Fast Parallel Algorithms for Edge-Switching to Achieve a Target Visit Rate in Heterogeneous Graphs

Hasanuzzaman Bhuiyan*†, Jiangzhuo Chen†, Maleq Khan†, and Madhav V. Marathe*†
*Department of Computer Science
†Network Dynamics and Simulation Science Laboratory, Virginia Bioinformatics Institute
Virginia Tech, Blacksburg, VA 24061, USA
Email: {mhb, chenj, maleq, mmarathe}@vbi.vt.edu

***Abstract*—An edge switch is an operation on a network (graph) where two edges are selected randomly and one of their end vertices are swapped with each other. Usually, a sequence of these operations are performed to generate network perturbations having the same degree sequence of the original network. Edge switch operations have important applications in graph theory and network analysis, such as in generating random networks with a given degree sequence, modeling and analyzing dynamic networks (e.g., peer-to-peer networks), studying various dynamic phenomena over a network (e.g., disease dynamics over a social contact network). The growth of real-world networks motivates the need to develop efficient parallel algorithms for performing a large sequence of edge switch operations. The dependencies among successive edge switch operations and the requirement of keeping the graph simple (i.e., no self-loops or parallel edges) as the edges are switched lead to significant challenges in designing a parallel algorithm. Addressing these challenges requires complex synchronization and communication among the processors. In this paper, we present a distributed memory parallel algorithm for switching edges in massive networks (networks with billions of edges) and achieve a speedup factor of 85 with 1024 processors.**

One of the steps in our edge switch algorithm requires the computation of multinomial random variables in parallel. The paper presents the first non-trivial parallel algorithm for the problem. The algorithm achieves a speedup of 925 using 1024 processors.

Keywords*—*edge switch, parallel algorithms, massive networks.

I. Introduction

Edge switch, also known as edge swap, edge flip, edge shuffle, edge rewiring etc., is an operation that swaps the end vertices of the edges in a network. We refer such an edge switch operation simply as "*edge-switch*". Many variations of this problem have been studied [6, 9, 10, 13, 15, 18, 20, 21, 23] with many real-world applications. In the most commonly used edge-switch, two randomly selected edges (a, b) and (c, d) are replaced with edges (a, d) and (b, c) respectively, i.e., the end vertices of the selected edges are swapped with each other. This process is repeated as many times as desired. It is easy to see that this edge-switch preserves the degree of each vertex.

Edge-switch problem has many important applications. It can be used in generating random networks with a given degree sequence. There has been a lot of work on random graph generation, because of the popularity of network models in diverse applications. Most of the prior work involves sequential algorithms, and much of it is restricted to regular graphs; we briefly summarize the main approaches here. A popular method for random graph generation is the *configuration model* (also referred to as the "pairing" model) [17, 4, 25], which involves creating stubs for vertices, choosing pairs of stubs at random, and then connecting them by edges. Unfortunately, this leads to parallel edges, unless the degrees are very small. This basic approach has been modified in various ways to avoid parallel edges in the case of regular graphs [25, 22, 16] (see [4] for a good discussion). Blitzstein et al. [4] gives a simple algorithm for generating random graphs with a given degree sequence using sequential importance sampling, based on the Erdős-Gallai characterization.

By using Havel-Hakimi method [12], a network can be generated following a given degree sequence. Since Havel-Hakimi method is deterministic, this method generates the same network each time it is run with the same degree sequence whereas there can be many different networks with the same degree distribution. However, edge-switch can be combined with Havel-Hakimi method to generate a random network with a given degree sequence [10, 6, 9]. Once a network is generated using Havel-Hakimi method, by randomly switching the edges we can generate a random network with the given degree sequence. The mixing time was shown to be bounded by a large polynomial by Cooper et al. [6], which was extended by Feder et al. [9] to variants of the edge-switch process.

Edge-switch is also used in modeling and studying various dynamic networks such as peer-to-peer networks [9]. Other applications of edge-switch include generation of randomly labeled bipartite graphs with a given degree sequence [15], independent realizations of graphs with a prescribed joint degree distribution using a Markov chain Monte Carlo approach [18], and studying the sensitivity of network topology on dynamics over a network such as disease dynamics over a social contact network [8].

Edge-switch can be paired with additional constraints such as imposing a connectivity requirement, allowing or not allowing parallel edges and loops, etc. NetworkX [11] has a sequential implementation of edge-switch that does not allow parallel edges, but allows loops, and provides the option of imposing connectivity constraints on the graph. A connectivity constraint requires a graph to remain connected after edge-switch. Some theoretical studies of edge-switch for restricted graph classes can be found in literature, such as the study of mixing time of the Markov chain introduced by this operation [6, 10]. However, no effort was given to design parallel algorithms of edge-switch. For smaller graphs, sequential implementation of edge-switch suffices. However, such an algorithm may not work for massive networks for the following reasons: (*i*) a massive network with billions of edges simply may not fit in the memory of a single computing machine, and (*ii*) a sequential algorithm may take a prohibitively long time. These issues can be addressed by a distributed memory parallel algorithm where the network is partitioned and each processor contains one partition.

In this paper, we present a distributed memory parallel algorithm for edge-switching in massive graphs with the constraint that the graph remains simple. The dependencies among successive edge-switches and the requirement of keeping the graph simple lead to significant challenges in designing a parallel algorithm. To deal with these challenges, it requires complex synchronization and communications among the processors, which in turn makes it very challenging to gain any speedup by parallelization. Our parallel algorithm achieves a speedup of 85 with 1024 processors. It can perform 115 billion edge-switches in a very large power-law network with 10 billion edges in less than 3 hours using 1024 processors. This algorithm requires generating multinomial random variables in parallel, which is also a non-trivial problem. To the best of our knowledge, there is no existing parallel algorithm for generating multinomial random variables. We present a parallel algorithm for generating multinomial random variables, which achieves a speedup of 925 using 1024 processors.

The rest of the paper is organized as follows. Section II describes the preliminaries and notations used in the paper. The edge-switch problem and the sequential algorithm are briefly explained in Section III. We present the parallel algorithm of edge-switch in Section IV and the algorithm for generating multinomial random variables in Section V. Finally, we conclude in Section VI.

II. Preliminaries and Notations

We are given a simple graph $G = (V, E)$, where V is the set of vertices, and E is the set of edges. A *simple graph* is an undirected graph with no self-loops and parallel edges. A *self-loop* is an edge from a vertex to itself. *Parallel edges* are two or more edges connecting the same pair of vertices. There are total $n = |V|$ vertices labeled as $0, 1, 2, \ldots, n-1$, and $m = |E|$ edges in the graph G. If $(u, v) \in E$, we say u and v are *neighbors* of each other. The neighbors of a vertex $u \in V$ are stored in the *adjacency list* of u, denoted as $N(u)$, i.e., $N(u) = \{v \in V | (u, v) \in E\}$. The degree of u is $d_u = |N(u)|$. The terms *node* and *vertex*, *graph* and *network*, *neighbor list* and *adjacency list*, *loop* and *self-loop* are used interchangeably throughout the paper. We use H, K, M and B to denote hundreds, thousands, millions and billions, respectively; e.g., $1M$ stands for one million.

Edge-switch: An edge-switch replaces two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$, picked uniformly at random from E, by new edges $e_3 = (u_1, v_2)$ and $e_4 = (u_2, v_1)$, as shown in Fig. 1. If $u_1 = v_2$ or $u_2 = v_1$, then the above edge-switch creates self-loops. The edge-switch creates parallel edges, if edge (u_1, v_2) or (u_2, v_1) have existed in the graph.

Visit Rate: Due to edge-switches, some edges of the given graph G are changed (visited), and some edges that do not participate in any edge-switch remain unchanged (not visited). We define *visit rate* as the fraction of edges of G that have been changed by a sequence of edge-switches. If m' is the number of edges of G that have been changed due to edge-switches, then visit rate $x = \frac{m'}{m}$.

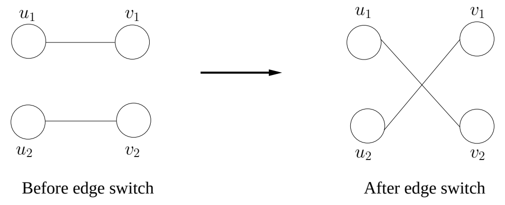

Fig. 1: Edge-switch.

Binomial Distribution: Suppose that N independent trials are to be performed, where each trial results in a success with probability q, and in a failure with probability $(1 - q)$. If X represents the number of successes that occur among N trials, then X is said to be a binomial random variable. The distribution of X is a binomial distribution with parameters N and q, and denoted by equation (1). The probability of getting exactly i successes in N trials is given in equation (2).

$$X \sim \mathcal{B}(N, q) \tag{1}$$

$$\Pr\{X = i\} = \binom{N}{i} q^i (1 - q)^{N-i} \tag{2}$$

Multinomial Distribution: Let N be the number of independent trials to be performed, where each trial has ℓ possible outcomes $0, 1, \ldots, \ell - 1$ with probability $q_0, q_1, \ldots, q_{\ell-1}$ respectively, such that $q_i \geq 0$ for $0 \leq i \leq \ell - 1$ and $\sum_i q_i = 1$. Let the random variable X_i indicates the number of times the outcome i appears among N independent trials. Then $X = \langle X_0, X_1, \ldots, X_{\ell-1} \rangle$ has a multinomial distribution with parameters $N, q_0, q_1, \ldots, q_{\ell-1}$, and denoted as follows.

$$\langle X_0, X_1, \ldots, X_{\ell-1} \rangle \sim \mathcal{M}(N, q_0, q_1, \ldots, q_{\ell-1}) \tag{3}$$

Computation Model: We develop algorithms for distributed memory parallel systems. Each processor has its own local memory. The processors do not have any shared memory and can communicate with each other and exchange data by message passing.

III. Sequential Edge-switch

Problem Statement: Given a simple undirected graph $G = (V, E)$, and a visit rate $x \in (0, 1]$, perform t random edge-switches to achieve the given visit rate x.

A random edge-switch comprises of choosing a pair of edges e and e' uniformly at random from the set of edges in the graph. As a result t is a random variable. In this paper, we *do not enforce* the condition that the graph is connected after each edge-switch. Enforcing this would require testing connectivity after each such switch and is substantially more computationally expensive.

A. Determining the Number of Edges to Switch for a Given Visit Rate

During an edge-switch, a selected edge can be categorized as one of the following two types. (*i*) *Original edge:* an edge that has not participated in any of the previous edge-switches and is still unchanged. (*ii*) *Modified edge:* any edge participating in an edge-switch is replaced by a new edge, and such a new edge is called a modified edge.

Calculating the expected value of t for a given x is similar to the coupon collector problem [1]. Our goal is to have $m' = mx$ modified edges in the graph by switching a sequence of t pairs of edges. The rest $(m - m')$ of the edges remain unchanged. Let at some point there are already $(i-1)$ modified edges in the graph. From this point to have the i-th modified edge we need t_i number of edges to be switched. The probability of selecting the i-th original edge from the graph, given there are $(i-1)$ modified edges, is $p_i = \frac{m-(i-1)}{m}$. Let T be the total number of edges that are switched to have mx modified edges. Here, T and t_i are random variables, and t_i has geometric distribution with expectation $\frac{1}{p_i}$. Using the linearity of expectation,

$$\begin{aligned} E(T) &= \sum_{i=1}^{mx} E(t_i) = \sum_{i=1}^{mx} \frac{1}{p_i} = \sum_{i=1}^{mx} \frac{m}{m-(i-1)} \\ &= m\left(\sum_{i=1}^{m} \frac{1}{i} - \sum_{i=1}^{m(1-x)} \frac{1}{i}\right) \\ &= m\left(H_m - H_{m(1-x)}\right) \end{aligned} \tag{4}$$

where H_m is the m-th harmonic number. Since every edge-switch involves two edges, $t = -\frac{1}{2}m\ln(1-x)$ for $x < 1$, and $t = \frac{1}{2}m\ln m$ for $x = 1$.

Note that we can mark the modified edges and always select two original edges for the next edge-switch. In such a case for a visit rate x to have mx modified edges, we simply need to perform $mx/2$ edge-switches. For a specific application, one can do so. If we do not allow a modified edge to participate in any later edge-switch, the process may not produce many networks with the same degree sequence. Unrestricted and independent random choice of the edges help us obtain a random graph from the space of the graphs with the same degree sequence.

Furthermore, visit rate can also be defined in many other ways and converted to t. At the end, our parallel algorithm can be used to perform t edge-switches, irrespective of how t is obtained.

B. Keeping the Graph Simple

As the edge-switch problem deals with simple graph, we need to make sure that none of the edge-switches create self-loops or parallel edges. Edge-switch between edges (u_1, v_1) and (u_2, v_2) creates

- **Parallel edge:** if $u_1 \in N(v_2)$, $v_2 \in N(u_1)$, $u_2 \in N(v_1)$ or $v_1 \in N(u_2)$.
- **Self-loop:** if $u_1 = v_2$ or $u_2 = v_1$.

An edge-switch does not make any change to the graph if the pair of edges remain the same after switching the edges, and we say such an edge-switch is *useless*. Edge-switch between (u_1, v_1) and (u_2, v_2) is useless if $u_1 = u_2$ or $v_1 = v_2$. For an edge-switch, two edges are selected and switched if the switch is not useless and does not create parallel edges or loops.

C. Switching Edges Sequentially

A sequence of t pairs of edges are switched such that the resultant graph remains simple. The graph, specifically the edge set, dynamically changes with edge-switches. Let $G' = (V, E')$ be such a graph where E' is the current set of edges at a given time. Algorithm 1 shows the pseudocode of switching edges sequentially. The adjacency list of a vertex can be stored using a balanced binary tree. Searching such an adjacency list of a vertex u to determine possibility of parallel edge creation takes $O(\log d_u)$ time. If (u_1, v_1) and (u_2, v_2) are the edges participating in the i-th edge-switch, then the time to perform t edge-switches is $O\left(\sum_{i=1}^{t} \sum_{j \in \{u_1, v_1, u_2, v_2\}} \log d_j\right) \leq O(t \log d_{max})$, where d_{max} is the maximum degree of a vertex in the graph.

Algorithm 1 SEQUENTIAL EDGE-SWITCH (G, x)

```
1: if $x = 1$ then $t \leftarrow \frac{1}{2}m\ln m$
2: else $t \leftarrow -\frac{1}{2}m\ln(1-x)$
3: for $i = 1$ to $t$ do
4:     $(u_1, v_1), (u_2, v_2) \leftarrow$ two uniform random edges $\in E'$
5:     if $u_1 = u_2$, $v_1 = v_2$, $u_1 = v_2$, $u_2 = v_1$, $u_1 \in N(v_2)$, or
       $u_2 \in N(v_1)$ then
6:         go to line 4
7:     Replace $(u_1, v_1)$ and $(u_2, v_2)$ by $(u_1, v_2)$ and $(u_2, v_1)$
       respectively
```

IV. PARALLEL EDGE-SWITCH

In this section, we propose an efficient distributed memory parallel algorithm of edge-switch. Let p be the number of processors, and they are denoted by $P_0, P_1, \ldots, P_{p-1}$. We are given a simple graph $G = (V, E)$ and a visit rate x. The graph is partitioned and distributed among the processors. We need to consider two cases for an edge-switch: (*a*) both edges may belong to the same processor, called *local switch*; (*b*) the edges may belong to different processors; referred as *global switch*. In the later case, the processors need to communicate with each other in order to complete the edge-switch. We explain the details below in the following order: (*i*) data structures, (*ii*) partitioning, (*iii*) switching a pair of edges by a single processor, (*iv*) simultaneous edge-switches by all processors, (*v*) properties of parallel edge-switch and (*vi*) experimental results.

A. Data Structures

A graph can be stored as adjacency lists or an adjacency matrix. In an adjacency matrix, the existence of any edge can be determined in constant time, however it takes $O(n^2)$ memory. Our algorithm uses adjacency lists which takes $O(m+n)$ memory. Usually, $N(u)$ contains all neighbors of u. Instead, we keep the neighbors with labels higher than u, i.e., $N(u) = \{v \in V | (u, v) \in E, u < v\}$; this is referred as *reduced adjacency list*. Below we explain the advantages of using reduced adjacency list.

(*i*) If an adjacency list contains all neighbors of a vertex, every edge (u, v) can be picked up in two ways, one from $N(u)$ and another from $N(v)$. $N(u)$ and $N(v)$ may belong to two different processors. The same edge (u, v) can be picked from two different processors and participate in two different edge-switches at the same time leading to an inconsistency. *Reduced adjacency list* ensures that any edge (u, v) can be picked up only from one processor. Although it is possible to avoid such inconsistency by keeping all neighbors in the list, it will incur more communication cost.

(ii) Every edge-switch involves four vertices' adjacency lists update, one update for each end vertex of an edge. *Reduced adjacency list* minimizes the number of updates to only two or three vertices' adjacency lists leading to efficiency. The details are discussed later in Section IV-C.

However, a difficulty arises from using reduced adjacency list. If $N(u)$ contains all the neighbors of u, any edge (u_1, v_1) can be picked up as (u_1, v_1) from $N(u_1)$ (considering as ordered pair), and as (v_1, u_1) from $N(v_1)$. The probability of being picked each way is $\frac{1}{2m}$. If (u_1, v_1) and (u_2, v_2) (considering no ordering) are two edges selected for edge-switch, then the new edges are either (u_1, v_2) and (u_2, v_1), or (u_1, u_2) and (v_1, v_2) depending on edges being selected from which vertices' adjacency lists. However, in a reduced adjacency list, (u_1, v_1) is selected from $N(u_1)$ and (u_2, v_2) is selected from $N(u_2)$, assuming $u_1 < v_1$ and $u_2 < v_2$, the new edges become (u_1, v_2) and (u_2, v_1). We then miss the chance of generating the edges (u_1, u_2) and (v_1, v_2). This is adjusted by replacing the selected edges by (u_1, u_2) and (v_1, v_2) with probability $\frac{1}{2}$, called *straight switch*, and by (u_1, v_2) and (u_2, v_1) with probability $\frac{1}{2}$, called *cross switch*, as shown in Fig. 2.

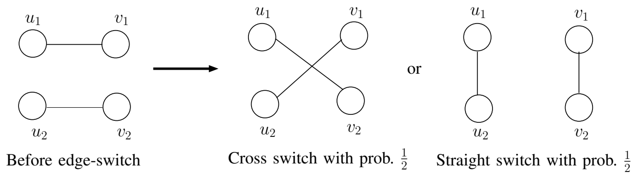
u_1
v_1
u_2
v_2
or
Before edge-switch
Cross switch with prob. $\frac{1}{2}$
Straight switch with prob. $\frac{1}{2}$

Fig. 2: Straight and cross edge-switch.

B. Partitioning

For a given simple graph $G = (V, E)$, we partition V into p disjoint subsets, $V_0, V_1, \ldots, V_{p-1}$, such that $\bigcup_i V_i = V$. The graph is partitioned such that the vertices are sorted in ascending order of their labels among the processors, one vertex's reduced adjacency list belongs to a unique processor, and each processor contains roughly the same number of edges $(\frac{m}{p})$. Each processor P_i contains a set of edges, $E_i = \{(u, v) \in E | u \in V_i, u < v\}$, $\bigcup_i E_i = E$ and $E_i \bigcap E_j = \emptyset$ for $i \neq j$. As the graph is dynamically changing with edge-switches, let us denote E'_i to be the current set of edges in P_i at a given time.

C. Switching a Pair of Edges by a Single Processor

A simple approach of performing one edge-switch is that processor P_i picks one pair of edges uniformly at random from the entire graph (i.e., selecting two processors from $[0, p-1]$ and request them for edges) and switch them through message passing among the processors. However, this approach incurs significant synchronization and communication costs. Instead, P_i selects one edge (u_1, v_1) uniformly at random from its current edge set E'_i, and another edge (u_2, v_2) from the entire graph, i.e., P_i selects a processor P_j with probability $\frac{|E'_j|}{|E|}$ and requests P_j to select an edge (u_2, v_2) from E'_j uniformly at random. If $P_i = P_j$, then it is a *local switch*, otherwise it is a *global switch*. The high level overview of an edge-switch is given in Algorithm 2. The selected edges are switched only if the graph remains simple and the edge-switch is not useless. During the course of an edge-switch process, if any processor P_x detects violation of any constraints, P_x notifies all other processors that are involved in that edge-switch. Then, the initiating processor (P_i for the above example) repeats the edge-switch process to switch a new pair of edges.

Algorithm 2 SWITCHING A PAIR OF EDGES

```
1:  Processor $P_i$ executes the following for an edge-switch:
2:  $e_1 \leftarrow$ a uniform random edge in $E'_i$
3:  $P_j \leftarrow$ a random processor in $[0, p-1]$, where probability
    of choosing $P_x$ is $\frac{|E'_x|}{|E|}$
4:  if $P_i = P_j$ then
5:      Choose an edge $e_2$ from $E'_i$ to switch with edge $e_1$
6:      Switch the edges $e_1$ and $e_2$ ($P_i$ may communicate with
        a different processor $P_k$ requesting to add a new edge)
7:  else
8:      Send message $< e_1$, request to pick an edge from $E'_j >$
        to $P_j$
9:      Upon receipt of the above message, $P_j$ executes the
        following:
10:     Choose an edge $e_2$ from $E'_j$ to switch with edge $e_1$
11:     $P_i$ and $P_j$ work together to switch $e_1$ and $e_2$ ($P_j$ may
        communicate with a different processor $P_k$ requesting
        to add a new edge)
```

Local Switch: P_i selects two edges (u_1, v_1) and (u_2, v_2) from E'_i uniformly at random such that the edge-switch does not create loops and it is not useless. P_i decides whether it will be a *straight* or a *cross switch* with equal probability. If it is a *cross switch*, P_i checks whether (u_1, v_2) and (u_2, v_1) create parallel edges. If not, P_i removes (u_1, v_1) and (u_2, v_2), adds (u_1, v_2) and (u_2, v_1), thus completing the edge-switch process. If the edge-switch is a *straight switch*, P_i determines P_k such that $\min(v_1, v_2) \in V_k$. If $P_i = P_k$, P_i checks whether (u_1, u_2) and (v_1, v_2) create parallel edges. If not, P_i removes (u_1, v_1) and (u_2, v_2), adds (u_1, u_2) and (v_1, v_2) and completes the edge-switch process. If $P_i \neq P_k$, P_i checks whether (u_1, u_2) creates parallel edges. If not, P_i sends a message to P_k requesting to add (v_1, v_2). If (v_1, v_2) does not create parallel edges, P_k adds (v_1, v_2) and sends a message to P_i indicating the updates at P_k. Upon receiving this message, P_i removes (u_1, v_1), (u_2, v_2) and adds (u_1, u_2).

Global Switch: In a *global switch*, two edges are selected from two different processors, say P_i and P_j, $i < j$. Assuming P_i initiates the edge-switch process, P_i selects an edge $e_1 = (u_1, v_1) \in E'_i$ uniformly at random. P_i sends a message, containing the edge e_1 and a request to select an edge from E'_j, to P_j. Upon receiving this message from P_i, processor P_j selects $e_2 = (u_2, v_2) \in E'_j$ uniformly at random, and decides whether the edge-switch will be a *straight* or a *cross* switch with equal probability. At this point, P_j knows the new edges that will be added to the graph after removing e_1 and e_2; we refer these new edges as *potential edges* until the updates take place. Let us describe the *cross switch* in detail.

Processor P_j checks whether $u_2 = v_1$ and $v_1 = v_2$ to detect a loop and a useless edge-switch respectively. If it does not create a loop and is not useless, P_j determines P_k such that $\min(u_2, v_1) \in V_k$. We need to consider the following three cases.

(i) If $P_k = P_j$, P_j checks whether (u_2, v_1) creates parallel edges. If not, then P_j sends v_2 to P_i. P_i checks whether (u_1, v_2) creates parallel edges. If the graph remains simple, P_i removes edge (u_1, v_1), adds edge (u_1, v_2), and sends a message to P_j indicating the updates at P_i. Upon receiving this message, P_j removes (u_2, v_2) and adds (u_2, v_1), thus completing the edge-switch process.

(ii) If $P_k = P_i$, P_j sends a message, containing e_2 and a request to add both the new edges to P_i. Processor P_i checks whether (u_1, v_2) and (u_2, v_1) create parallel edges. If not, P_i removes (u_1, v_1), adds edges (u_1, v_2) and (u_2, v_1), and sends a message to P_j indicating the updates at P_i. Then P_j completes the edge-switch by removing (u_2, v_2).

(iii) If $P_i \neq P_k \neq P_j$, P_j sends (u_2, v_1) and v_2 to P_k. P_k checks whether (u_2, v_1) creates parallel edges. If not, P_k sends v_2 to P_i. P_i checks whether (u_1, v_2) creates parallel edges. If not, P_i removes (u_1, v_1), adds (u_1, v_2), and sends messages to P_j and P_k indicating the updates taken place at P_i. Then P_j removes edge (u_2, v_2), and P_k adds edge (u_2, v_1), thus completing the edge-switch process.

Similar approach is followed for a straight switch and for $i > j$. The data structure we use eliminates the following two constraints: *(i)* $u_1 = u_2$, and *(ii)* $u_1 = v_2$ if $i < j$, or $u_2 = v_1$ if $i > j$.

D. Simultaneous Edge-switches by All Processors

In the sequential algorithm, pairs of edges are selected sequentially, one pair after another. As the edges are selected randomly, the number of edges selected from each partition E_i may not be equal. To have an equivalent parallel algorithm, we need to select the same number of edges from each partition E_i as the sequential algorithm would do. In the parallel algorithm, for each edge-switch, processor P_i selects the first edge from its own partition E_i and the second edge from the entire graph. Let X_i be the number of first edges selected from partition E_i by a sequential algorithm. A sequential algorithm does not need to know X_i in advance. However, for the parallel algorithm, X_i for each i need to be determined in advance so that processor P_i knows how many edge-switches it needs to perform where the first edge is selected from its own partition E_i.

For any edge-switch, the probability that the first edge is selected from E_i is $q_i = \frac{|E_i|}{|E|}$ for $i = 0, 1, \ldots, p-1$, and we have $\sum_{i=0}^{p-1} q_i = 1$. Then it is easy to see that the random variables X_i for $i = 0, 1, \ldots, p-1$ are multinomially distributed with parameters $(t, q_0, q_1, \ldots, q_{p-1})$; i.e.,

$$\langle X_0, X_1, \ldots, X_{p-1}\rangle \sim \mathcal{M}(t, q_0, q_1, \ldots, q_{p-1}) \qquad (5)$$

The time complexity of the best known conditional distributed method [7] for generating multinomial variables is $\theta(t)$. Thus to have an efficient parallel algorithm for edge-switch, we also need an efficient parallel algorithm for generating multinomial random variables. To the best of our knowledge, there is no existing parallel algorithm for this problem. In Section V, we present an efficient parallel algorithm for computing multinomial random variables that runs in $O(\frac{t}{p} + p\log p)$ time.

Each processor P_i simultaneously performs X_i number of edge-switches. After completing one edge-switch, P_i proceeds to its next edge-switch, while serving other processors' requests in the mean-time. Below we discuss some *issues* that arise from performing edge-switches simultaneously.

(a) Even after maintaining all the constraints to keep a graph simple, parallel edges can be created in a different way. As multiple pairs of edges are switched by multiple processors simultaneously, a same new edge can be created by multiple processors at the same time. For example, more than one instance of an edge (u, v) is created simultaneously if more than one of the following four edge-switches are performed simultaneously by different processors, where '$-$' denotes an end vertex of an edge. *(i)* Cross edge-switch between $(u, -)$ and $(-, v)$. *(ii)* Cross edge-switch between $(-, u)$ and $(v, -)$. *(iii)* Straight edge-switch between $(u, -)$ and $(v, -)$. *(iv)* Straight edge-switch between $(-, u)$ and $(-, v)$. Keeping track of *potential edges* at each processor ensures no parallel edges will be created in the above mentioned way.

(b) The number of edges changes (i.e., increases or decreases) among the processors with edge-switches. Hence, the initial assumption of picking edges from different processors with constant probabilities (i.e., $q_i = \frac{|E_i|}{|E|}$) does not hold with progress in edge-switches. On the other hand, updating the probability values after every edge-switch incurs a huge amount of communication costs, leading the algorithm to slow down significantly. To deal with this difficulty, our algorithm performs edge-switches in a number of steps and the probability values are updated at the end of each step. We denote *step-size* to be the number of edge-switches performed by all the processors in a step. With a reasonable step-size, we achieve a close approximation. The experimental results are shown later in Section IV-F5.

Summarizing the Parallel Algorithm of Edge-switch: First t is calculated for a given x. Let s be the *step-size*, and q be the probability vector $\langle q_0, q_1, \ldots, q_{p-1}\rangle$. Edge-switches are performed in $\lceil \frac{t}{s} \rceil$ steps. All the processors perform s number of edge-switches in one step. If $t\%s \neq 0$, $(t - s\lfloor \frac{t}{s} \rfloor)$ number of edge-switches are performed in the last step. The summary of the parallel algorithm is given below.

1) Multinomial Distribution: First s is multinomially distributed among p processors with q to determine S_i, the number of edge-switches that each processor P_i will perform in the current step. This takes $O(\frac{s}{p} + p\log p)$ time.

2) Performing Edge-switches: P_i picks one edge e_1 from E'_i, and the other edge e_2 from the entire graph, and performs edge-switch as described before in Section IV-C. Each processor P_i performs such S_i number of edge-switches simultaneously. For an edge-switch, a constant amount of message passing is required; edges are updated in constant time and checking for parallel edges takes $O(\log d_{max})$ time. Thus, performing S_i edge-switches at P_i takes $O(S_i \log d_{max})$ time.

3) Updating Probability Vector and Termination: After completing S_i edge-switches in the current step, P_i sends *end-of-step* signals (messages) to each processor that requires $O(\log p)$ time. P_i continues to serve requests from other processors until receiving end-of-step signals from every processor, i.e., the end of current step. At the end of each step, P_i receives $|E'_j|$ from each P_j through message passing and it takes $O(\log p)$ time. P_i updates q with the received $|E'_j|$s in $O(p)$ time. Then, s number of edge-switches are again multinomially distributed among p processors using the updated q for the next step and edge-switches are performed. This process continues until t pairs of edges are switched.

E. Properties of Parallel Edge-switch

In this section, we examine some stochastic properties of the parallel edge-switch process and study how stochastically similar it is to the sequential edge-switch process.

Recall that in the sequential edge-switch process, one pair of edges is selected uniformly at random, and the edges are switched before selecting the next pair of edges. After completing the i-th edge-switch, one or both of the two new edges generated by the i-th switch can be selected for the $(i+1)$-th edge-switch. In the parallel edge-switch process, multiple pairs of edges are selected and switched simultaneously by different processors, and thus, the edges generated simultaneously by multiple processors cannot be selected for a simultaneous edge-switch (restricting its choice). It raises the question of whether these two processes are stochastically equivalent or how close are they stochastically? We try to answer this question by studying the similarity of their effect, i.e., the resultant graphs generated by these two edge-switch processes beginning with the same initial graph.

The *stochastic equivalence* of the sequential and parallel edge-switch processes can be defined as follows. Let G_s^t and G_p^t be the resultant graphs after performing t number of edge-switches by the sequential and parallel edge-switch processes, respectively, where both processes begin with the same initial graph G. We say the two processes are stochastically equivalent if $\Pr\{G_s^t = G'\} = \Pr\{G_p^t = G'\}$ for all graphs G' with the same degree sequence as G.

Theoretical analysis of the above stochastic equivalence seems to be difficult. Experimental analysis can also be prohibitively time consuming. As the space of the graphs with a given degree sequence can be very large, estimating probabilities of generating G' by a reasonable number repetitions of the edge-switch processes can be very error prone.

Instead, we measure "similarity" of the two stochastic processes. We say the sequential and parallel processes are *similar* if they satisfy the following two conditions.

1) The distribution of the number of edges switched among different partitions (i.e., subsets of edges) is the same in both G_s^t and G_p^t, the resultant graphs of the sequential and parallel processes, respectively. This goal is achieved by the use of multinomial distribution as described before in Section IV-D.
2) At the end of the edge-switch processes, the distribution of the number of edges across different sets of vertices is the same for both sequential and parallel processes. Let $n_s(V_i, V_j)$ and $n_p(V_i, V_j)$ be the number of cross edges between the sets of vertices V_i and V_j in the resultant graphs G_s^t and G_p^t, respectively. For any positive integer t, after switching t pairs of edges, the distribution of $n_s(V_i, V_j)$ and $n_p(V_i, V_j)$, for all i, j, are same.

The resultant graphs, G_s^t and G_p^t, are divided into r partitions (i.e., $0 \le i, j \le r-1$), with each partition having an equal number of vertices. The *edge difference* across different sets of vertices between G_s^t and G_p^t is computed using equation (6). We define *error rate* between G_s^t and G_p^t as shown in equation (7), where the maximum value of the *edge difference* can be $2m$. Due to randomness, some error rate can be observed even between two resultant graphs, G_{s1}^t and G_{s2}^t, generated by the sequential process in two different runs. If the error rate between G_s^t and G_p^t is roughly equal to the error rate between G_{s1}^t and G_{s2}^t, then the sequential and parallel processes are said to be *similar*. The experimental results are explained in Section IV-F5.

$$\textit{Edge difference} = \sum_{i,j \ge i} |n_s(V_i, V_j) - n_p(V_i, V_j)| \quad (6)$$

$$\textit{Error rate} = \frac{\textit{Edge difference}}{2m} \times 100\% \quad (7)$$

F. Experimental Results

In this section, we present strong and weak scaling of our parallel algorithm, demonstrate the *similarity* of the sequential and parallel algorithms, and analyze the trade off between step-size, error rate and speedup.

1) Experimental Setup: We use a high performance computing cluster of 64 Intel Sandy Bridge compute nodes (Dell C6220). Each computing node consists of dual-socket Intel Sandy Bridge E5-2670 2.60GHz 8-core processor (16 cores per node) and 64GB of 1600MHz DDR3 RAM. The computing nodes are interconnected by Qlogic QDR Infiniband interconnects. To implement our algorithm, we use MPICH2 implementation (version 1.9) of MPI.

TABLE I: Datasets used in the experiments

Network	Type of network	Vertices	Edges	Avg. Degree
New York	Social Contact	20.38M	587.3M	57.63
Los Angeles	Social Contact	16.33M	479.4M	58.66
Miami	Social Contact	2.1M	52.7M	50.4
Flickr	Online Community	2.3M	22.8M	19.83
LiveJournal	Social	4.8M	42.8M	17.83
Small World	Random	4.8M	48M	20
Erdős-Rényi	Erdős-Rényi Random	4.8M	48M	20
PA-100M	Pref. Attachment	100M	1B	20
PA-1B	Pref. Attachment	1B	10B	20

2) Datasets: We use both real-world and artificial networks for the experiments. A summary of the networks is provided in Table I. New York, Los Angeles, and Miami are realistic, yet synthetic social contact networks [3]. Each vertex represents a person in that city, and each edge represents any 'physical' contact between two persons within a 24 hour time period. Flickr is an image based online community network [19]. LiveJournal is a social network blogging site [19]. Small world graph is generated using Watts-Strogatz small world graph model [24], Erdős-Rényi is generated using Erdős-Rényi graph

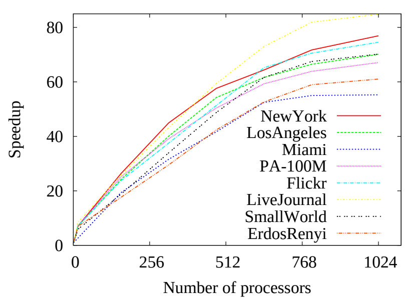

Fig. 3: Strong scaling on different graphs.

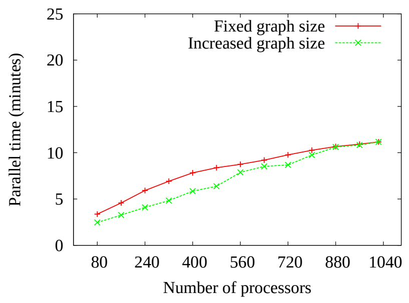

Fig. 4: Weak scaling on Preferential Attachment graphs.

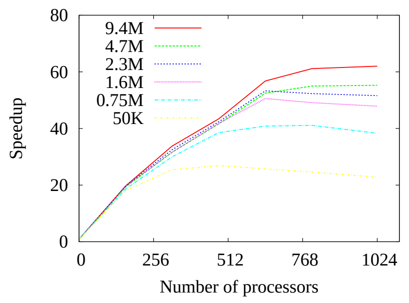

Fig. 5: Strong scaling performance comparison using different step-sizes on Miami graph.

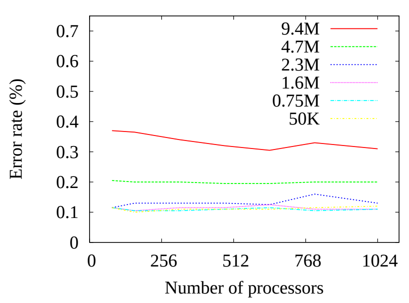

Fig. 6: Error rate with increasing number of processors using different step-sizes on Miami graph.

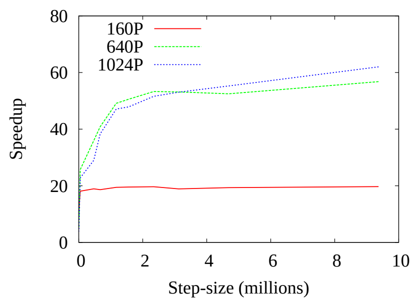

Fig. 7: Speedup with increasing step-size using 160, 640 and 1024 processors on Miami graph.

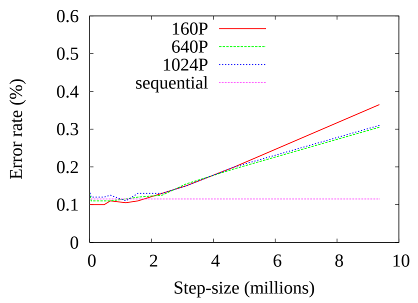

Fig. 8: Error rate with increasing step-size using 160, 640 and 1024 processors on Miami graph.

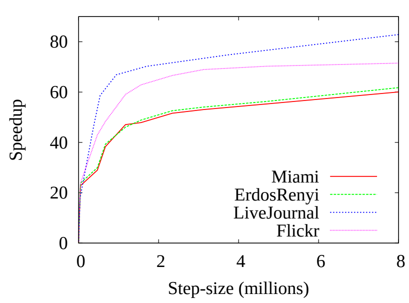

Fig. 9: Speedup with increasing step-size for different graphs using 1024 processors.

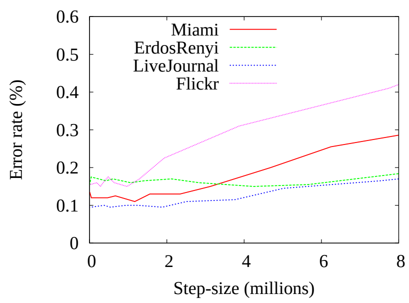

Fig. 10: Error rate with increasing step-size for different graphs using 1024 processors.

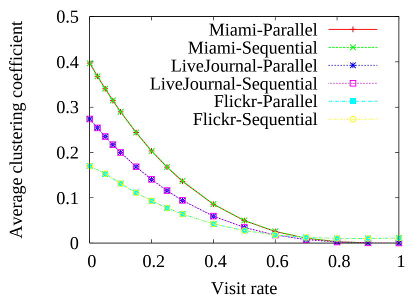

Fig. 11: Average clustering coefficient changes similarly with edge-switches by the sequential and parallel algorithms.

model [5], and PA is generated using Preferential Attachment graph model [2].

3) Strong Scaling: Fig. 3 showcases the strong scaling of the parallel algorithm of edge switch. We use visit rate $x = 1$ and step-size = $\frac{t}{100}$. We have experimented with eight different graphs, and achieved a maximum speedup of 85 using 1024 processors for the LiveJournal graph.

4) Weak Scaling: The weak scaling of our parallel algorithm is shown in Fig. 4. In one experiment, we increase the graph size with the increase of processors, and use Preferential Attachment graphs with $(p \times 0.1M)$ vertices and an average degree of 20. In another experiment, we use a fixed Preferential Attachment graph with $102.4M$ vertices and $1.024B$ edges. In both the experiments we use $t = p \times 10M$ and step size = $\frac{t}{1000}$. Ideally, the parallel runtime should remain constant. However, in practice, the communication increases with the increase of processors, leading to a higher runtime. Our algorithm shows good weak scaling as the runtime increases linearly in both the cases.

5) Similarity of the Parallel and Sequential Algorithm and Determining Suitable Step-size: We use visit rate $x = 1$, and $r = 20$ partitions to measure error rate for all the experiments, unless otherwise specified. Fig. 5 shows better strong scaling is achieved for larger step-size on the Miami graph. For a particular step-size, error rate remains roughly constant with the increase of processors on the Miami graph, as shown in Fig. 6. The effects of step-size on speedup and error rate for the Miami graph are shown in Fig. 7 and Fig. 8 respectively. Both the speedup and error rate increase with the increase of step-size.

By keeping the error rate to a minimum, we want to achieve as much speedup as possible. From Fig. 8, we observe that with up to a $2M$ step-size, the error rate between the resultant graphs generated by the sequential and parallel algorithms is roughly same as the error rate between the resultant graphs generated by two different runs of the sequential algorithm. Hence, $2M$ can be a *suitable* step-size for the Miami graph, since the error rate is minimal, and a good speedup factor of 50 using 1024 processors is achieved at the same time. If we further increase the step-size, both the speedup and error rate increase. For example, using a step-size of $9.4M$, the error rate is a negligible 0.4%, however a higher speedup factor of 62 is achieved using 1024 processors. Fig. 9 and 10 shows the effect of step-size on speedup and error rate for different graphs. Suitable step-size may vary from graph to graph, depending on the graph size and type of the graph. For example, the error rate is roughly constant for different step-sizes on Erdős-Rényi and LiveJournal graphs, though it varies for Flickr and Miami graphs as shown in Fig. 10. A *suitable* step-size for Flickr, Miami, LiveJournal and Erdős-Rényi graphs can be $1.5M$, $2M$, $4M$ and $8M$ respectively. In general, if we use a lower step-size, say $2M$, for any medium-sized graph (having more than $20M$ edges), we expect to have a very small error rate along with a good speedup. The above experiments show that the sequential and the parallel edge-switch processes are similar with a suitable step-size.

We also analyze how some network properties change with edge-switches by sequential and parallel algorithms. We use Miami, LiveJournal, and Flickr graphs with a step-size of $2M$, and vary the visit rate from 0.1 to 1. Fig. 11 and 12 shows that the average clustering coefficient and average shortest path distance of a graph change exactly the same way with edge-switches by the sequential and parallel algorithms. Small variation in average shortest path distance is observed due to using approximate computation, since the exact computation is very time consuming.

Our algorithm is able to perform more than $115B$ edge switches on a Preferential Attachment graph with $1B$ vertices and $10B$ edges in less than 3 hours using 1024 processors. The sequential algorithm is not even able to load such a large graph.

V. Binomial and Multinomial Distribution

In this section we present a parallel algorithm for computing multinomial distribution of very large numbers. First we briefly review the current state-of-the-art sequential algorithm.

A. Sequential Algorithm for Computing Multinomial Distribution

One simple approach for computing multinomial random variables is to perform N independent trials, where the outcome of each trial can be $0, 1, \ldots, \ell - 1$ with probability $q_0, q_1, \ldots, q_{\ell-1}$, respectively. This algorithm takes at least $\Omega(N \log \ell)$ time. An efficient state-of-the-art algorithm is *conditional distributed method* [7], which runs in $O(N)$ time. This method generates multinomial random variables $\langle X_0, X_1, \ldots, X_{\ell-1} \rangle$ by iteratively generating ℓ binomial random variables

$$X_i \sim \mathcal{B}\left(N - \sum_{j=0}^{i-1} X_j, \frac{q_i}{1 - \sum\limits_{j=0}^{i-1} q_j} \right) \tag{8}$$

Inverse transformation method (BINV) [14] is the best known algorithm for computing binomial random variables. To generate a binomial random variable X with parameters N and q, it takes $O(X)$ time. Note that the expected value of X is Nq.

The algorithms for the inverse transformation method (BINV) [14] to compute binomial random variables and conditional distributed method [7] to compute multinomial random variables are shown in Algorithm 3 and 4, respectively. For additional details, see [14] and [7].

Algorithm 3 BINOMIAL(N, q)

```
1: if $q = 1$ then return $N$
2: $i \leftarrow 0$    $\{i$ is the binomial random variable$\}$
3: Generate $u \sim U(0,1)$ uniformly at random
4: $Q \leftarrow (1-q)^N$, $S \leftarrow Q$
5: while $S < u$ do
6:     $i \leftarrow i + 1$
7:     $Q \leftarrow Q \left(\frac{N-i+1}{i}\right)\left(\frac{q}{1-q}\right)$
8:     $S \leftarrow S + Q$
9: return $i$
```

Algorithm 4 MULTINOMIAL($N, q_0, q_1, \ldots, q_{\ell-1}$)

```
1: $X_s \leftarrow 0, Q_s \leftarrow 0$
2: for $i = 0$ to $\ell - 1$ do
3:     if $Q_s < 1$ then
4:         $X_i \leftarrow$ BINOMIAL$\left(N - X_s, \frac{q_i}{1-Q_s}\right)$
5:         $X_s \leftarrow X_s + X_i$
6:         $Q_s \leftarrow Q_s + q_i$
7:     else $X_i \leftarrow 0$
8: return $\langle X_0, X_1, \ldots, X_{\ell-1} \rangle$
```

The conditional distributed method shown in Algorithm 4 runs in $\sum_{i=0}^{\ell-1} O(X_i) = O(N)$ time. In the next section, we present an efficient parallelization of Algorithm 4.

B. Parallel Algorithm for Computing Multinomial Distribution

Based on the conditional distributed method shown in Algorithm 4, we propose a parallel algorithm for computing multinomial distribution $X \sim \mathcal{M}(N, q)$, where q denotes probability vector $\langle q_0, q_1, \dots, q_{\ell-1} \rangle$. One tempting approach to parallelize the conditional distributed method is to distribute the generation of X_i, $0 \le i < \ell$ (Line 4 of Algorithm 4) among the processors. However, a difficulty arises from the sequential nature of computing X_is due to the dependencies of X_i on X_{i-1} for all $i > 0$. We overcome this difficulty by exploiting some properties of binomial and multinomial random variables, as described below.

Let N_i, for $0 \le i < k$, be some integers such that $N = \sum_{i=0}^{k-1} N_i$. If $X_i \sim \mathcal{B}(N_i, q)$, then

$$X = \sum_{i=0}^{k-1} X_i \sim \mathcal{B}\left(\sum_{i=0}^{k-1} N_i, q\right) = \mathcal{B}(N, q) \tag{9}$$

The above property of the binomial random variables leads to the following property of the multinomial random variables. If

$$\langle X_{0,i}, X_{1,i}, \dots, X_{\ell-1,i} \rangle \sim \mathcal{M}(N_i, q_0, q_1, \dots, q_{\ell-1})$$

for $0 \le i < k$, then

$$\langle X_0, X_1, \dots, X_{\ell-1} \rangle \sim \mathcal{M}(N, q_0, q_1, \dots, q_{\ell-1}) \tag{10}$$

where $X_j = \sum_{i=0}^{k-1} X_{j,i}$ for $0 \le j < \ell$ and $N = \sum_{i=0}^{k-1} N_i$.

Now we describe the parallel algorithm for computing multinomial distribution, which uses the above property. First, we explain the case of $p = \ell$. Our algorithm divides the number of trials N into p almost equal small number of trials N_i, and assign N_i to P_i. Then each processor P_i computes the multinomial distribution of N_i using the same probability vector q. At the end, the results of all the processors are aggregated. The pseudocode is given in Algorithm 5, where processor P_i holds the multinomial random variable X_i at the end of computation.

Algorithm 5 PARALLEL MULTINOMIAL$(N, q_0, \dots, q_{\ell-1})$

```
1: Each processor $P_i$ executes the following in parallel:
2: **if** $i < N\%p$ **then** $N_i \leftarrow \lfloor \frac{N}{p} \rfloor + 1$
3: **else** $N_i \leftarrow \lfloor \frac{N}{p} \rfloor$
4: $\langle X_{0,i}, X_{1,i}, \dots, X_{\ell-1,i} \rangle \sim \mathcal{M}(N_i, q_0, q_1, \dots, q_{\ell-1})$
5: Send $X_{j,i}$ to processor $P_j$
6: Upon receiving $X_{i,k}$ from every processor $P_k$:
7:     $X_i \leftarrow \sum_{k=0}^{p-1} X_{i,k}$
```

For $p \ne \ell$, the algorithm is same up to the multinomial distribution computation of N_i at P_i, i.e., lines 1-4 of Algorithm 5. The only difference is how the generated multinomial random variables will be stored among the processors. The variables can be stored in many ways, e.g., all the X_is can be gathered to the root processor P_0, or they (X_is) can be distributed among the processors in a round robin fashion, i.e., assigning X_i to processor $P_{(i\%p)}$, etc. X_i is always computed by summing up all the $X_{i,k}$s ($0 \le k < p$), after receiving them from all the processors.

The parallel computation is almost perfectly load balanced among the processors since each processor computes multinomial distribution of $\frac{N}{p}$ independently, taking $O(\frac{N}{p})$ time. The communication cost at the end takes $O(\ell \log p)$ time. Hence, the time complexity of this algorithm is $O\left(\frac{N}{p} + \ell \log p\right)$. The algorithm is almost perfectly parallelized because the number of processors, p (which is in the range of hundreds or at most thousands), and the number of outcomes ℓ, are significantly smaller than the number of trials N (which is in the range of billions), in general case. Algorithm 5 computes binomial distribution for $\ell = 2$.

During binomial random variable generation, the computation of $(1-q)^N$ (Line 4 of Algorithm 3) results in underflow occurrence for large values of N, e.g., billions. Using *long double* data type cannot solve this underflow occurrence for large N. In addition, some round off errors may appear. We deal with these difficulties by using the property of the binomial distribution again, i.e., we divide N into small N_is such that $\sum_i N_i = N$, compute X using equation (9). The upper threshold value of N_i is set such that no underflow occurs, that is,

$$(1-q)^{N_i} \ge z \tag{11}$$

$$N_i \le \frac{-\log z}{\log(1-q)} \le \frac{-\log z}{2q} \tag{12}$$

where z is the smallest positive real number that can be represented by the data type (e.g., float, double) used and $q < 1$.

C. Performance Analysis of the Parallel Algorithm

In this section, the speedup of the parallel algorithm of multinomial distribution is demonstrated by strong scaling and weak scaling.

1) Strong Scaling: The *strong scaling* of the parallel algorithm is illustrated in Fig. 13. We keep the problem size fixed ($N = 10000B$, $\ell = 20$ and $q_i = \frac{1}{\ell}$), and achieve a speedup of 925 using 1024 processors. The speedup increases almost linearly with the increase of processors. The parallel algorithm can compute multinomial distribution of $10000B$ in 71 seconds using 1024 processors.

2) Weak Scaling: Fig. 14 shows the *weak scaling* of our parallel algorithm. We use $\ell = p$ (i.e., total number of processors), $N = p \times 20B$ (i.e., $20B$ per processor), and equal probability values, $q_i = \frac{1}{\ell}$. The parallel run time is almost constant indicating a very good weak scaling.

VI. CONCLUSION

We have devised a new parallel algorithm of edge-switch, that can generate massive scale random graphs by achieving a target visit rate. It can be used in studying various properties of large dynamic networks. We have experimentally shown that the parallel algorithm is similar to the sequential algorithm. In addition, we have developed a parallel algorithm for computing multinomial random variables that is almost perfectly parallelized. This algorithm can be of independent interest and prove useful in parallelizing many stochastic processes. We believe that the two parallel algorithms will contribute significantly in dealing with big data, one of the most challenging problems in today's research world.

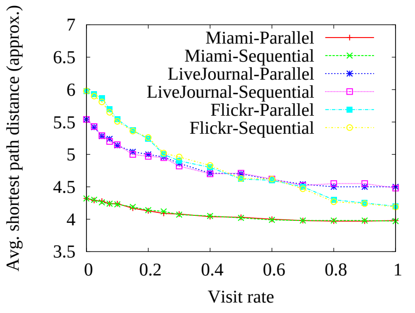

Fig. 12: Average shortest path distance (approximate) changes similarly with edge-switches by the sequential and parallel algorithms.

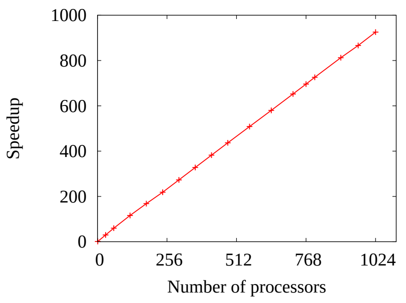

Fig. 13: Strong scaling of the parallel algorithm of multinomial distribution.

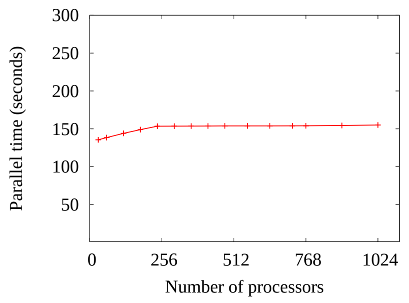

Fig. 14: Weak scaling of the parallel algorithm of multinomial distribution.

References

[1] I. Adler, S. Oren, and S. M. Ross, "The coupon-collector's problem revisited," *Journal of Applied Probability*, vol. 40, no. 2, pp. pp. 513–518, 2003. [Online]. Available: http://www.jstor.org/stable/3215807

[2] A. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, pp. 509–512, 1999.

[3] C. Barrett, R. Beckman *et al.*, "Generation and analysis of large synthetic social contact networks," in *WSC*, 2009, pp. 103–114.

[4] J. Blitzstein and P. Diaconis, "A sequential importance sampling algorithm for generating random graphs with prescribed degrees," Harvard University, Tech. Rep., 2006.

[5] B. Bollobas, *Random Graphs*, W. Fulton, A. Katok, F. Kirwan, P. Sarnak, B. Simon, and B. Totaro, Eds. Cambridge Univ. Press, 2001.

[6] C. Cooper, M. Dyer, and C. Greenhill, "Sampling regular graphs and a peer-to-peer network," *Combinatorics, Probability and Computing*, vol. 16, no. 4, pp. 557–593, 2007.

[7] C. Davis, "The computer generation of multinomial random variates," *Computational Statistics & Data Analytics*, vol. 16, no. 2, pp. 205–217, 1993.

[8] S. Eubank, A. Vullikanti, M. Khan, M. Marathe, and C. Barrett, "Beyond degree distributions: Local to global structure of social contact graphs," in *Proc. of the Third International Conference on Social Computing, Behavioral Modeling, and Prediction (SBP)*, 2010.

[9] T. Feder, A. Guetz, M. Mihail, and A. Saberi, "A local switch markov chain on given degree graphs with application in connectivity of peer-to-peer networks," in *FOCS*, 2006, pp. 69–76.

[10] C. Gkantsidis, M. Mihail, and E. Zegura, "The markov chain simulation method for generating connected power law random graphs," in *5th Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM*, 2003.

[11] A. Hagberg, D. Schult, and P. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Pasadena, CA, USA, Aug. 2008, pp. 11–15.

[12] S. L. Hakimi, "On realizability of a set of integers as degrees of the vertices of a linear graph," *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 3, pp. 496–506, 1962.

[13] M. Jerrum and A. Sinclair, "Fast uniform generation of regular graphs," *Theoretical Computer Science*, vol. 73, no. 1, pp. 91–100, 1990.

[14] V. Kachitvichyanukul and B. Schmeiser, "Binomial random variate generation," *Communications of the ACM*, vol. 31, no. 2, pp. 216–222, 1988.

[15] R. Kannan, P. Tetali, and S. Vempala, "Simple markov-chain algorithms for generating bipartite graphs and tournaments," *Random Struct. Algorithms*, vol. 14, no. 4, pp. 293–308, 1999.

[16] J. Kim and V. Vu, "Sandwiching random graphs: universality between random graph models," 2004.

[17] M. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, 2003.

[18] J. Ray, A. Pinar, and C. Seshadhri, "Are we there yet? when to stop a markov chain while generating random graphs?" in *9th Workshop on Algorithms and Models for the Web Graph*, 2012, pp. 155–164.

[19] SNAP, "Stanford network analysis project," 2012. [Online]. Available: http://snap.stanford.edu/

[20] I. Stanton and A. Pinar, "Constructing and sampling graphs with a prescribed joint degree distribution," *CoRR*, vol. abs/1103.4875, 2011.

[21] A. Stauffer and V. Barbosa, "A study of the edge-switching markov-chain method for the generation of random graphs," Tech. Rep. cs.DM/0512.105, Dec 2005.

[22] A. Steger and N. Wormald, "Generating random regular graphs quickly," *Combin. Probab. Comput.*, pp. 377–396, 1999.

[23] L. Tabourier, C. Roth, and J. Cointet, "Generating constrained random graphs using multiple edge switches," *Journal of Experimental Algorithmics*, vol. 16, pp. 205–217, 2011.

[24] D. Watts and S. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, Jun. 1998.

[25] N. Wormald, "Models of random regular graphs," 1999.