$ for all i and the difference is at least $\frac{n^2}{SP^2} \overline{W}_i \overline{W}_{i+1}$, where $\overline{W}_i = \frac{1}{|V_i|} \sum_{u \in V_i} w_u$, the average weight (degree) of the nodes in V_i.

Lemma 3.2. *Let $c(V_i)$ be the computational cost for partition V_i. In the naïve partitioning scheme, we have $c(V_i) - c(V_{i+1}) \geq \frac{n^2}{SP^2} \overline{W}_i \overline{W}_{i+1}$, where $\overline{W}_i = \frac{1}{|V_i|} \sum_{u \in V_i} w_u$, the average weight of the nodes in V_i.*

Proof. In the naıve partitioning scheme, each of the partitions has $x = \frac{n}{P}$ nodes, except the last partition which can have smaller than x nodes. For the ease of discussion, assume that for $u \geq n$, $w_u = 0$ and consequently $e_u = 0$. Now, $V_i = \{ix, ix + 1, \ldots, (i + 1)x - 1\}$. Using Equation 3, we have:

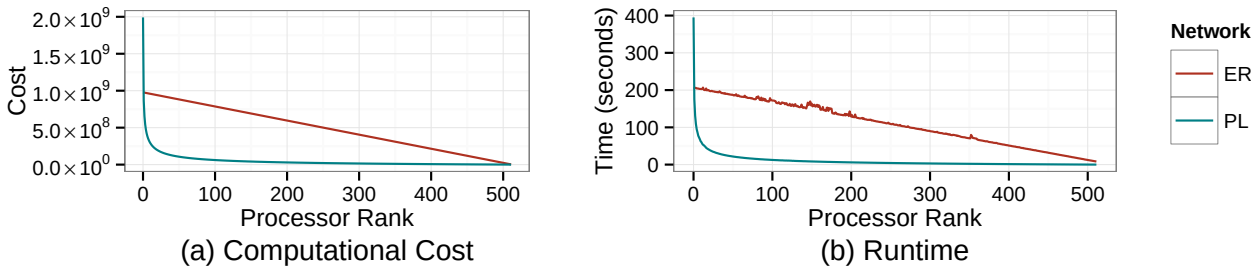

Figure 1: Computational cost and runtime in naïve CP scheme

$$
\begin{aligned}
c(V_i) - c(V_{i+1}) &= \sum_{u \in V_i} (e_u + 1) - \sum_{u \in V_{i+1}} (e_u + 1) \\
&\geq \sum_{u=ix}^{(i+1)x-1} (e_u + 1) - \sum_{u=(i+1)x}^{(i+2)x-1} (e_u + 1) \\
&= \sum_{u=ix}^{(i+1)x-1} (e_u - e_{u+x}) \\
&= \sum_{u=ix}^{(i+1)x-1} \left(\frac{w_u}{S} \sum_{v=u+1}^{n-1} w_v - \frac{w_{u+x}}{S} \sum_{v=u+x+1}^{n-1} w_v \right) \\
&\geq \sum_{u=ix}^{(i+1)x-1} \frac{w_u}{S} \sum_{v=u+1}^{u+x} w_v \geq \sum_{u=ix}^{(i+1)x-1} \frac{w_u}{S} x\overline{W}_{i+1} \\
&= \frac{x\overline{W}_{i+1}}{S} \cdot x\overline{W}_i = \frac{n^2}{SP^2} \overline{W}_i \overline{W}_{i+1}
\end{aligned}
$$

□

Thus $c(V_i)$ gradually decreases with i by a large amount leading to a very imbalanced distribution of the computational cost.

To demonstrate that naïve CP scheme leads to imbalanced distribution of computational cost, we generated two networks, both with one billion nodes: *i*) Erdős–Rényi network with an average degree of 500, and *ii*) Power–Law network with an average degree of 49.72. We used 512 processors, which is good enough for this experiment. Figure 1 shows the computational cost and runtime per processor. In both cases, the cost is not well-balanced. For power-law network the imbalance of computational cost is more prominent. Observe that the runtime is almost directly proportional to the cost, which justifies our choice of cost function. That is balancing the cost would also balance the runtime.

We need to find the partitions V_i such that each partition has equal cost, i.e., $c(V_i) \approx \overline{Z}$, where $\overline{Z} = \frac{(m+n)}{P}$ is the average cost per processor. We refer such partitioning scheme as uniform cost partitioning (UCP). Although determining the partition boundaries in the naïve scheme is very easy, finding the boundaries in UCP scheme is a non trivial problem and requires: (i) computing the cost c_u for each node $u \in V$ and (ii) finding the boundaries of the partitions such that every partition has a cost of $\overline{Z}$. Naïvely computing costs for all nodes takes $O\left(n^2\right)$ time as each node independently requires $O\left(n\right)$ time using Equation 1 and 2. A trivial parallelization achieves $O\left(\frac{n^2}{P}\right)$ time. Our algorithm performs this computation in parallel in $O\left(\frac{n}{P} + \log P\right)$ time.

Finding the partition boundaries such that the maximum cost of a partition is minimized is a well-known problem named *chains-on-chains partitioning* (CCP) problem [18]. In CCP, a sequence of $P-1$ separators are determined to divide a chain of n tasks with associated non-negative weights (c_u) into P partitions so that the maximum cost in the partitions is minimized. Sequential algorithms for CCP are studied quite

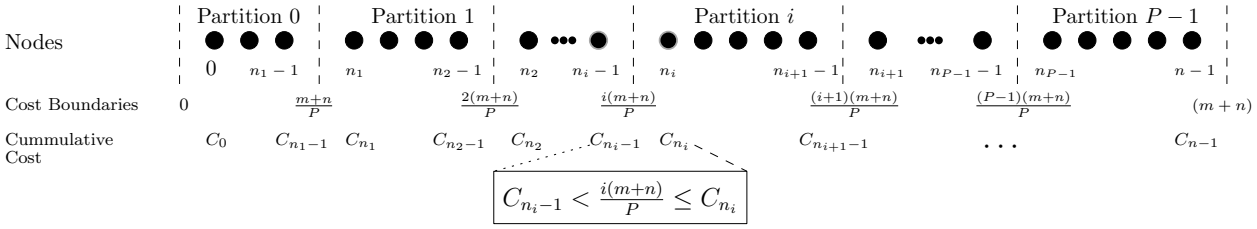

Figure 2: Uniform cost partitioning (UCP) scheme

extensively [15, 17, 18]. Since these algorithms take at least $\Omega(n+P\log n)$ time, using any of these sequential algorithms to find the partitions, along with the parallel algorithm for the CL model, does not scale well. To the best of our knowledge, there is no parallel algorithm for CCP problem. We present a novel parallel algorithm for determining the partition boundaries which takes $O\left(\frac{n}{P}+P\right)$ time in the worst case.

To determine the partition boundaries, instead of using c_u directly, we use the cumulative cost $C_u = \sum_{v=0}^{u} c_v$. We call a partition V_i a *balanced partition* if the computational cost of V_i is $c(V_i) = \sum_{u=n_i}^{n_{i+1}-1} c_u = C_{n_{i+1}-1} - C_{n_i-1} \approx \overline{Z}$. Also note that for lower boundary n_i of partition V_i we have, $C_{n_i-1} < i\overline{Z} \leq C_{n_i}$ for $0 < i \leq P-1$. Thus, we have:

$$n_i = \arg\min_u \left(C_u \geq i\overline{Z}\right) \tag{5}$$

In other words, a node u with cumulative cost C_u belongs to partition V_i such that $i = \left\lfloor \frac{C_u}{\overline{Z}} \right\rfloor$. The partition scheme is shown visually in Figure 2.

Computing C_u in Parallel. Computing C_u has two difficulties: i) for a node u, computing c_u by using Equation 1 and 2 directly is inefficient and ii) C_u is dependent on C_{u-1}. To overcome the first difficulty, we use the following form of e_u to calculate c_u. From Equation 1 we have:

$$\begin{aligned}
e_u &= \frac{w_u}{S}\sum_{v=u+1}^{n-1} w_v \\
&= \frac{w_u}{S}\left(\sum_{v=0}^{n-1} w_v - \sum_{v=0}^{u} w_v\right) \\
&= \frac{w_u}{S}\left(\sum_{v=0}^{n-1} w_v - \sum_{v=0}^{u-1} w_v - w_u\right) \\
c_u &= e_u + 1 = \frac{w_u}{S}\left(S - \sigma_u - w_u\right) + 1 \qquad \left[\text{where } \sigma_u = \sum_{v=0}^{u-1} w_v\right]
\end{aligned} \tag{6}$$

Therefore, c_u can be computed by successively updating $\sigma_u = \sigma_{u-1} + w_{u-1}$.

To deal with the second difficulty, we compute C_u in several steps using procedure Calc–Cost as shown in Algorithm 3.1 (see Figure 3 for a visual representation of the algorithm). In each processor, the partitioning algorithm starts with procedure UCP that calculates the cumulative costs using procedure Calc–Cost. Then procedure Make–Partition is used to compute the partitioning boundaries. At the beginning of the Calc–Cost procedure, the task of computing costs for the n nodes are distributed among the P processors equally, i.e., processor P_i is responsible for computing costs for the nodes from $i\frac{n}{P}$ to $(i+1)\frac{n}{P}-1$. Note that these are the nodes that processor P_i works with while executing the partitioning algorithm to find the boundaries of the partitions.

In Step **1** (Line 6), P_i computes a partial sum $s_i = \sum_{u=\frac{in}{P}}^{\frac{(i+1)n}{P}-1} w_u$ independently of other processors. In Step **2** (Line 7), *exclusive prefix sum* $S_i = \sum_{j=0}^{i-1} s_j$ is calculated for all s_i where $0 \leq i \leq P-1$ and $S_0 = 0$.

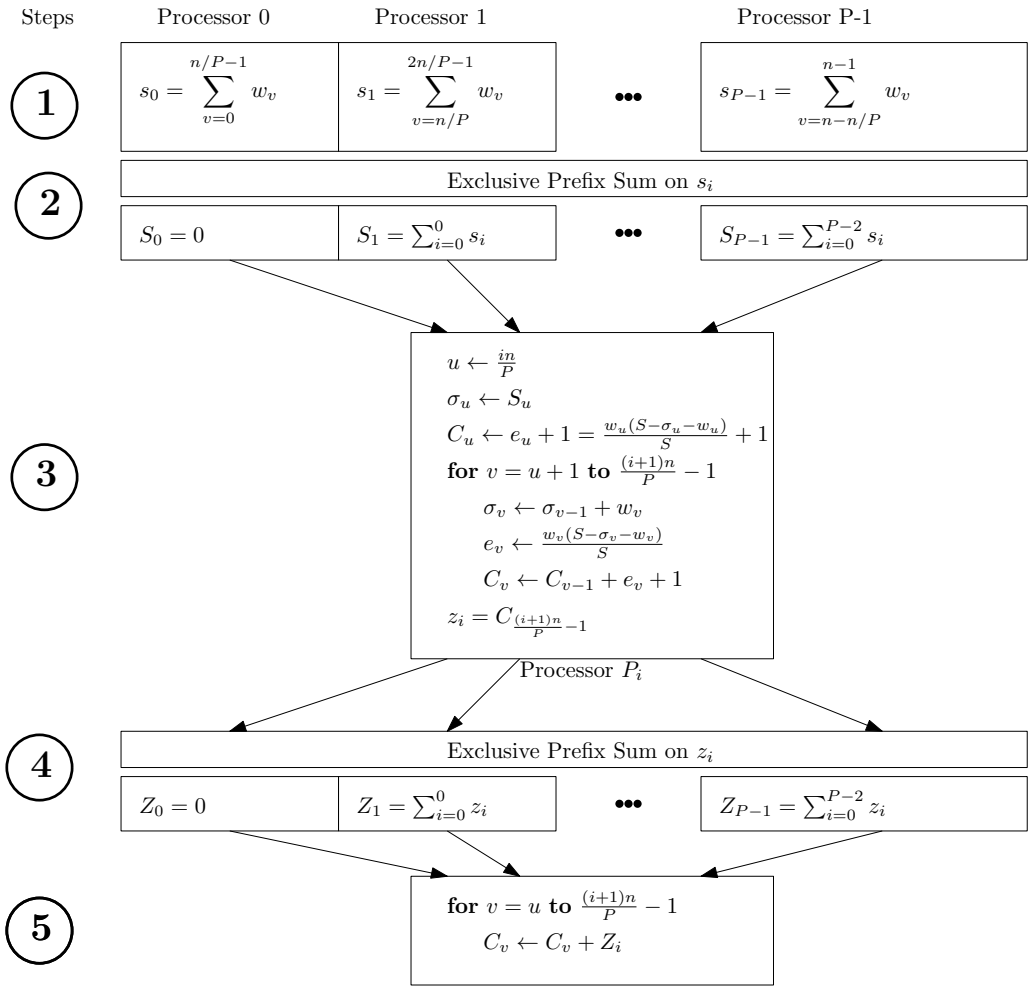

Figure 3: Steps for determining cumulative cost in UCP

This exclusive prefix sum can be computed in parallel in $O(\log P)$ time [21]. We have:

$$S_i = \sum_{j=0}^{i-1} s_j = \sum_{j=0}^{i-1} \sum_{u=\frac{jn}{P}}^{\frac{(j+1)n}{P}-1} w_u = \sum_{u=0}^{\frac{in}{P}-1} w_u = \sigma_{\frac{in}{P}}$$

In Step **3**, P_i partially computes C_u, where $\frac{in}{P} \leq u < \frac{(i+1)n}{P}$. By assigning $\sigma_{\frac{in}{P}} = S_i$, $C_{\frac{in}{P}}$ is determined partially using Equation 6 in constant time (Line 10). For each u, values of σ_u , e_u and C_u are also determined in constant time (Line 11–14), where $\frac{in}{P} + 1 \leq u \leq \frac{(i+1)n}{P} - 1$. After Step **3**, we have $C_u = \sum_{v=\frac{in}{P}}^{u} c_v$. To get the final value of $C_u = \sum_{v=0}^{u} c_v$, the value $\sum_{v=0}^{v=\frac{in}{P}-1} c_v$ needs to be added. For a processor P_i, let $z_i = C_{\frac{(i+1)n}{P}-1} = \sum_{v=\frac{in}{P}}^{\frac{(i+1)n}{P}-1} c_v$. In Step **4** (Line 16), another exclusive parallel prefix sum operation is performed on z_i so that

$$Z_i = \sum_{j=0}^{i-1} z_j = \sum_{j=0}^{i-1} \sum_{v=\frac{jn}{P}}^{\frac{(j+1)n}{P}-1} c_v = \sum_{v=0}^{\frac{in}{P}-1} c_v.$$

Note that Z_i is exactly the value required to get the final cumulative cost C_u. In Step **5** (Lines 17–18), Z_i is added to C_u for $\frac{in}{P} \leq u \leq \frac{(i+1)n}{P} - 1$.

Finding Partition Boundaries in Parallel. The partition boundaries are determined using Equation 5. The procedure MAKE–PARTITION generates the partition boundaries. In Line 20, parallel sum is performed on z_i to determine $Z = \sum_0^{P-1} z_i = \sum_0^{n-1} c_u = n + m$, the total cost and $\overline{Z} = \frac{Z}{P}$, the average cost per processor (Line 21). FIND–BOUNDARIES is called to determine the boundaries (Line 22). From Equation 5, it is easy to show that a partition boundary is found between two consecutive nodes u and $u+1$, such that $\left\lfloor \frac{C_u}{\overline{Z}} \right\rfloor \neq \left\lfloor \frac{C_{u+1}}{\overline{Z}} \right\rfloor$. Node $u+1$ is the lower boundary of partition V_i, where $i = \left\lfloor \frac{C_{u+1}}{\overline{Z}} \right\rfloor$. P_i executes FIND–BOUNDARIES from nodes $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. FIND–BOUNDARIES is a divide & conquer based algorithm to find all the boundaries in that range efficiently using the cumulative costs. All the found boundaries are stored in a local list. In Line 28, it is determined whether the range contains any boundary. If the range does not have any boundary, i.e., if $\left\lfloor \frac{C_s}{\overline{Z}} \right\rfloor = \left\lfloor \frac{C_e}{\overline{Z}} \right\rfloor$, the algorithm returns immediately. Otherwise, it determines the middle of the range m in Line 29. In Line 30, the existence of a boundary between m and $m+1$ is evaluated. If $m+1$ is indeed a lower partition boundary, it is stored in local list in Line 31. In Line 32 and 33, FIND–BOUNDARIES is called with the ranges $[s, m]$ and $[m+1, e]$ respectively. Note that the range $\left[\frac{in}{P}, \frac{(i+1)n}{P} - 1\right]$ may contain none, one or more boundaries. Let B_i be the set of those boundaries. Once the set of boundaries B_i, for all i, are determined, the processors exchange these boundaries with each other as follows. Node n_k, in some B_i, is the boundary between the partitions V_k and V_{k+1}, i.e., $n_k - 1$ is the upper boundary of V_k, and n_k is the lower boundary of V_{k+1}. In Line 23, for each n_k in the range $\left[\frac{in}{P}, \frac{(i+1)n}{P} - 1\right]$, processor P_i sends a boundary message containing n_k to processors P_k and P_{k+1}. Notice that each processor i receives exactly two boundary messages from other processors (Line 25), and these two messages determine the lower and upper boundary of the i-th partition V_i. That is, now each processor i has partition V_i and is ready to execute the parallel algorithm for the CL model with UCP scheme.

The runtime of parallel Algorithm 3.1 is $O\left(\frac{n}{P} + P\right)$ as shown in Theorem 3.3.

Theorem 3.3. *The parallel algorithm for determining the partition boundaries of the UCP scheme runs in $O\left(\frac{n}{P} + P\right)$ time, where n and P are the number of nodes and processors, respectively.*

Proof. The parallel algorithm for determining the partition boundaries is shown in Algorithm 3.1. For each processor, Line 6 takes $O\left(\frac{n}{P}\right)$ time. The exclusive parallel prefix sum operation requires $O(\log P)$ time in Line 7. Lines 8–10 take constant time. The for loop at Line 11 iterates $\frac{n}{P} - 1$ times. Each execution of the

Algorithm 3.1 Uniform Consecutive Partition

1: **procedure** UCP(V, w, S)
2: CALC–COST(w, V, S)
3: MAKE–PARTITION(w, V, S)

4: **procedure** CALC–COST(w, V, S)
5: $i \leftarrow$ processor id
6: $s_i \leftarrow \sum_{u=i\frac{n}{P}}^{(i+1)\frac{n}{P}-1} w_u$
7: **In Parallel:** $S_i \leftarrow \sum_{j=0}^{i-1} s_j$
8: $u \leftarrow \frac{in}{P}$
9: $\sigma_u \leftarrow S_i$
10: $C_u \leftarrow e_u + 1 = \frac{w_u}{S}(S - \sigma_u - w_u) + 1$
11: **for** $u = \frac{in}{P} + 1$ to $\frac{(i+1)n}{P} - 1$ **do**
12: $\sigma_u \leftarrow \sigma_u + w_u$
13: $e_u \leftarrow \frac{w_u}{S}(S - \sigma_u - w_u)$
14: $C_u \leftarrow C_{u-1} + e_u + 1$
15: $z_i \leftarrow C_{\frac{(i+1)n}{P}-1}$
16: **In Parallel:** $Z_i \leftarrow \sum_{j=0}^{i-1} z_j$
17: **for** $u = \frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$ **do**
18: $C_u = C_u + Z_i$

19: **procedure** MAKE–PARTITION(w, V, S)
20: **In Parallel:** $Z \leftarrow \sum_{i=0}^{P-1} z_i$
21: $\overline{Z} \leftarrow \frac{Z}{P}$
22: FIND–BOUNDARIES($\frac{in}{P}$, $\frac{(i+1)n}{P} - 1$, C, $\overline{Z}$)
23: **for all** $n_k \in B_i$ **do**
24: Send n_k to P_k and P_{k+1}
25: Receive boundaries n_i and n_{i+1}
26: **return** $V_i = [n_i, n_{i+1} - 1]$

27: **procedure** FIND–BOUNDARIES(s, e, C, $\overline{Z}$)
28: **if** $\left\lfloor \frac{C_s}{\overline{Z}} \right\rfloor = \left\lfloor \frac{C_e}{\overline{Z}} \right\rfloor$ **then return**
29: $m \leftarrow \frac{(e+s)}{2}$
30: **if** $\left\lfloor \frac{C_m}{\overline{Z}} \right\rfloor \neq \left\lfloor \frac{C_{m+1}}{\overline{Z}} \right\rfloor$ **then**
31: $n_{\left\lfloor \frac{C_{m+1}}{\overline{Z}} \right\rfloor} \leftarrow m + 1$
32: FIND–BOUNDARIES($s, m, C, \overline{Z}$)
33: FIND–BOUNDARIES($m + 1, e, C, \overline{Z}$)

for loop takes constant time for Lines 12–14. Hence, the for loop at Line 11 takes $O\left(\frac{n}{P}\right)$ time. The prefix sum in Line 16 takes $O(\log P)$ time. The for loop at Line 17 takes $O\left(\frac{n}{P}\right)$ time.

The parallel sum operation in Line 20 takes $O(\log P)$ time using `MPI_Reduce` function. For each processor P_i, n_k's are determined in FIND–BOUNDARIES on the range of $\left[\frac{in}{P}, \frac{(i+1)n}{P} - 1\right]$. Finding a single partition boundary on these $\frac{n}{P}$ nodes require $O\left(\log \frac{n}{P}\right)$ time. If the range contains x partition boundaries, then it takes $O\left(\min\left\{\frac{n}{P}, x \log \frac{n}{P}\right\}\right)$ time. For each partition boundary n_k, processor i sends exactly two messages to the processors P_k and P_{k-1}. Thus each processor receives exactly two messages. There are at most P boundaries in $\left[\frac{in}{P}, \frac{(i+1)n}{P} - 1\right]$. Thus, in the worst case, a processor may need to send at most $2P$ messages, which takes $O(P)$ time. Therefore, the total time in the worst case is $O\left(\frac{n}{P} + \min\left\{\frac{n}{P}, P \log \frac{n}{P}\right\} + P\right) = O\left(\frac{n}{P} + P\right)$. □

Theorem 3.3 shows the worst case runtime of $O\left(\frac{n}{P} + P\right)$. Notice that this bound on time is obtained considering the case that all P partition boundaries n_k can be in a single processor. However, in most real-world networks, it is an unlikely event, especially when the number of processors P is large. Thus it is safe to say that for most practical cases, this algorithm will scale to a larger number of processors than the runtime analysis suggests. Now we experimentally show the number of partition boundaries found in the first partition for some popular networks. For the ER networks, the maximum number of boundaries in a processor is 2, regardless of the number of processors. Even for the power–law networks, which has very skewed degree distribution, the maximum number of boundaries in a single processor is very small. Figure 4 shows the maximum number of boundaries found in a single processor. Two fitted plots of $\log^2 P$ and $\log P$ is added in the figure for comparison. From the trend, it appears the maximum number of partition boundaries

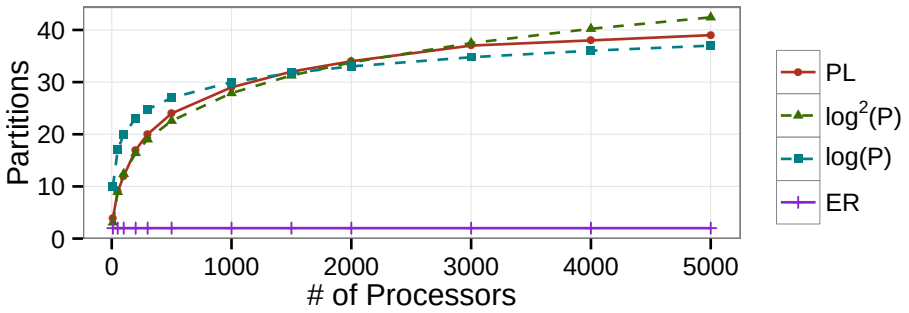

Figure 4: Maximum number of boundaries in a single processor

in a processor is somewhere between $O(\log P)$ and $O(\log^2 P)$. Since power–law has one of the most skewed degree distribution among real-world networks, we can expect the runtime to find partition boundaries to be approximately $O\left(\frac{n}{P} + \log^2 P\right)$ time.

Using the UCP scheme, our parallel algorithm for generating random networks with the CL model runs in $O\left(\frac{m+n}{P} + P\right)$ time as shown in Theorem 3.5. To prove Theorem 3.5, we need a bound on computation cost which is shown in Theorem 3.4.

Theorem 3.4. *The computational cost in each processor is $O\left(\frac{m+n}{P}\right)$ w.h.p.*

Proof. For each $u \in V_i$ and $v > u$, (u, v) is a potential edge in processor P_i, and P_i creates the edge with probability $p_{u,v} = \frac{w_u w_v}{S}$ where $S = \sum_{v \in V} w_v$. Let x be the number of potential edges in P_i, and these potential edges are denoted by $f_1, f_2, \ldots, f_x$ (in any arbitrary order). Let X_k be an indicator random variable such that $X_k = 1$ if P_i creates f_k and $X_k = 0$ otherwise. Then the number of edges created by P_i is $X = \sum_{k=1}^{x} X_k$.

As discussed in Section 2, generating the edges efficiently by applying the edge skipping technique is stochastically equivalent to generating each edge (u, v) independently with probability $p_{u,v} = \frac{w_u w_v}{S}$. Let ξ_e be the event that edge e is generated. Regardless of the occurrence of any event ξ_e with $e \neq (u, v)$, we always have $\Pr\{\xi_{(u,v)}\} = p_{u,v} = \frac{w_u w_v}{S}$. Thus, the events ξ_e for all edges e are mutually independent. Following the definitions and formalism given in Section 3.1, we have the expected number of edges created by P_i, denoted by μ, as

$$\mu = E[X] = \sum_{u \in V_i} e_u = m_i.$$

Now we use the following standard Chernoff bound for independent indicator random variables and for any $0 < \delta < 1$,

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

Using this Chernoff bound with $\delta = \frac{1}{2}$, we have

$$\Pr\left\{X \geq \frac{3}{2} m_i\right\} \leq e^{-\frac{m_i}{12}} \leq \frac{1}{m_i^3}$$

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

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

for all i simultaneously. Then with probability at least $1 - \frac{1}{m_i^2}$, the computation cost $X + |V_i|$ is bounded by $\frac{3}{2} m_i + |V_i| = O(m_i + |V_i|)$. By construction of the partitions by our algorithm, we have $O(m_i + |V_i|) = O\left(\frac{m+n}{P}\right)$. Thus the computation cost in all processors is $O\left(\frac{m+n}{P}\right)$ w.h.p. □

Theorem 3.5. *Our parallel algorithm with UCP scheme for generating random networks with the CL model runs in $O\left(\frac{m+n}{P} + P\right)$ time w.h.p.*

Proof. Computing the sum S in parallel takes $O\left(\frac{n}{P} + \log P\right)$ time. Using the UCP scheme, node partitioning takes $O\left(\frac{n}{P} + P\right)$ time (Theorem 3.3). In the UCP scheme, each partition has $O\left(\frac{m+n}{P}\right)$ computation cost w.h.p. (Theorem 3.4). Thus creating edges using procedure CREATE–EDGES requires $O\left(\frac{m+n}{P}\right)$ time, and the total time is $O\left(\frac{n}{P} + P + \frac{m+n}{P}\right) = O\left(\frac{m+n}{P} + P\right)$ w.h.p. □

3.2 Round-Robin Partitioning (RRP)

In RRP scheme nodes are distributed in a round robin fashion. Partition V_i has the nodes $\langle i, i+P, i+2P, \ldots, i+kP \rangle$ such that $i + kP \leq n < i + (k+1)P$; i.e., $V_i = \{j | j \mod P = i\}$. In other words node i is assigned to $V_{i \mod P}$. The number of nodes in each partition is almost equal, either $\lfloor \frac{n}{P} \rfloor$ or $\lceil \frac{n}{P} \rceil$.

In order to compare the computational cost, consider two partitions V_i and V_j with $i < j$. Now, for the x-th nodes in these two partitions, we have: $c_{i+(x-1)P} \geq c_{j+(x-1)P}$ as $i + (x-1)P < j + (x-1)P$ (see Lemma 3.1). Therefore, $c(V_i) = \sum_{u \in V_i} c_u \geq c(V_j) = \sum_{u \in V_j} c_u$ and by the definition of RRP scheme, $|V_i| \geq |V_j|$. The difference in cost between any two partitions is at most w_0, the maximum weight as shown in Lemma 3.6.

Lemma 3.6. *In Round Robin Partitioning (RRP) scheme, for any $i < j$, we have $c(V_i) - c(V_j) \leq w_i$.*

Proof. The difference in cost between two partitions V_i and V_j is given by:

$$\begin{aligned} c(V_i) - c(V_j) &= \sum_{u \in V_i} c_u - \sum_{u \in V_j} c_u = \sum_{x=0}^{k} (c_{i+xP} - c_{j+xP}) \\ &= c_i - \sum_{x=0}^{k-1} \left(c_{j+xP} - c_{i+(x+1)P}\right) - c_{j+kP} \\ &\leq c_i - c_{j+kP} \qquad \left[c_{j+xp} \geq c_{i+(x+1)P}\right] \\ &\leq e_i = \frac{w_i}{S} \sum_{v=i+1}^{n-1} w_v < \frac{w_i}{S} S = w_i \end{aligned}$$

□

Thus RRP scheme provides quite good load balancing. However, it is not as good as the UCP scheme. It is easy to see that in the RRP scheme, for any two partitions V_i and V_j such that $i < j$, we have $c(V_i) > c(V_j)$. But, by design, the UCP scheme makes the partition such that cost are equally distributed among the processors. Furthermore, although the RRP scheme is simple to implement and provides quite good load balancing, it has another subtle problem. In this scheme, the nodes of a partition are not consecutive and are scattered in the entire range leading to some serious efficiency issues in accessing these nodes. One major issue is that the locality of reference is not maintained leading to a very high rate of cache miss during the execution of the algorithm. This contrast of performance between UCP and RRP is even more prominent when the goal is to generate massive networks as shown by experimental results in Section 4.

4 Experimental Results

In this section, we experimentally show the accuracy and performance of our algorithm. The accuracy of our parallel algorithms is demonstrated by showing that the generated degree distributions closely match the input degree distribution. The strong scaling of our algorithm shows that it scales very well to a large number of processors. We also present experimental results showing the impact of the partitioning schemes on load balancing and performance of the algorithm.

Experimental Setup. We used a 81-node HPC cluster for the experiments. Each node is powered by two octa-core SandyBridge E5-2670 2.60GHz (3.3GHz Turbo) processors with 64 GB memory. The algorithm is developed with MPICH2 (v1.7), optimized for QLogic InfiniBand cards. In the experiments, degree distributions of real-world and artificial random networks were considered. The list of networks is shown in Table 1. The runtime does not include the I/O time to write the graph into the disk.

Table 1: Networks used in the experiments

Network	Type	Nodes	Edges
PL	Power Law Network	1**B**	249**B**
ER	Erdős–Rényi Network	1**M**	200**M**
Miami [2]	Contact Network	2.1**M**	51.4**B**
Twitter [24]	Real–World Social Network	41.65**M**	1.37**B**
Friendster [25]	Real–World Social Network	65.61**M**	1.81**B**

Degree Distribution of Generated Networks. Figure 5 shows the input and generated degree distributions for ER, PL, Miami, Twitter, and Friendster networks. As observed from the plots, the generated degree distributions closely follow the input degree distributions reassuring that our parallel algorithms generate random networks with given expected degree sequences accurately.

Effect of Partitioning Schemes. As discussed in Section 3.1, partitioning significantly affects load balancing and performance of the algorithm. We demonstrate the effects of the partitioning schemes in

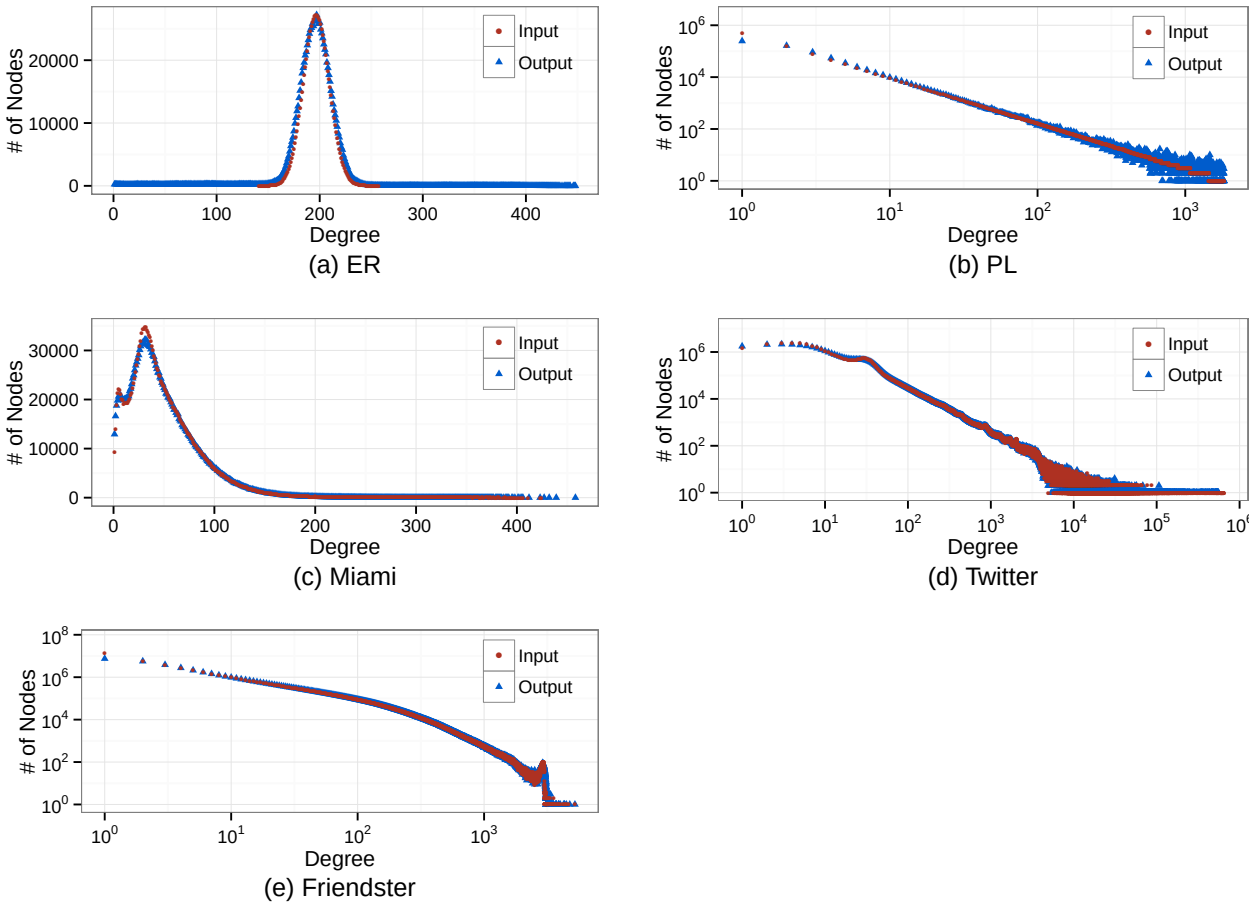

Figure 5: Degree distributions of input and generated degree sequences

terms of computing time in each processor as shown in Figure 6 using ER, Twitter, and PL networks. Computational time fo naïve scheme is skewed. For all the networks, the computational times for UCP and RRP stay almost constant in all processors, indicating good load-balancing. RRP is little slower than UCP because the locality of references is not maintained in RRP, leading to high cache miss as discussed in Section 3.2.

Strong and Weak Scaling. Strong scaling of a parallel algorithm shows its performance with the increasing number of processors while keeping the problem size fixed. Figure 7 shows the speedup of naïve, UCP, and RRP partitioning schemes using PL and Twitter networks. Speedups are measured as $\frac{T_s}{T_p}$, where T_s and T_p are the running time of the sequential and the parallel algorithm, respectively. The number of processors were varied from 1 to 1024. As Figure 7 shows, UCP and RRP achieve excellent linear speedups. Naïve scheme performs the worst as expected. The speedup of PL is greater than that of Twitter network. As Twitter is smaller than the PL network, the impact of the parallel communication overheads is higher contributing to decreased speedup. Still the algorithm to generate Twitter network has a speedup of 400 using 1024 processors.

The weak scaling measures the performance of a parallel algorithm when the input size per processor remains constant. For this experiment, we varied the number of processors from 16 to 1024. For P processors, a PL network with $10^6 P$ nodes and $10^8 P$ edges is generated. Note that weak scaling can only be performed on artificial networks. Figure 7(c) shows the weak scaling for UCP and RRP schemes using PL networks. Both RRP and UCP show very good weak scaling with almost constant runtime.

Generating Large Networks. The primary objective of the parallel algorithm is to generate massive random networks. Using the algorithm with UCP scheme, we have generated power law networks with one billion nodes and 249 billion edges in one minute using 1024 processors with a speedup of about 800.

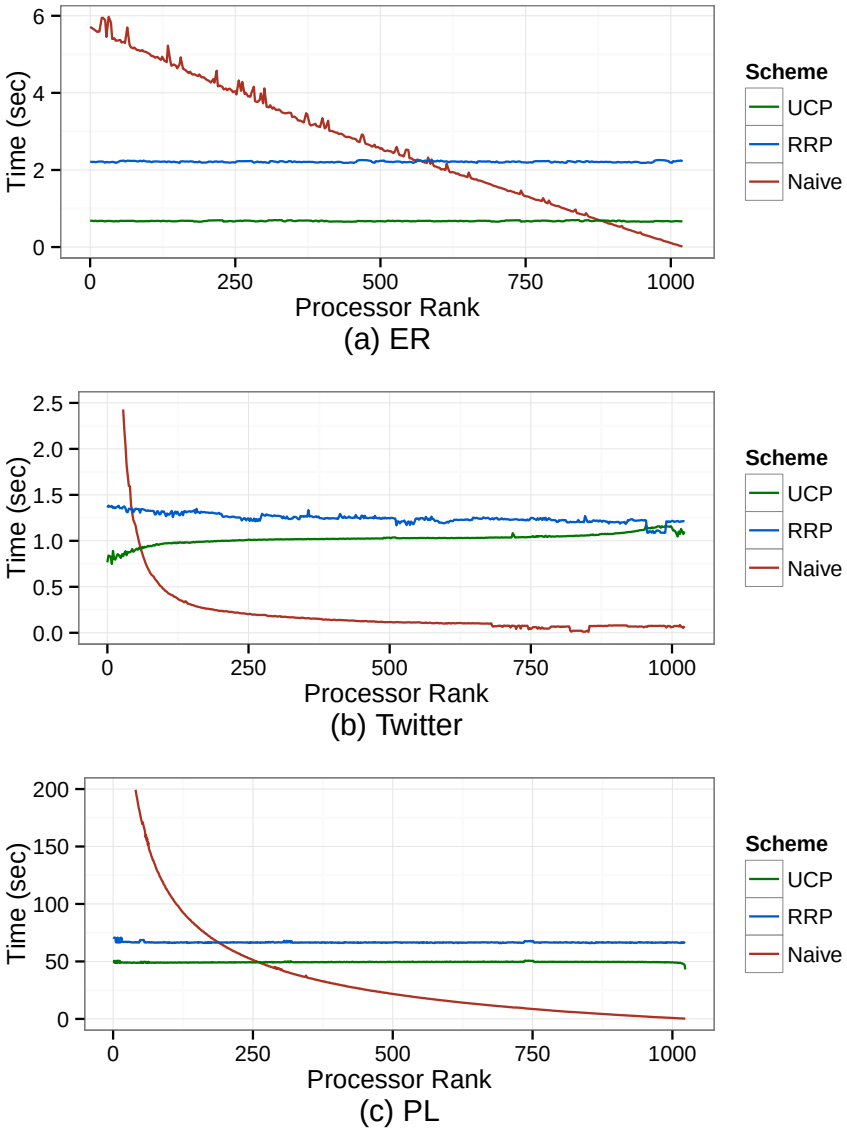

Figure 6: Comparison of partitioning schemes

5 Conclusion

We have developed an efficient parallel algorithm for generating massive networks with a given degree sequence using the Chung–Lu model. The main challenge in developing this algorithm is load balancing. To overcome this challenge, we have developed a novel parallel algorithm for balancing computational loads that results in a significant improvement in efficiency. We believe that the presented parallel algorithm for the Chung–Lu model will prove useful for modeling and analyzing emerging massive complex systems and uncovering patterns that emerges only in massive networks. As the algorithm can generate networks from any given degree sequence, its application will encompass a wide range of complex systems.

Acknowledgements

This work has been partially supported by DTRA Grant HDTRA1-11-1-0016, DTRA CNIMS Contract HDTRA1-11-D-0016-0001, NSF NetSE Grant CNS-1011769, NSF SDCI Grant OCI-1032677, and NSF DIBBs Grant ACI-1443054.

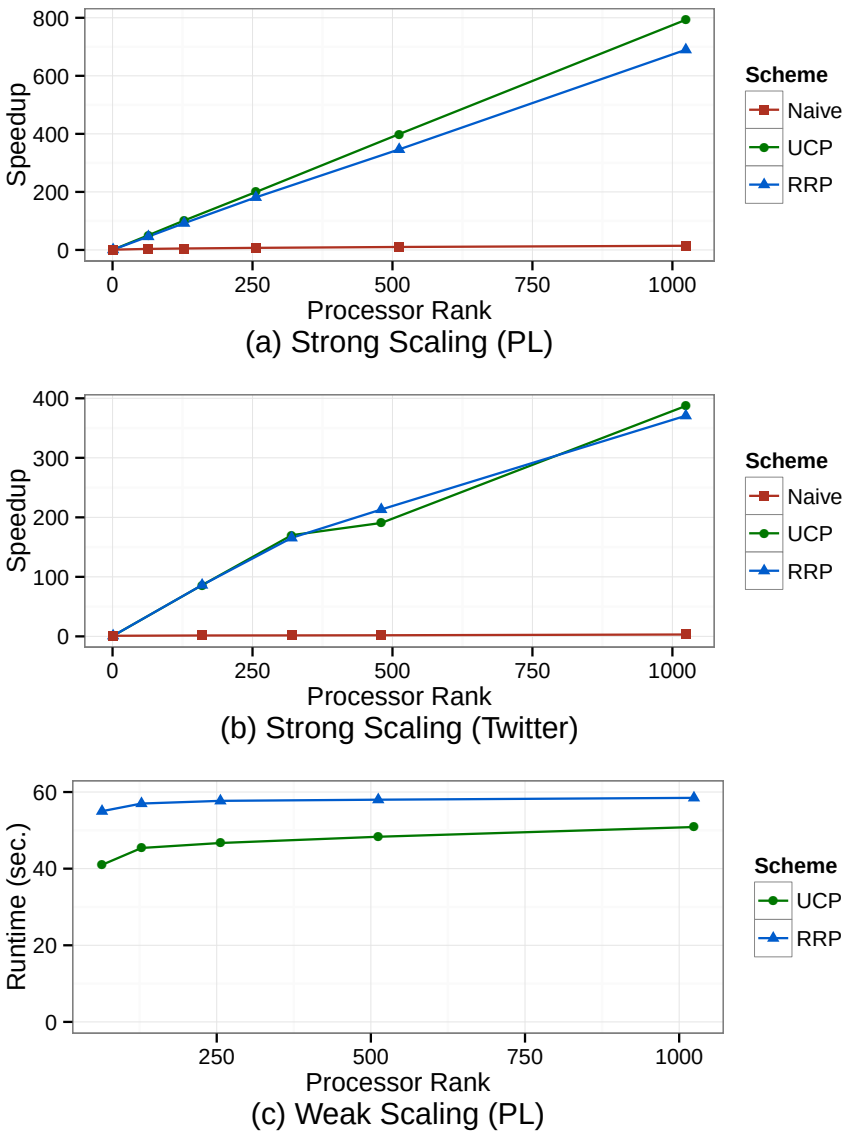

Figure 7: Strong and weak scaling of the parallel algorithms

References

[1] Albert Barabási and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439): 509–512, 1999.

[2] C. Barrett, R. Beckman, M. Khan, V. Kumar, M. Marathe, P. Stretz, T. Dutta, and B. Lewis. Generation and analysis of large synthetic social contact networks. In *Proc. of the Winter Sim. Conf.*, pages 1003–1014, 2009.

[3] Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

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

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Fourth SIAM International Conference on Data Mining*, volume 4, pages 442–446, 2004.

[6] David 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):667–677, 2005.

[7] F. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *Annals of Combinatorics*, 6(2):125–145, 2002.

[8] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. In *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, volume 5, pages 17–61, 1960.

[9] M. Girvan and M. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[10] Graph500. Graph 500. http://www.graph500.org/, 2010.

[11] Vito Latora and Massimo Marchiori. Vulnerability and protection of infrastructure networks. *Physical Review E*, 71(1):015103, 2005.

[12] Jure Leskovec. *Dynamics of large networks*. PhD thesis, Carnegie Mellon University, 2008.

[13] Jure Leskovec and Christos Faloutsos. Scalable modeling of real graphs using kronecker multiplication. In *Proc. of the 24th Intl. Conf. on Machine Learning*, pages 497–504, 2007.

[14] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11: 985–1042, 2010.

[15] Fredrik Manne and Tor Sørevik. Optimal partitioning of sequences. *Journal of Algorithms*, 19(2): 235–249, 1995.

[16] Joel Miller and Aric Hagberg. Efficient generation of networks with given expected degrees. In *Proceedings of Algorithms and Models for the Web-Graph*, volume 6732, pages 115–126, 2011.

[17] B. Olstad and Fredrik Manne. Efficient partitioning of sequences. *IEEE Transactions on Computers*, 44(11):1322–1326, 1995.

[18] Ali Pinar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004.

[19] Ali Pinar, Comandur Seshadhri, and Tamara Kolda. The similarity between stochastic Kronecker and Chung–Lu graph models. In *Proceedings of the Twelfth SIAM International Conference on Data Mining*, volume 12, pages 1071–1082, 2012.

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

[21] Peter Sanders and Jesper Träff. Parallel prefix (scan) algorithms for MPI. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192, pages 49–57, 2006.

[22] 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, 2003.

[23] Duncan Watts and Steven Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684): 409–410, 1998.

[24] Jaewon Yang and Jure Leskovec. Patterns of temporal variation in online media. In *Proc. of the 4th ACM Intel. Conf. on Web Search and Data Mining*, pages 177–186, 2011.

[25] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, number 3, pages 1–8, 2012.