J. Parallel Distrib. Comput. 104 (2017) 19–35

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Parallel algorithms for switching edges in heterogeneous graphs☆

Hasanuzzaman Bhuiyan[a,c,*], Maleq Khan[b,**], Jiangzhuo Chen[c], Madhav Marathe[a,c]

[a] Department of Computer Science, Virginia Tech, 2202 Kraft Drive, Blacksburg, VA 24061, USA
[b] Department of Electrical Engineering and Computer Science, Texas A&M University—Kingsville, Kingsville, TX 78363, USA
[c] Network Dynamics and Simulation Science Laboratory, Biocomplexity Institute of Virginia Tech, 1015 Life Science Circle, Blacksburg, VA 24061, USA

HIGHLIGHTS

- We present distributed memory parallel algorithms for switching edges in a network.
- The algorithms are carefully designed to work efficiently with massive networks.
- Our algorithms deal with complex synchronization and computation dependency issues.
- We study several partitioning schemes in conjunction with the parallel algorithms.
- We also present the first parallel algorithm for computing multinomial distribution.

ARTICLE INFO

Article history:
Received 8 September 2015
Received in revised form
2 October 2016
Accepted 1 December 2016
Available online 28 December 2016

Keywords:
Edge switch
Random network generation
Network dynamics
Multinomial distribution
Parallel algorithms

ABSTRACT

An edge switch is an operation on a graph (or network) where two edges are selected randomly and one of their end vertices is swapped with each other. 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, and in studying various dynamic phenomena over a network. The recent growth of real-world networks motivates the need for efficient parallel algorithms. The dependencies among successive edge switch operations and the requirement to keep 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 leading to difficulties in achieving a good speedup by parallelization. In this paper, we present distributed memory parallel algorithms for switching edges in massive networks. These algorithms provide good speedup and scale well to a large number of processors. A harmonic mean speedup of 73.25 is achieved on eight different networks with 1024 processors. One of the steps in our edge switch algorithms requires the computation of multinomial random variables in parallel. This paper presents the first non-trivial parallel algorithm for the problem, achieving a speedup of 925 using 1024 processors.

1. 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. Many variations of this problem have been studied [4,7,11,12,16,18,25–27,29] with diverse real-world applications. In the most commonly used edge switch operation, two randomly selected edges (a, b) and (c, d) are replaced with edges (a, d) and (c, b) respectively, i.e., the end vertices of the selected edges are swapped with each other. This operation is repeated either a given number of times or until a specified criterion is satisfied. It is easy to see that an edge switch operation preserves the degree of each vertex.

This problem has many important applications. It can be used in generating random networks with a given degree sequence.

☆ A preliminary version of this paper entitled "Fast Parallel Algorithms for Edge-Switching to Achieve a Target Visit Rate in Heterogeneous Graphs" (Bhuiyan et al., 2014, [4]) appeared in the Proceedings of the 43rd International Conference on Parallel Processing (ICPP), 2014. This work has been partially supported by DTRA CNIMS Contract HDTRA1-11-D-0016-0001, DTRA Grant HDTRA1-11-1-0016, NSF NetSE Grant CNS-1011769, NSF SDCI Grant OCI-1032677, NIH MIDAS Grant 5U01GM070694-11, NSF DIBBs Grant ACI-1443054 and NSF Big Data Grant IIS-1633028.

* Corresponding author at: Department of Computer Science, Virginia Tech, 2202 Kraft Drive, Blacksburg, VA 24061, USA.

** Corresponding author.

E-mail addresses: mhb@vbi.vt.edu (H. Bhuiyan), maleq.khan@tamuk.edu (M. Khan), chenj@vbi.vt.edu (J. Chen), mmarathe@vbi.vt.edu (M. Marathe).

http://dx.doi.org/10.1016/j.jpdc.2016.12.005
0743-7315/Published by Elsevier Inc.

There has been significant 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) [22,5,31], 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 [31,28,20] (see [5] for a good discussion). Blitzstein et al. [5] 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 the Havel–Hakimi method [15], a network can be generated following a given degree sequence. Since it 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 the Havel–Hakimi method to generate a random network with a given degree sequence [12,7,11]. Once a network is generated using the Havel–Hakimi method, by randomly switching the edges we can generate a random network with the same degree sequence. The mixing time was shown to be bounded by a large polynomial by Cooper et al. [7], and extended by Feder et al. [11] 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 [11]. Other applications of edge switch include the generation of randomly labeled bipartite graphs with a given degree sequence [18], independent realizations of graphs with a prescribed joint degree distribution using a Markov chain Monte Carlo approach [25], and studying the sensitivity of network topology on dynamics over a network such as disease dynamics over a social contact network [10].

Edge switch can be paired with additional constraints such as imposing a connectivity requirement, allowing or not allowing parallel edges and loops, etc. NetworkX [14] has a sequential implementation of edge switch that does not allow parallel edges but does allow loops, and provides the option of imposing connectivity constraints on the graph. A connectivity constraint requires a graph to remain connected after an edge switch operation. Some theoretical studies of edge switch for restricted graph classes can be found in the literature, such as the study of mixing time of the Markov chain introduced by this operation [7,12]. However, no effort was given to design parallel algorithms for switching edges in a graph. For smaller graphs, sequential implementation of edge switch suffices, but this 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.

Our contributions. In this paper, we present distributed memory parallel algorithms for switching edges in massive graphs with the constraint that the graph remains simple. The dependencies among successive edge switch operations and the requirement of keeping the graph simple lead to significant challenges in designing a parallel algorithm. Dealing with these requires complex synchronization and communication among the processors, which in turn makes it challenging to gain any speedup by parallelization. The performance of the algorithms also depends on the partitioning of the graph. We study several partitioning schemes in conjunction with the algorithms and present their respective trade-offs. A harmonic mean speedup (compared to the sequential algorithm's runtime) of 73.25 is achieved on eight different networks with 1024 processors. The algorithms require 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 the problem, and we present here a novel parallel algorithm for generating multinomial random variables, which achieves a speedup of 925 using 1024 processors.

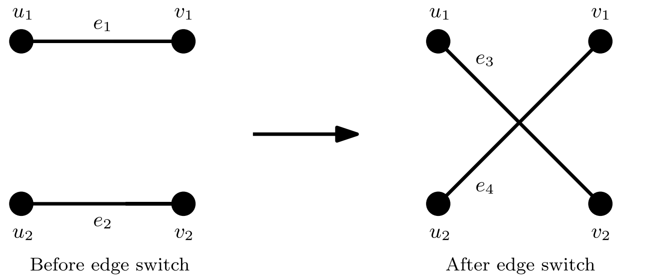


Fig. 1. An edge switch operation replaces two randomly selected edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ by new edges $e_3 = (u_1, v_2)$ and $e_4 = (u_2, v_1)$.

Organization. The rest of the paper is organized as follows. Section 2 describes the preliminaries and notations used in the paper. The edge switch problem and the sequential algorithm are briefly explained in Section 3. We present our main parallel algorithms for switching edges in Section 4. The parallel algorithm for generating multinomial random variables is presented in Section 5. Finally, we conclude in Section 6.

2. Preliminaries

Below are the notations, definitions and computation model used in this paper.

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 or 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 a total of $n = |V|$ 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*, *label* and *vertex-id* 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. For the parallel algorithms, let p be the number of processors, and P_i the processor with rank i. A summary of the frequently used notations (some of them are introduced later for convenience) is provided in Table 1.

Edge switch. An edge switch operation replaces two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$, selected 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) already exists in the graph.

Note that the set of edges E changes dynamically during the course of an edge switch process and the edges are selected from the current set of edges at a given time. During an edge switch operation, 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 switch operations and is still unchanged. (ii) *Modified edge*: any edge participating in an edge switch operation is replaced by a new edge, and such a new edge (e.g., e_3 and e_4 in Fig. 1) is called a modified edge.

Table 1
Notations used frequently in the paper.

Symbol	Description	Symbol	Description
G	Graph	H	Hundreds
V	Set of vertices	K	Thousands
n	Number of vertices	M	Millions
E	Set of edges	B	Billions
m	Number of edges	p	Number of processors
$N(u)$	Adjacency list of vertex u	P_i	Processor with rank i
d_u	Degree of vertex u	V_i	Subset of vertices in P_i
t	No. of edge switch operations	E_i	Subset of edges in P_i
x	Visit rate		

Visit rate. We define the *visit rate* as the ratio of the number of modified edges to the total number of edges in G. Let m be the number of edges in G and m' be the number of modified edges. Then the visit rate is $x = m'/m$.

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 Eq. (1). The probability of getting exactly i successes in N trials is given in Eq. (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 $0, 1, \dots, \ell - 1$ with probability $q_0, q_1, \dots, q_{\ell-1}$ respectively, such that $q_i \geq 0$ for $0 \leq i \leq \ell - 1$ and $\sum_i q_i = 1$. Let X_i be the random variable denoting the number of times the outcome i appears among N independent trials. Then $X = \langle X_0, X_1, \dots, X_{\ell-1}\rangle$ has a multinomial distribution with parameters $N, q_0, q_1, \dots, q_{\ell-1}$, and is denoted as follows.

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

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.

3. Edge switch

In this section, we first determine the expected number of edge switch operations for a given visit rate, and then we present the sequential algorithm for switching edges.

3.1. Determining the number of edges to switch for a given visit rate

Let t be the total number of edge switch operations and $T = 2t$ be the number of edges switched to achieve a visit rate x. Since edge switch is a random process, performing the same number of edge switch operations in different executions of the same edge switch algorithm may exhibit different visit rates. Thus having an exact value of T in advance is not possible. However, we can calculate the *expected* value of T as described below. As we demonstrate later in this section, using this expected value of T leads to a very close approximation of the visit rate. Finding the expected value of T 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 edge pairs. The remainder $(m-m')$ of the edges remain unchanged. At some point there are already $(i-1)$ modified edges in the graph. From this point to have the ith modified edge we need T_i number of edges switched. The probability of selecting the ith original edge from the graph, given that there are $(i - 1)$ modified edges, is $p_i = \frac{m-(i-1)}{m}$. Here, T and T_i are random variables, and T_i has a geometric distribution with expectation $1/p_i$. Using the linearity of expectation,

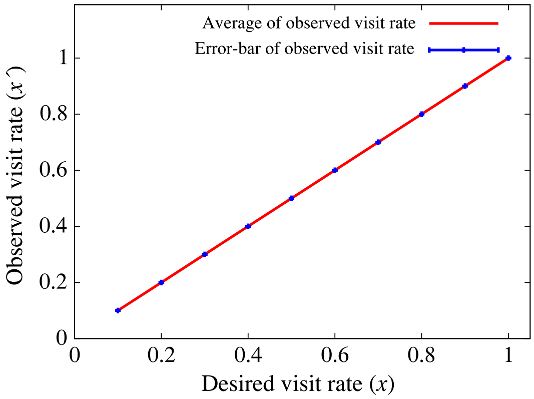


Fig. 2. Observed visit rate is almost equal to the desired visit rate for the Miami network. The error is so small that the error-bar is almost invisible.

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

where H_m is the mth harmonic number. For large m, $H_m \approx \ln m$, and consequently $E[T] \approx -m\ln(1-x)$ for $x < 1$, and $E[T] \approx m \ln m$ for $x = 1$. Note that every edge switch operation involves two edges. Now if we assign t to be $E[T]/2$, we obtain a visit rate extremely close to x as demonstrated below.

We perform experiments on a contact network of the city of Miami having $m = 52.7$M edges (see Section 4.7 for details) to achieve a visit rate of $x = 1$, i.e., visit all of the 52.7M edges. The expected value of T is calculated using $E[T] \approx m \ln m$, and the edge switch algorithm performs $t = 468.5$M edge switch operations. We repeat this experiment 10 times and observe a visit rate of $x' = 1$ (visiting all edges) for 20% of the time, $x' = 0.99999998$ (visiting all but one edge) for 60% of the time and $x' = 0.99999994$ (visiting all but three edges) for 20% of the time. Thus the observed visit rates are extremely close to x. We perform additional experiments for desired visit rates $x = 0.1, 0.2, \dots, 1$ on the Miami network. Each experiment is repeated 10 times. Fig. 2 demonstrates that the observed visit rates are almost equal to the desired visit rates. We plot the minimum and maximum of observed visit rates using error-bars. These values are so close to the desired visit rates that they almost overlap with each other and it is difficult to distinguish them in the figure. To better understand the differences

Table 2
Average error rate and standard deviation of observed visit rates for the Miami network are near 0. For each desired visit rate, 10 experiments are performed.

Desired visit rate	Observed visit rate	
	Average error rate (%)	Standard deviation
0.1	0.00745	8.13E−6
0.2	0.00858	1.41E−5
0.3	0.00907	1.76E−5
0.4	0.00802	3.52E−5
0.5	0.00687	2.34E−5
0.6	0.00650	3.38E−5
0.7	0.00701	4.37E−5
0.8	0.01030	5.55E−5
0.9	0.00824	4.46E−5
1.0	2.4E−6	2.06E−8

between the desired and observed visit rates, we further compute the average error rate and standard deviation of the observed visit rates, which are shown in Table 2. The average error rate (%) is calculated as $\frac{\sum_i |x_i - x'_i|}{e x_i}$ × 100%, where x_i and x'_i are the desired and observed visit rates, respectively, in the ith experiment and e is the total number of experiments. The maximum, minimum and average error rates of the total 100 experiments are 0.027%, 0% and 0.007%, respectively, which are almost negligible. Therefore, for large m, we achieve a very close approximation of x, which is sufficient for almost all practical purposes.

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

Furthermore, the visit rate can also be defined in other ways and converted to t. Our parallel algorithms can be used to perform t edge switch operations, irrespective of how t is obtained.

3.2. *Keeping the graph simple*

Because the edge switch problem deals with a simple graph, we need to ensure that none of the edge switch operations create self-loops or parallel edges. An edge switch between edges (u_1, v_1) and (u_2, v_2) may create a

- **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 operation does not make any change to the graph if the pair of edges remain the same after switching, and we say such an edge switch operation is *useless*. An 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 operation, two edges are selected and switched if the switch is not useless and does not create parallel edges or loops.

3.3. *Sequential edge switch*

We are given a simple graph $G = (V, E)$ and the number of edge switch operations t to be performed. The sequential algorithm is quite simple. Select a pair of edges uniformly at random and switch them if the resultant graph remains simple. This operation is repeated until t pairs of edges are switched. The graph, specifically the edge set, dynamically changes with the course of the edge switch process. 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 the 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 ith edge switch operation, then the time to switch t pairs of edges 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. Note that if an edge switch operation attempts to create a parallel edge or a loop, or is useless, the edge switch operation is restarted by selecting a new pair of edges. For a large and relatively sparse network, this probability is very small. As a result, the number of edge switch operations restarted is significantly smaller than t. Thus we have the runtime $O(t \log d_{\max})$.

Algorithm 1 SEQUENTIAL EDGE SWITCH (G, t)

```
1: for $i = 1$ to $t$ do
2:     $(u_1, v_1), (u_2, v_2) \leftarrow$ two uniform random edges in $E'$
3:     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
4:         go to line 2
5:     Replace $(u_1, v_1)$ and $(u_2, v_2)$ by $(u_1, v_2)$ and $(u_2, v_1)$ respectively
```

4. Parallel edge switch

Although the sequential algorithm is very simple, parallelizing the simple edge switch operations turns out to be a non-trivial problem for the following reasons:

- Multiple pairs of edges are selected and switched simultaneously by different processors in the parallel process, whereas the sequential process selects and switches a sequence of pairs of edges, one pair after another. Designing a parallel algorithm by maintaining a stochastic process equivalent to the sequential one leads to significant challenges.
- The requirement of keeping the graph simple requires complex synchronization and communication among the processors. To achieve a good speedup by parallelization, we need to design an efficient algorithm by minimizing such communication and computation costs.

In this section, we present an efficient parallel algorithm for switching edges in massive graphs, accompanied by a rigorous comparative study of several partitioning schemes.

4.1. *Overview of the algorithm*

The input graph G is partitioned and distributed among the p processors. Each partition contains a subset of the vertices and their adjacent edges and is assigned to a processor. All the processors then perform t edge switch operations in parallel. We need to consider two cases for an edge switch operation:

- **Local switch**. Both edges may be selected from the same partition (or processor), and this is referred to as a *local switch*.
- **Global switch**. The edges may be selected from different partitions, which is referred to as a *global switch*. The processors may need to communicate with each other in order to complete the edge switch operation.

4.2. *Data structures*

A graph can be stored as adjacency lists or as an adjacency matrix. In an adjacency matrix, the existence of any edge can be

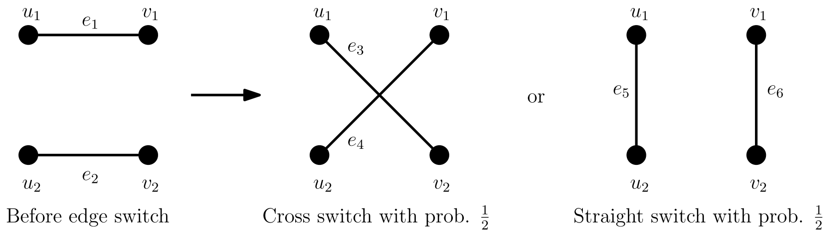


Fig. 3. Straight and cross edge switch.

determined in constant time, however it takes $O(n^2)$ space. Our algorithms use adjacency lists, which require $O(m + n)$ space. Usually, $N(u)$ contains all neighbors of u. The graph can also be presented in many different representations such as Compressed Sparse Row (CSR) [13], which also requires $O(m + n)$ space.

Reduced adjacency list. For an edge (u, v), if $N(u)$ and $N(v)$ belong to different partitions, the edge can be selected from two different partitions and participate in two different edge switch operations at the same time, leading to an inconsistency. To ensure that any edge (u, v) can be selected only from one partition, only neighbors with higher labels are kept in the adjacency list of a vertex u, i.e., $N(u) = \{v \in V | (u, v) \in E, u < v\}$, which is referred to as the *reduced adjacency list*. Although it is possible to deal with the above issue by storing all neighbors in the adjacency list, it will incur higher communication costs. Every edge switch operation involves updating four vertices' adjacency lists: one update for each end vertex of an edge. A reduced adjacency list minimizes the number of updates to only two or three vertices' adjacency lists; the details are discussed later in Section 4.4. Thus a reduced adjacency list reduces memory footprint, communication cost, and computation cost.

Straight and cross edge switch. A difficulty arises from using the reduced adjacency list. If $N(u)$ contains all the neighbors of u, any edge (u_1, v_1) can be selected either as (u_1, v_1) from $N(u_1)$ (considering ordered pair), or as (v_1, u_1) from $N(v_1)$. The probability of being selected each way is $1/2m$. Let (u_1, v_1) and (u_2, v_2) (considering no ordering) be two edges selected for an edge switch operation. Depending on whether the edge (u_1, v_1) is selected from $N(u_1)$ or $N(v_1)$, and the other edge (u_2, v_2) is chosen from $N(u_2)$ or $N(v_2)$, the edges are replaced by either (u_1, v_2) and (u_2, v_1), or (u_1, u_2) and (v_1, v_2). Assuming $u_1 < v_1$ and $u_2 < v_2$, there is no possibility of selecting edges as (v_1, u_1) and (v_2, u_2) (considering ordered pair) due to the use of a reduced adjacency list. Therefore, an edge switch between (u_1, v_1) and (u_2, v_2) (considering unordered pair) misses the chance of generating the edges (u_1, u_2) and (v_1, v_2). This problem is solved by replacing the selected edges by either (u_1, u_2) and (v_1, v_2) with probability $1/2$, referred to as *straight switch*, or (u_1, v_2) and (u_2, v_1) with probability $1/2$, referred to as *cross switch*, as shown in Fig. 3.

4.3. Partitioning the network

Partitioning schemes usually have a significant impact on the performance of parallel graph algorithms in terms of both runtime and memory. A good partitioning scheme for the parallel algorithm should have the following properties:

- It can efficiently (in terms of runtime) partition a given network.
- Given a vertex v, the partition where v belongs can be efficiently determined.
- The workload is uniformly distributed among the processors for different types of networks. The ***workload*** at a processor P_i is the number of edge switch operations P_i performs.

For a given simple graph $G = (V, E)$, we partition V into p disjoint subsets, $V_0, V_1, \dots, V_{p-1}$, such that $\bigcup_i V_i = V$. Let V_i be the subset of vertices and E_i be the subset of edges in the partition $G_i = (V_i, E_i)$ belonging to P_i such that $E_i = \{(u, v) \in E | u \in V_i, u < v\}$. The reduced adjacency list of a vertex entirely belongs to one partition. Note that the partitions are disjoint, i.e., $E_i \bigcap E_j = \emptyset$ for $i \neq j$, and $\bigcup_i E_i = E$. The subset of edges E_i at P_i dynamically changes with edge switch operations and the edges are selected from the current subset of edges at a given time.

We study four different partitioning schemes in conjunction with the algorithm and they are described below.

4.3.1. Consecutive partitioning

The graph is partitioned such that a subset of consecutive (in terms of vertex labels) vertices is assigned to each partition $G_i = (V_i, E_i)$ and each partition contains roughly m/p edges. It is easy to determine which vertex belongs to which partition. We refer to this partitioning scheme as ***Consecutive Partitioning (CP)***. Assigning a consecutive set of vertices to a partition is also known as 1D or row-wise partitioning and is employed by the Parallel Boost Graph Library [13].

4.3.2. Hash-based partitioning

Another approach can be to use a ***Hash-based Partitioning (HP)*** scheme. A hash function can be a simple algebraic expression mapping vertex labels to partitions. Hash functions are deterministic in nature, and by using some simple hash functions it can be very easy and efficient to determine which vertex belongs to which partition, thus obeying the first two criteria of a good partitioning scheme. Hash functions may assign different number of vertices and edges to partitions.

A good hash function for the partitioning schemes should have the following properties.

- It is simple and efficient to determine which vertex belongs to which partition.
- Vertices are dispersed and well-distributed among the processors, i.e., all of the partitions are almost equal in size.

Division hash function (HP-D), ***multiplication hash function (HP-M)***, and ***universal hashing (HP-U)*** are a few such hash functions and they are described below.

1. **Division hash function**. A simple hash function can be a division function (HP-D) [8]. This scheme uses the following function:

   $$h(v) = v \bmod p \tag{5}$$

   where p is the number of processors.
2. **Multiplication hash function**. Another simple hash function is a multiplication function (HP-M) [8]. The hash function is:

   $$h(v) = \lfloor p(va - \lfloor va \rfloor) \rfloor \tag{6}$$

   where $a \in (0, 1)$ is a constant. The fractional part of va is extracted by $va - \lfloor va \rfloor$ and is then multiplied by the number of processors p to determine the partition where v belongs.

Although this scheme works with any value of $a \in (0, 1)$, we use $a = (\sqrt{5} - 1)/2$ as suggested in [8] to obtain reasonably good performance.

3. **Universal hashing**. The division and multiplication hash functions are quite simple. However, their workload distributions among the processors are dependent on the vertex labels of the input graph. If there is an adversary who knows the hash function being used in advance, the adversary can artificially manipulate the graph by assigning vertex labels in such a way that the workload distribution becomes skewed. For example, many high degree vertices can be assigned to a partition making the workload at the processor containing that partition significantly higher compared to the other processors. To deal with such exploitation of hash functions by an adversary, universal hashing [8] can be a good choice. This scheme uses the following hash function:

$$h(v) = (((av + b) \bmod c) \bmod p) \tag{7}$$

where c is a large prime number such that all vertex labels are in the range $[0, c - 1]$, $a \in [1, c - 1]$ is a random integer, and $b \in [0, c - 1]$ is another random integer. Since a and b are selected randomly, this method arbitrarily selects a hash function from a large set of hash functions. As a result, there is no way for the adversary to know the exact hash function in advance or to exploit it to create a worse case scenario.

4.3.3. ParMETIS partitioning

ParMETIS [19] is a well-known MPI-based parallel library for partitioning various types of unstructured graphs. It can efficiently compute high quality partitioning of large graphs. We use the parallel multilevel k-way graph partitioning scheme and refer to this scheme as ***ParMETIS Partitioning (PP)***. Since the partitions may contain non-contiguous vertices, each processor requires $O(n)$ space to store the mapping of the vertices to partitions. To get rid of the $O(n)$ space requirement, the vertex labels can be reordered after the partitioning such that the vertices belonging to a processor are reassigned consecutive vertex labels, the lower ranked processors contain the lower vertex labels and each processor stores the starting vertex label in every processor.

4.3.4. Random partitioning

Among other options, one simple way to partition a given network is assigning vertices to partitions uniformly at random. This approach may assign almost an equal number of vertices to the partitions although the number of edges may vary among them. To determine which vertex belongs to which partition, each processor requires $O(n)$ space to store the mapping of the vertices to partitions; and the vertex labels can be reordered to eliminate the $O(n)$ space requirement (as mentioned in the PP scheme in Section 4.3.3). We refer to this scheme as ***Random Partitioning (RP)***.

4.4. Switching a pair of edges by a single processor

A simple approach to perform an edge switch operation is that processor P_i can select 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 by exchanging messages among the processors. However, this approach incurs significant synchronization and communication costs. Instead, P_i selects one edge (u_1, v_1), referred to as *first edge*, uniformly at random from E_i, and another edge (u_2, v_2), referred to as *second edge*, from the entire graph. To select a second edge, P_i selects a processor P_j with probability $|E_j|/|E|$ 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*. Due to the use of reduced adjacency lists, one of the replacing edges (e_3, e_4, e_5 or e_6 in Fig. 3) may belong to a different processor P_k ($P_i \neq P_k \neq P_j$); in this case, processors P_i, P_j and P_k work together to update the reduced adjacency lists of respective vertices by exchanging messages and thus complete the edge switch operation. A high-level overview of an edge switch operation is given in Algorithm 2. During the course of an edge switch operation, if any processor detects a possibility of creating loops or parallel edges, it notifies all other processors that are involved in the edge switch operation. Then the initiating processor (P_i in the above example) restarts the edge switch operation by selecting a new pair of edges.

Algorithm 2 Switching a Pair of Edges Initiated by P_i

```
$e_1 \leftarrow$ a uniform random edge in $E_i$
$P_j \leftarrow$ a random processor in $[0, p - 1]$, where probability of choosing $P_x$ is $\frac{|E_x|}{|E|}$
if $P_i = P_j$ then
    Choose an edge $e_2$ from $E_i$ to switch with edge $e_1$
    Switch the edges $e_1$ and $e_2$ ($P_i$ may communicate with a different processor $P_k$ to complete the edge switch operation)
else
    Send message $\langle e_1$, request to select an edge from $E_j \rangle$ to $P_j$
    Upon receipt of the above message, $P_j$ executes the following:
    Choose an edge $e_2$ from $E_j$ to switch with edge $e_1$
    $P_i$ and $P_j$ work together to switch $e_1$ and $e_2$ ($P_j$ may communicate with a different processor $P_k$ to complete the edge switch operation)
```

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 is not useless. P_i decides between a *straight* and 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 no parallel edge is created, 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 operation. 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 determines whether (u_1, u_2) and (v_1, v_2) create parallel edges. If they do not create any parallel edge, 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 operation. If $P_i \neq P_k$, P_i checks whether (u_1, u_2) creates parallel edges. If the graph remains simple, 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 back to P_i informing it of 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 operation, P_i selects an edge $e_1 = (u_1, v_1)$ from 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 $e_2 = (u_2, v_2)$ from E_j uniformly at random, and decides between a *straight* and a *cross* switch with equal probability. At this point, P_j knows the new edges that will replace e_1 and e_2; we refer to these new edges as *potential edges* until the updates take place. Next we 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. *Case* $P_k = P_j$:
P_j checks whether (u_2, v_1) creates parallel edges. If a parallel edge is not created, 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 back to P_j informing 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 operation.

ii. *Case* $P_k = P_i$:
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 no parallel edge is created, P_i removes (u_1, v_1), adds edges (u_1, v_2) and (u_2, v_1), and sends a message back to P_j indicating the updates at P_i. Then P_j completes the edge switch operation by removing (u_2, v_2).

iii. *Case* $P_i \neq P_k \neq P_j$:
P_j sends (u_2, v_1) and v_2 to P_k. If (u_2, v_1) does not create any parallel edge, P_k sends v_2 to P_i. P_i checks whether (u_1, v_2) creates any parallel edge. If the graph remains simple, P_i removes (u_1, v_1), adds (u_1, v_2), and sends messages to P_j and P_k notifying 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 operation.

A similar approach is followed for $i > j$ and for a straight switch as well. The use of reduced adjacency lists 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 $i > j$.

4.5. Simultaneous edge switches by all processors

In a sequential algorithm, pairs of edges are selected randomly, one pair after another; as a result, 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. Let X_i be the number of first edges selected from E_i by a sequential algorithm. A sequential algorithm does not need to know X_i in advance. However, for the parallel algorithm, for each i, X_i needs to be determined in advance so that processors can simultaneously perform edge switches in parallel. For any edge switch operation, the probability that the first edge is selected from E_i is $q_i = |E_i|/|E|$ 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.,

$$\langle X_0, X_1, \ldots, X_{p-1}\rangle \sim \mathcal{M}(t, q_0, q_1, \ldots, q_{p-1}). \tag{8}$$

The time complexity of the best known sequential algorithm, known as the *conditional distributed method* [9], for generating multinomial variables is $\Theta(t)$. Thus to have an efficient parallel algorithm for our edge switch problem, we need to use 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 5, we present an efficient parallel algorithm for computing multinomial random variables that runs in $O\left(\frac{t}{p} + p \log p\right)$ time.

Each processor P_i simultaneously performs X_i number of edge switches and serves other processors' requests as well. After completing one edge switch, P_i proceeds to its next edge switch operation. Below we discuss two *issues* that arise from performing edge switch operations simultaneously.

1. **Creating parallel edges in a new way**. 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, the 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.
2. **Changing probability values with the course of edge switch process**. As the edges are switched, the number of edges changes (i.e., increases or decreases) among the partitions due to the use of reduced adjacency lists. As a result, the probability values (q_i) of selecting edges from different partitions change, which need to be updated dynamically. However, updating the probability values after every edge switch operation incurs large communication costs, which in turn slows down the algorithm significantly. To deal with this difficulty, the processors perform a fixed number of edge switch operations (referred to as *step-size* and denoted by s) in a step, and then update the probability values that are used in the next step. Therefore, the algorithm performs edge switch operations in a number of steps. At the beginning of each step, s edge switch operations are distributed among p processors using the multinomial distribution. The program terminates when all of the t edge switch operations are performed in $\lceil \frac{t}{s} \rceil$ steps. With a reasonable step-size, a very close approximation of the sequential algorithm is achieved. The experimental results are shown later in Section 4.7.

Summary of the parallel algorithm. Let s be the *step-size*, and q be the probability vector $\langle q_0, q_1, \ldots, q_{p-1}\rangle$. All the processors perform s edge switch operations in one step, thus requiring a total of $\lceil \frac{t}{s} \rceil$ steps. If $t\%s \neq 0$, $(t - s\lfloor \frac{t}{s} \rfloor)$ number of edge switch operations are performed in the last step. Below is a summary of the parallel algorithm.

(1) **Generating multinomial random variables**. At the beginning of each step, s edge switch operations are distributed among p processors using the parallel algorithm for generating multinomial random variables with parameters $(s, q_0, q_1, \ldots, q_{p-1})$. This takes $O\left(\frac{s}{p} + p \log p\right)$ time. Let us denote S_i to be the number of edge switch operations that a processor P_i performs in the current step.

(2) **Performing edge switch operations**. To perform an edge switch operation, a processor P_i selects one edge e_1 from E_i and the other edge e_2 from the entire graph, and completes the edge switch operation in conjunction with other processors (details in Section 4.4). Each processor P_i simultaneously performs S_i number of edge switch operations and serves other processors' requests as well. For an edge switch operation, a constant amount of message exchange is required; edges are updated in constant time and checking for parallel edges takes $O(\log d_{\max})$ time. Thus, performing S_i edge switch operations at P_i takes $O(S_i \log d_{\max})$ time.

(3) **Updating probability vector and termination**. After completing S_i edge switch operations in the current step, P_i sends *end-of-step* signals (or messages) to each processor requiring $O(\log p)$ time. P_i continues to serve requests from other processors until receiving end-of-step signals from every processor, i.e., the end of the current step. At the end of each step, P_i receives $|E_j|$ from each P_j by exchanging messages and it takes $O(\log p)$ time. P_i updates q with the received $|E_j|$ values in $O(p)$ time. Then, in the next step, s number of edge switch operations are again distributed

among p processors using multinomial distribution with the updated q and edge switch operations are performed. This process continues until t edge switch operations are performed in $\lceil \frac{t}{s} \rceil$ steps.

4.6. *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 ith edge switch operation, one or both of the two new edges generated by the ith switch can be selected for the $(i+1)$th edge switch operation. 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 operation (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 switch operations 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 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 in Section 4.5.
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 distributions of $n_s(V_i, V_j)$ and $n_p(V_i, V_j)$, for all i, j, are the 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 containing an equal number of vertices having consecutive vertex labels. Note that the ith partitions V_i of G_s^t and G_p^t have the same set of vertices with vertex labels in $\left[\frac{i|V|}{r}, \frac{(i+1)|V|}{r} - 1\right]$ (assuming n is a multiple of r). The *edge difference* $ED(G_s^t, G_p^t)$ across different sets of vertices between G_s^t and G_p^t is computed using Eq. (9). We define the *error rate* $ER(G_s^t, G_p^t)$ between G_s^t and G_p^t as shown in Eq. (10), where the maximum value of $ED(G_s^t, G_p^t)$ 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 $ER(G_s^t, G_p^t)$ is roughly equal to $ER(G_{s1}^t, G_{s2}^t)$, then the sequential and parallel processes are said to be *similar*. For a same pair of resultant graphs G_s^t and G_p^t, the value of $ER(G_s^t, G_p^t)$ is different for different values of r. As a result, for a particular value of r, we are interested in how close $ER(G_s^t, G_p^t)$ and $ER(G_{s1}^t, G_{s2}^t)$ are to each other rather than the value of the error rate. The experimental results are explained in next section.

Table 3
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

$$ED(G_s^t, G_p^t) = \sum_{i,j \ge i} \left| n_s(V_i, V_j) - n_p(V_i, V_j) \right| \tag{9}$$

$$ER(G_s^t, G_p^t) = \frac{ED(G_s^t, G_p^t)}{2m} \times 100\%. \tag{10}$$

4.7. *Experimental results*

In this section, we present strong and weak scaling of our parallel algorithm, demonstrate the *similarity* of the sequential and parallel edge switch processes, and analyze the trade-offs among step-size, error rate, and speedup. We also present a comparative study of the performance exhibited by the partitioning schemes along with the algorithms.

Experimental setup. We use a high performance computing cluster of 64 Intel Sandy Bridge compute nodes (Dell C6220). Each computing node consists of a dual-socket Intel Sandy Bridge E5-2670 2.60 GHz 8-core processor (16 processors per node) and 64 GB of 1600 MHz DDR3 RAM. The computing nodes are interconnected by Qlogic QDR Infiniband interconnects. To implement our algorithm, we use C++ and MPICH2 implementation (version 1.9) of MPI. The CP, HP and RP schemes are implemented as part of the algorithms and we use ParMETIS [19] for the PP scheme.

Datasets. We use both real-world and artificial networks for the experiments. A summary of the networks is provided in Table 3. New York, Los Angeles, and Miami are synthetic, yet realistic, social contact networks [3]. Each vertex represents a person in the corresponding city, and each edge represents any 'physical' contact between two persons within a 24 h time period. Flickr is an image-based online community network [21]. LiveJournal is a social network blogging site [21]. The small world graph is generated using the Watts–Strogatz small world graph model [30], Erdős–Rényi is generated using the Erdős–Rényi graph model [6], and PA is generated using the Preferential Attachment graph model [2].

Strong scaling. Figs. 4, 5, 6, and 7 show the strong scaling of the parallel algorithm for edge switch using the CP, PP, HP-U, and RP schemes, respectively. The algorithm performs t edge switch operations for a visit rate of $x = 1$ using a step-size of $t/100$. We have experimented with eight different graphs, and achieved a maximum speedup of 115 with 800 processors using the RP scheme on the Miami graph. Using the RP scheme, the harmonic mean speedup is 73.25 with 1024 processors. The absolute runtime of the parallel algorithm using the HP-U scheme is shown in Fig. 8.

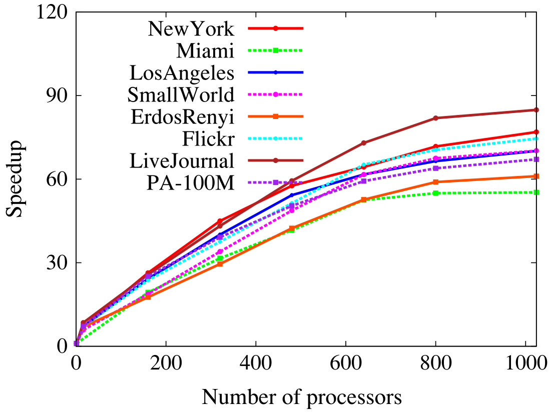


Fig. 4. Strong scaling of our algorithm on eight different graphs using the CP scheme.

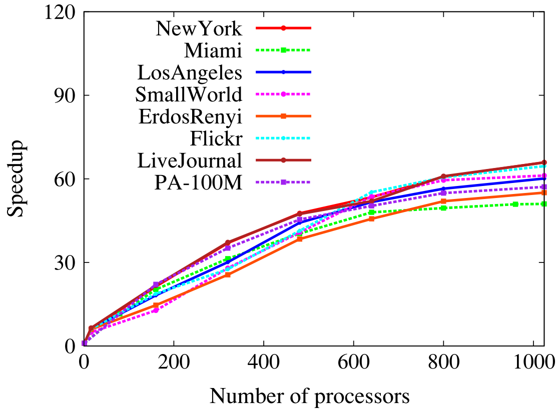


Fig. 5. Strong scaling of our algorithm on eight different graphs using the RP scheme.

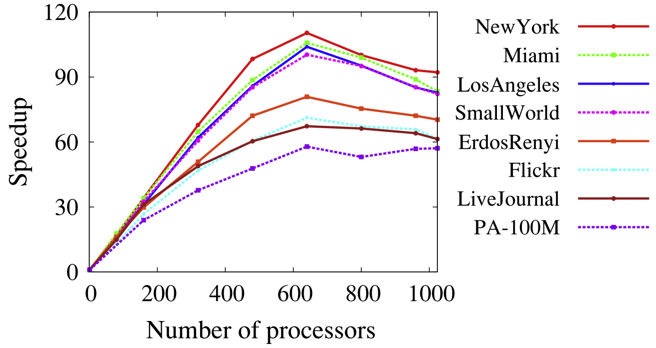


Fig. 6. Strong scaling of our algorithm on eight different graphs using the HP-U scheme.

The maximum, minimum, average and standard deviation of the speedup from 25 experiments of the parallel algorithm using the HP-U scheme for the Miami graph is shown in Table 5. All the speedups are measured against the runtime of the sequential algorithm given in Table 4. Speedup varies for different graphs because of the types of graphs and difference in workload distribution among the processors. Speedup starts decreasing after some point with the increase of the number of processors indicating the domination of communication costs over computation costs.

A comparison of strong scaling performance of the parallel algorithms using different schemes on the Miami and PA-100M graphs is demonstrated in Fig. 9. The RP scheme shows better strong scaling for the Miami graph whereas CP outperforms the other schemes for the PA-100M graph. To understand why speedup varies for different schemes and how well the schemes perform for different types of graphs, we further investigate workload distributions of different schemes for the Miami and PA-100M graphs. We use $p = 1024$ processors for the remainder of the experiments in this section.

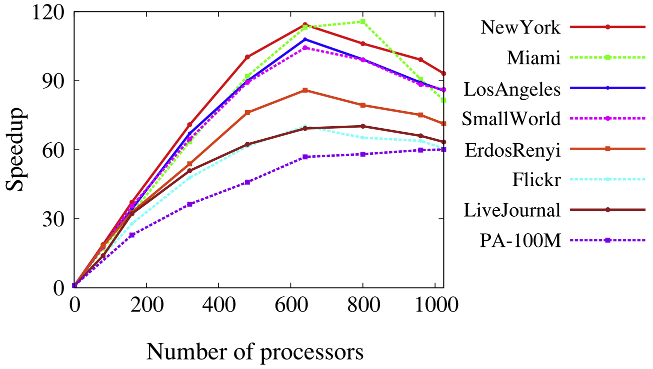


Fig. 7. Strong scaling of our algorithm on eight different graphs using the RP scheme.

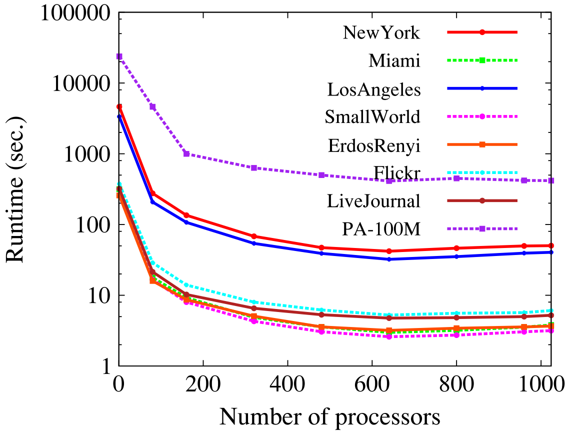


Fig. 8. Runtime of the parallel algorithm using the HP-U scheme for visit rate = 1.

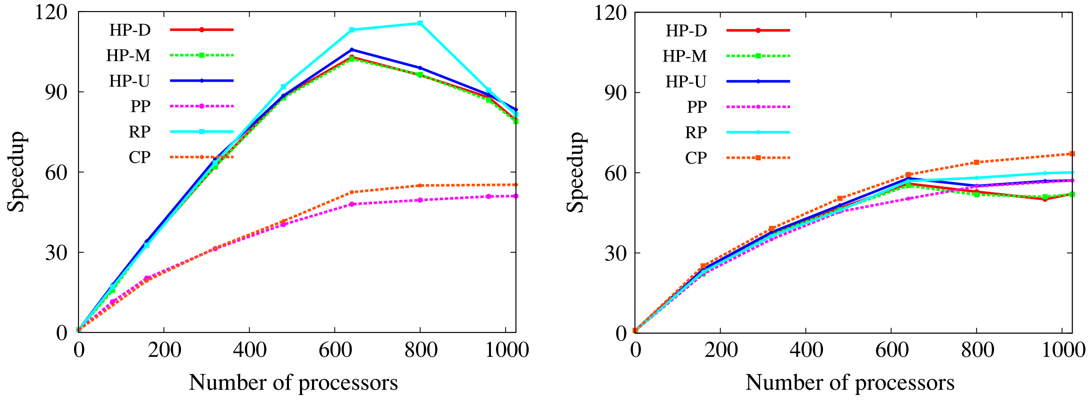
Table 4
Runtime of the sequential algorithm for visit rate = 1.

Network	Time (s)
New York	4634.6
Los Angeles	3386.5
Miami	316.3
Flickr	374.6
LiveJournal	320.0
Small world	260.1
Erdős–Rényi	258.9
PA-100M	23 849.3

Load balancing. Figs. 10 and 11 show the distributions of vertices and edges (at the beginning of execution), respectively, among the processors in different schemes on the Miami graph. The HP, RP, and PP schemes assign roughly an equal number of vertices whereas the CP scheme initially assigns almost an equal number of edges among the processors. Due to the use of reduced adjacency lists, the number of vertices assigned to the processors by the CP scheme gradually increases with the increase of processor ranks despite having an equal number of edges among the processors. The numbers of edges initially assigned to all the processors by the HP and RP schemes are very close and the distributions can be considered as roughly load balanced although are not as perfect as that of the CP scheme. On the other hand, the PP scheme considers two copies of each edge (as (u, v) and (v, u)) during the partitioning process, whereas our algorithm stores only

Table 5
Maximum, minimum, average and standard deviation of the speedup provided by the parallel algorithm using the HP-U scheme for the Miami graph. We use values of 25 experiments.

Speedup	Number of processors							
	80	160	320	480	640	800	960	1024
Maximum	17.93	34.53	65.89	90.4	109.24	103.3	94.2	88.32
Minimum	17.61	33.34	62.88	85.56	102.63	94.4	82.96	77.49
Average	17.8	34.1	64.8	88.6	105.8	98.9	88.9	83.4
Standard deviation	0.08	0.41	0.81	1.43	2.13	2.89	3.56	3.67



(a) Miami graph.

(b) PA-100M graph.

Fig. 9. A comparison of strong scaling of the parallel algorithms using different partitioning schemes for the Miami and PA-100M graphs.

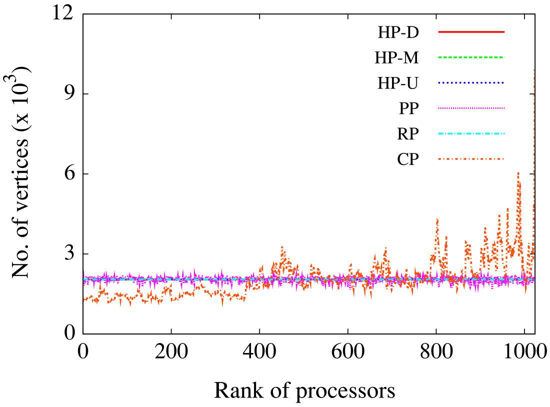


Fig. 10. Distribution of vertices among the processors in different partitioning schemes for the Miami graph.

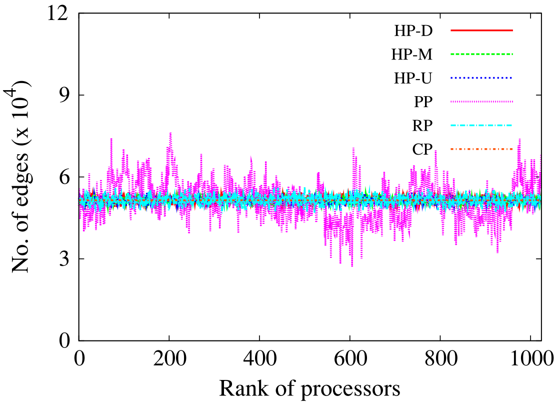


Fig. 11. Distribution of edges (at the beginning of execution) among the processors in different partitioning schemes for the Miami graph.

one copy of edge $((u, v)$ such that $u < v)$ to minimize the computation and communication costs and the memory footprint. Note that although the PP scheme may not assign contiguous vertices to partitions, in many cases, a partition produced by the PP scheme contains a large portion of vertices (compared to all the vertices assigned to that partition) having consecutive vertex labels. As a result, the partitioning by the PP scheme incurs poor distribution of edges among the processors for our edge switch algorithm.

Unlike the PP scheme, the parallel algorithms using the CP, RP, and HP schemes start the edge switch process with almost an equal number of edges at each processor as shown in Fig. 11. Recall that the number of edges gradually change among the processors with the progress of the edge switch process. As a result, at the completion of the edge switch process, the processors may end up with numbers of edges different than the numbers at the beginning of the process. Fig. 12 shows the distribution of edges at the completion of an edge switch process using different schemes on the Miami graph. The CP and PP schemes show highly skewed distributions of edges compared to that of the HP and RP schemes.

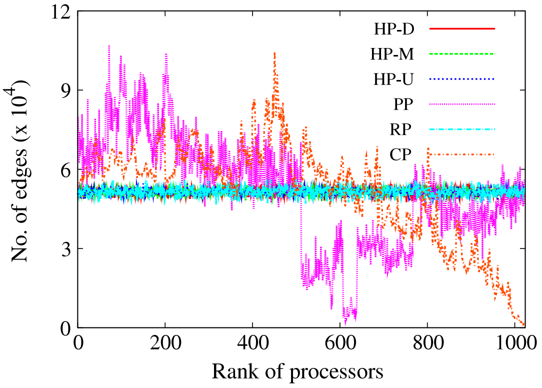


Fig. 12. Distribution of edges (after completing execution) among the processors in different partitioning schemes for the Miami graph.

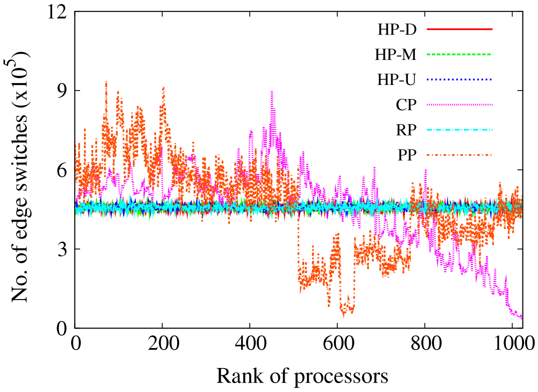


Fig. 13. Distribution of workload (number of edge switch operations) among the processors in different partitioning schemes for the Miami graph.

The skewness exhibited in the CP and PP schemes is a combined effect of the following reasons:

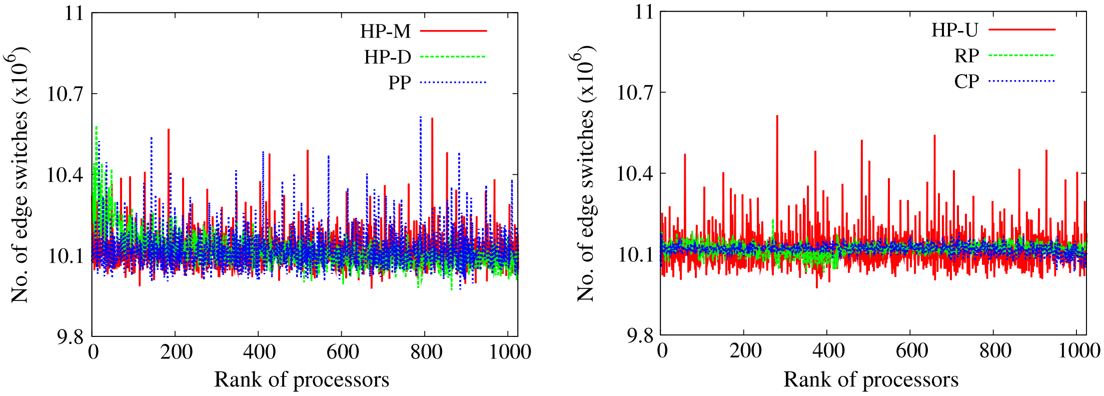
- A reduced adjacency list uses the ordering of vertex labels (from 0 to $n-1$) to store an edge (u, v): $N(u)$ stores v if and only if $u < v$.
- The same ordering of vertex labels is used to assign a consecutive subset of vertices to a partition.

For example, let (u_1, v_1) be an edge belonging to the partition in the highest ranked processor P_{p-1}, participating in an edge switch operation with another edge (u_2, v_2) belonging to the partition in P_i $(i < p-1)$. There is a probability that both replacing edges (edge e_3 and e_4, or e_5 and e_6 in Fig. 3) can belong to $N(u_2)$ and $N(v_2)$, which reside in some processors other than P_{p-1}, thus decreasing one edge from the partition in P_{p-1} and increasing one edge in the partition in P_j $(j \neq p-1)$. The occurrence of such a scenario increases for graphs having a high clustering coefficient. Note that Miami is a synthetic yet realistic contact network with maximum, minimum, and average degrees of 425, 1, and 50.4, respectively. It has a good clustering among the vertices that is gradually destroyed with the progression of the edge switch process. For the Miami graph, most of the edges in the partition belonging to the highest ranked processor are replaced by edges with one end vertex belonging to some other partition, thus destroying the clustering among the vertices in the highest ranked processor as well as decreasing the number of edges in the partition substantially. As a result, some processors contain a higher number of edges compared to other processors at the end of the edge switch process. Since the number of edge switch operations performed at a processor P_i depends on the number of edges at P_i, the skewness in the number of edges among the processors with the course of the edge switch process results in an imbalanced workload distribution as shown in Fig. 13 for the Miami graph.

In contrast, the HP and RP schemes do not assign consecutive vertices to a partition. Thus a subset of vertices having dispersed vertex labels along with their reduced adjacency lists belongs to a partition. As a result, the change in the number of edges among the partitions during the edge switch process is significantly less than that of the CP and PP schemes for the Miami graph, leading to a better workload distribution in the HP and RP schemes as shown in Fig. 13. Hence, all of the HP and RP schemes outperform the CP and PP schemes for the Miami graph, which is illustrated in Fig. 9. Among the RP and HP schemes, RP outperforms the HP schemes and HP-U outperforms the other HP schemes by a slight margin for the Miami graph.

On the other hand, Fig. 14 illustrates that the CP scheme exhibits better workload distribution for a Preferential Attachment graph having 100M vertices and 1B edges. PA graph has a very highly skewed degree distribution, i.e., it has a few very high degree and many low degree vertices. The maximum, minimum, and average degrees of the PA-100M graph are 55 225, 10, and 20, respectively. The CP scheme assigns a consecutive subset of vertices to partitions and uses the degrees of vertices to ensure that all the partitions have an equal number of edges; whereas the HP and RP schemes assign vertices to partitions using only vertex labels; they neither use the degree of vertices nor consider the number of edges already assigned to a partition. As a result, the HP and RP schemes assign several high degree vertices to some processors for the PA graph, thus making the initial edge distribution slightly more skewed compared to the CP scheme. Since PA is a random graph having a very low clustering coefficient, the number of edges initially assigned to a processor varies negligibly with the course of the edge switch process in the CP scheme. As a result, the CP scheme has an advantage of a better initial edge distribution, and thus demonstrates a better workload distribution and speedup compared to the other schemes as shown in Figs. 14 and 9 respectively.

A worse case scenario for the HP-D scheme. Unlike the CP, RP, and PP schemes, one potential disadvantage of the HP schemes is that if there is an adversary aware of the exact hash function being used as the partitioning scheme, the adversary may generate a worse case scenario by artificially manipulating vertex labels of a graph. We simulate such a scenario for the HP-D scheme using 1024 processors. We intentionally reassign vertex labels of the PA-100M graph in such a way that all of the n/p highest degree vertices are assigned to a single processor, say P_k. Thus P_k has a very high number of edges compared to the other processors despite having an equal number of vertices among the processors. As a result, P_k performs a substantially higher number of edge switch operations



(a) HP-M, HP-D and PP schemes.

(b) HP-U, CP and RP schemes.

Fig. 14. Distribution of workload (number of edge switch operations) among the processors in different partitioning schemes for the PA-100M graph.

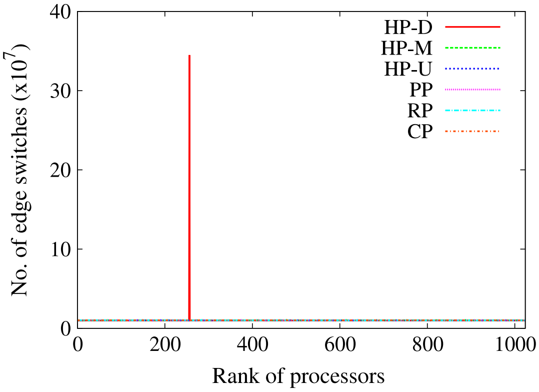


Fig. 15. A worse case scenario of distribution of workload (number of edge switch operations) among the processors for the HP-D scheme on the PA-100M graph.

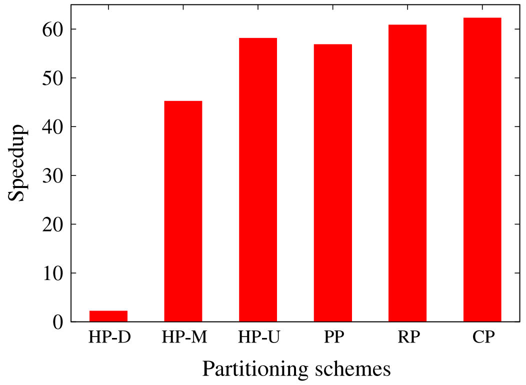


Fig. 16. A comparison of speedup of a worse case scenario for the HP-D scheme with other schemes on a PA-100M graph with 1024 processors.

compared to that of the other processors as shown in Fig. 15 (in this example, P_k is the processor with rank 256), whereas other schemes show good performance by executing faster on the same graph as shown in Fig. 16. An adversary can generate a similar worse case scenario for the HP-M scheme as well.

Advantage of the HP-U and RP schemes. Universal hashing randomly selects a hash function from a large set of hash functions. As a consequence, there is no way for an adversary to know in advance exactly which hash function will be used. Therefore, the HP-U scheme overcomes the drawbacks of the HP-D and HP-M schemes. The RP scheme also has the same advantage of randomly assigning vertices to partitions. In addition, HP-U and RP demonstrate good speedups for all types of graphs and outperform the other schemes in many cases. The HP-U does not require reordering the vertex labels after partitioning to eliminate the $O(n)$ space requirement for storing the mapping information of vertices like the PP and RP schemes. Although the CP scheme exhibits the best performance for the PA-100M graph, speedups achieved by the HP-U and RP schemes are very close to that of CP, justifying HP-U and RP as good choices in general. The PP scheme exhibits the poorest performance among all the schemes, because of poor load distribution among processors.

Similarity of the outcomes of the parallel and sequential algorithms and determining suitable step-size. For convenience, we first present the effect of step-size on the parallel algorithm using the CP scheme and then we discuss the effect of step-size for other schemes. We use visit rate $x = 1$, current calendar time as random seed, $r = 20$ partitions, $p = 1024$ processors, and average value of 10 experiments.

Effect of step-size on the CP scheme. Fig. 17 shows that better strong scaling is achieved for a larger step-size on the Miami graph.

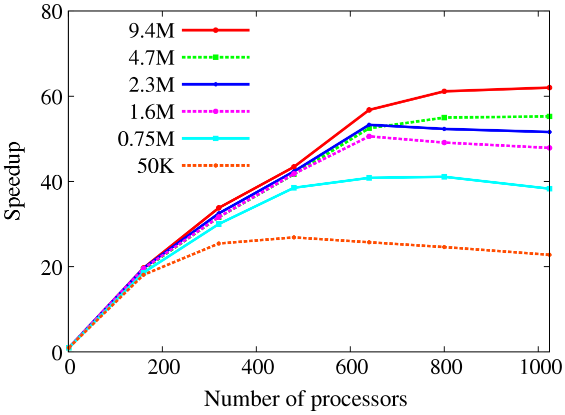


Fig. 17. A comparison of strong scaling performance on the Miami graph for different step-sizes: 9.4M, 4.7M, 2.3M, 1.6M, 0.75M, and 50K.

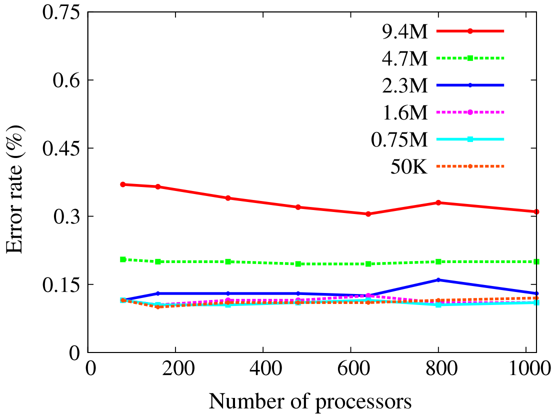


Fig. 18. Error rate with increasing number of processors on the Miami graph using different step-sizes: 9.4M, 4.7M, 2.3M, 1.6M, 0.75M, and 50K.

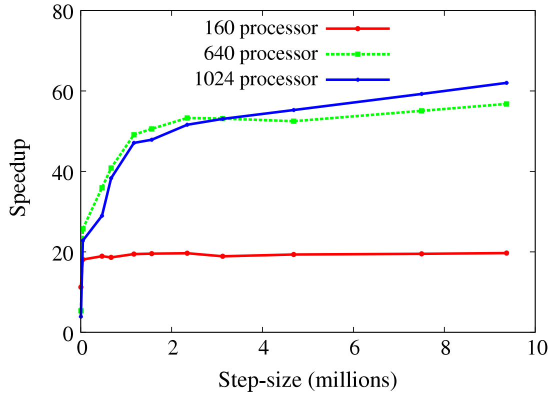


Fig. 19. Speedup with increasing step-size on the Miami graph using 160, 640, and 1024 processors.

For a particular step-size, error rate remains roughly constant with the increase of processors on the Miami graph, as shown in Fig. 18. The effects of step-size on speedup and error rate for the Miami graph are shown in Figs. 19 and 20, respectively. Both the speedup and error rate increase with the increase of step-size.

While keeping the error rate to a minimum, we want to achieve as much speedup as possible. From Fig. 20, we observe that with up to a step-size of 2M, the error rate between the resultant graphs generated by the sequential and parallel algorithms is roughly the same as the error rate between the resultant graphs generated by two different executions of the sequential algorithm. Hence, 2M can be a *suitable* step-size for the Miami graph, since the error rate

Table 6
Error rate comparison of the outcomes of the parallel algorithms using different partitioning schemes with that of the sequential algorithm for different graphs. We use the average of 10 experiments.

Networks	Error rates (%) for different schemes										
	Sequential	Parallel									
		Using 1 step				Using 100 steps					
		HP-D	HP-M	HP-U	RP	HP-D	HP-M	HP-U	RP	CP	PP
Miami	0.117	0.118	0.123	0.117	0.117	0.111	0.127	0.123	0.120	0.164	0.175
Small World	0.112	0.100	0.112	0.119	0.116	0.106	0.118	0.109	0.115	0.115	0.121
LiveJournal	0.116	0.117	0.118	0.117	0.117	0.116	0.116	0.116	0.1177	0.115	0.126

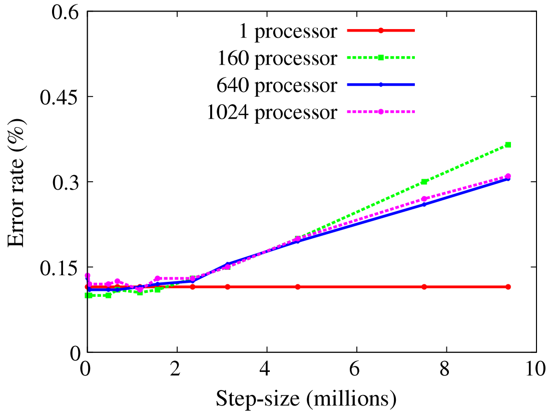


Fig. 20. Error rate with increasing step-size on the Miami graph using 1, 160, 640, and 1024 processors.

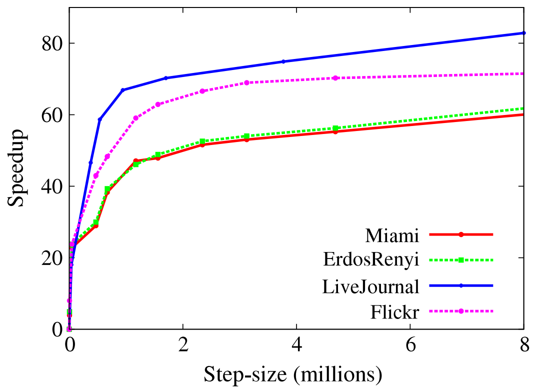


Fig. 21. Speedup with increasing step-size for different graphs using 1024 processors.

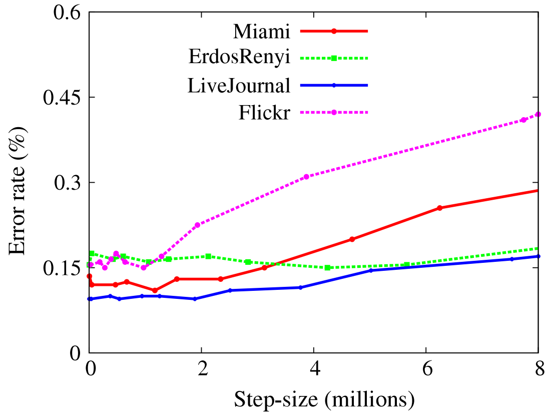


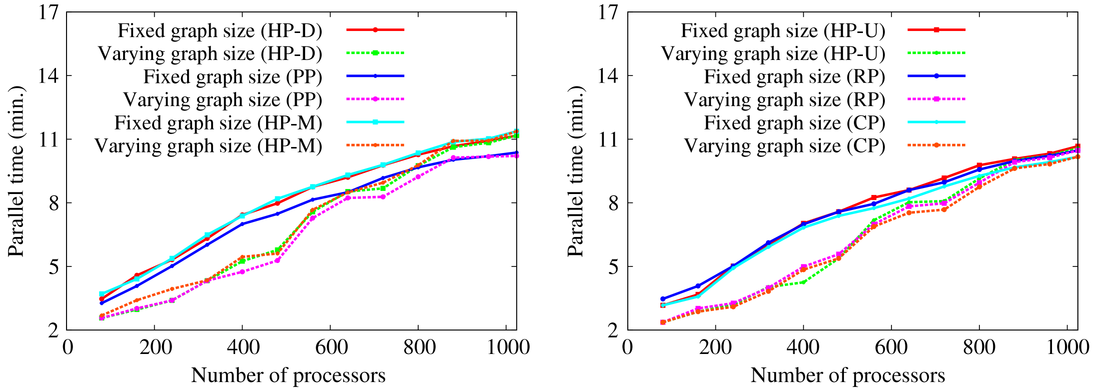
Fig. 22. Error rate with increasing step-size for different graphs using 1024 processors.

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. Figs. 21 and 22 illustrate the effect of step-size on speedup and error rate, respectively, 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 the Flickr and Miami graphs as shown in Fig. 22. A *suitable* step-size for the 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.

Effect of step-size on other schemes. Table 6 shows the error rate comparison of the outcomes of the parallel algorithms using different schemes, with that of the sequential one suggesting that even for performing edge switch operations in one step, the outcomes of the parallel algorithms using the HP and RP schemes are similar to that of the sequential algorithm with a negligible error rate deviation. Since the HP and RP schemes assign vertices dispersedly among the partitions, the number of edges initially belonging to a partition changes negligibly with edge switch operations compared to that of the CP and PP schemes. Hence the HP and RP schemes can perform edge switch operations in only one step with reasonable accuracy, which consequently makes computation faster. As a result, the parallel algorithms using the HP and RP schemes no longer need a suitable step-size. In contrast, finding a suitable step-size is important for the CP and PP schemes to obtain a close approximation of the outcome of the sequential algorithm.

Weak scaling. Fig. 23 shows weak scaling comparison of different schemes on the PA graphs. In one experiment, we increase the graph size with the increase of processors, and use the Preferential Attachment graphs with ($p \times 0.1$M) 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 $t = p \times 10$M and step size $= t/1000$. Ideally, the parallel runtime should remain constant. However, in practice the communication increases with the increase in the number of processors, leading to a higher runtime. Our algorithm shows good weak scaling as the runtime increases linearly in both the cases.

How do network properties change with switching edges? We also analyze how some network properties change with edge switch operations by the sequential and parallel algorithms. We use the Miami, LiveJournal, and Flickr graphs, and vary the visit rate from 0.1 to 1. Figs. 24 and 25 show that the average clustering coefficient and average shortest path distance of a graph change in exactly the same way with edge switches by the sequential and parallel algorithms. Average clustering coefficient measures the



(a) HP-D, PP and HP-M schemes.

(b) HP-U, RP and CP schemes.

Fig. 23. Weak scaling comparison of the parallel algorithms using various partitioning schemes with fixed and varying size PA graphs. In one experiment, we use a fixed graph having 102.4M vertices and 1.024B edges while in the other experiment, we increase (or vary) graph size with the increase of processors. The varying graphs have ($p \times 0.1$M) vertices and an average degree of 20, where p is the number of processors. For both experiments, we use $t = p \times 10$M and step-size $= t/1000$.

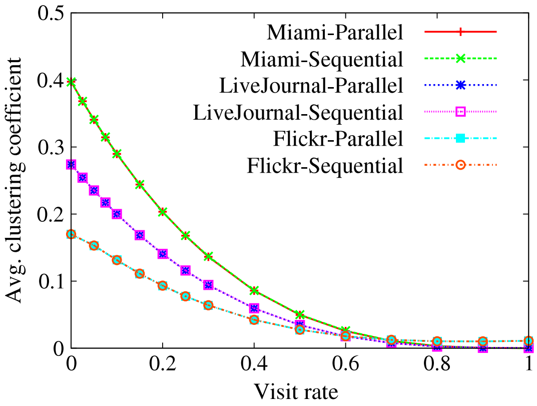


Fig. 24. Average clustering coefficient changes similarly with edge switch operations by the sequential and parallel algorithms.

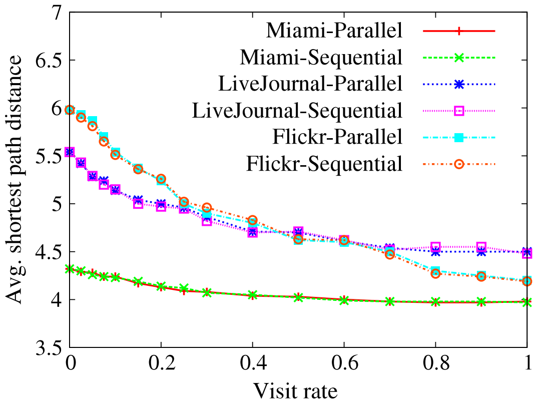


Fig. 25. Average shortest path distance changes similarly with edge switch operations by the sequential and parallel algorithms.

tendency of the vertices to cluster together or create tightly knit groups, which is a pervasive phenomena in many social networks where people with common friends tend to be friends themselves. With the progress of an edge switch process, the edges are replaced by random edges. Therefore, the clustering among the vertices gets destroyed rapidly, which eventually reaches very close to 0 for a visit rate of 1.0. As the edges are switched, the vertices further get connected by shorter paths, thus reducing the average shortest path distance as well. For both properties, the changes by the sequential and parallel algorithms are very similar; in fact, they overlap with each other and it is difficult to distinguish them in the figures.

Generating assortative networks by switching edges. We demonstrate how edge switch can be used to generate assortative networks. In a labeled network, each vertex u has an associated attribute $L(u)$. Such attributes can be, for example, the age of people in a contact network or the degree of the vertices. Adding vertex attribute constraints with edge switch leads to many interesting problems. Xulvi-Brunet et al. [32] proposed one such algorithm to produce assortative mixing to a desired degree by imposing constraints on vertex attributes during an edge switch process. *Assortative mixing* is an important network feature measuring the tendency of vertices to associate with similar or dissimilar vertices and is quantified by a metric named the *assortative coefficient (r)* [23,24]. In other words, assortativity measures the correlation of vertices based on vertex attributes. A network is called an *assortative network* if $r > 0$.

In this demonstration, we use the degree of a vertex as its attribute, i.e., $L(u) = d_u$. The level or extent of assortative mixing is controlled by a parameter p ($0 \leq p \leq 1$). Then an edge switch operation selects two edges randomly with the four end vertices having different degrees in general. The four vertices are ordered based on their degrees. Then with probability p, an edge switch operation connects the two higher degree vertices with an edge and the two lower degree vertices with another edge. With probability $(1 - p)$, the edges are switched randomly. This algorithm generates a random network with parameter $p = 0$. With the increase of p, the assortativity increases and it reaches a maximum value for $p = 1$ (see [32] for a good discussion). We apply the principle of [32] with our parallel algorithm for switching edges to generate assortative networks and present the results below.

Fig. 26 shows how the assortative coefficient changes with the edge switch process for different values of p on the Miami graph. For $p = 1$, we obtain a maximum assortative coefficient value of 0.999992, beyond which the assortativity does not increase due to the restriction imposed by the degree distribution. Fig. 27 shows the speedup obtained by the parallel algorithm using the HP-U scheme for performing 300M edge switch operations on the Miami graph.

Synopsis of the experimental results. All of the partitioning schemes demonstrate reasonably good performance. Below is a summary of the results.

- The hash-based and random partitioning schemes exhibit better performance for many graphs because of a well-balanced workload distribution.
- The HP and RP schemes can perform edge switch operations in only one step with reasonable accuracy, thus eliminating the

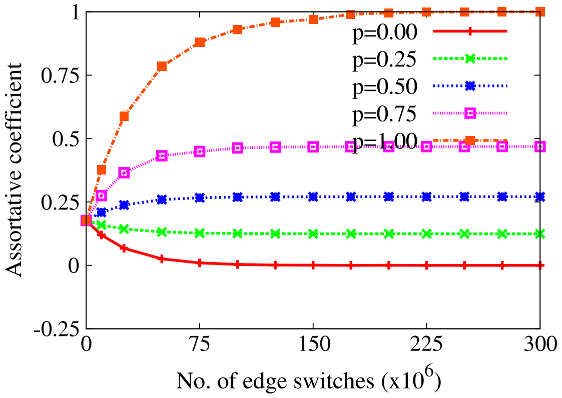


Fig. 26. Change of assortative coefficient (considering the degree of a vertex as its attribute) with the edge switch process on the Miami graph. The parameter p is varied from 0 to 1.

need for performing edge switch operations in a number of steps.

- There is a possibility of a worse case scenario arising for the HP-D and HP-M schemes that may slow down the algorithms significantly. The HP-U scheme overcomes this drawback by randomly choosing a hash function from a large set of hash functions. Like the HP-U scheme, the CP, PP, and RP schemes are also not vulnerable to adversaries for generating worse case scenarios.
- The CP scheme shows good performance with some computation overhead by performing edge switch operations with a suitable step-size for all types of graphs, and in some cases (e.g., PA-100M) outperforms the other schemes.
- The PP scheme exhibits the poorest performance among all the schemes due to poor workload distribution. It also requires to perform edge switch operations with a suitable step-size.
- Unlike the HP and CP schemes, the PP and RP schemes need to reassign the vertex labels after partitioning to get rid of the $O(n)$ space requirement at each processor to store the mapping of vertices to partitions.

5. Parallel algorithm for computing 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.

5.1. 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 the *conditional distributed method* [9], 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:

$$X_i \sim \mathcal{B}\left(N - \sum_{j=0}^{i-1} X_j, \frac{q_i}{1 - \sum_{j=0}^{i-1} q_j}\right). \tag{11}$$

The inverse transformation method (BINV) [17] 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.

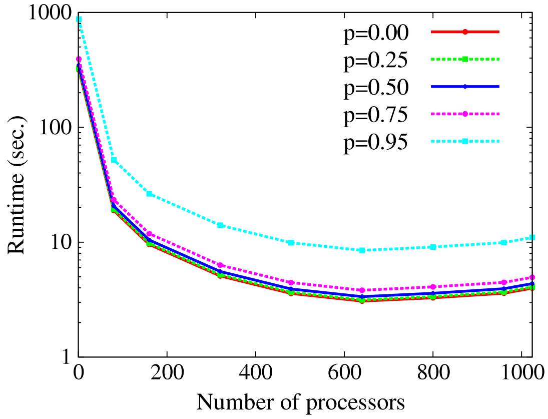


Fig. 27. Speedup gained by the parallel algorithm of edge switch for different values of p. The algorithm performs 300M edge switch operations on the Miami graph.

The algorithms for the inverse transformation method (BINV) [17] to generate binomial random variables and for the conditional distributed method [9] to generate multinomial random variables are shown in Algorithms 3 and 4, respectively. For additional details, see [17,9].

Algorithm 3 BINOMIAL(N, q)

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

5.2. 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 $\langle q_0, q_1, \ldots, q_{\ell-1}\rangle$. 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_i due to the dependency 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

$$X = \sum_{i=0}^{k-1} X_i \sim \mathcal{B}\left(\sum_{i=0}^{k-1} N_i, q\right) = \mathcal{B}(N, q). \tag{12}$$

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

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

for $0 \le i < k$, then

$$\langle X_0, X_1, \ldots, X_{\ell-1}\rangle \sim \mathcal{M}(N, q_0, q_1, \ldots, q_{\ell-1}) \tag{13}$$

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 assigns 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($N, q_0, \ldots, q_{\ell-1}$)

```
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: $\langle X_{0,i}, X_{1,i}, \ \ldots\ , X_{\ell-1,i}\rangle \sim \mathcal{M}(N_i, q_0, q_1, \ \ldots\ , q_{\ell-1})$
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 the 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 processors.

The parallel computation is almost perfectly load balanced among the processors since each processor computes multinomial distribution of N/p independently, taking $O\left(\frac{N}{p}\right)$ 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 a 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 numerical underflow for large values of N, e.g., billions. Using the *long double* data type cannot solve this underflow problem 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 Eq. (12). The upper threshold value of N_i is set such that no underflow occurs, that is,

$$(1-q)^{N_i} \ge z \tag{14}$$

$$N_i \le \frac{-\log z}{\log(1-q)} \le \frac{-\log z}{2q} \tag{15}$$

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

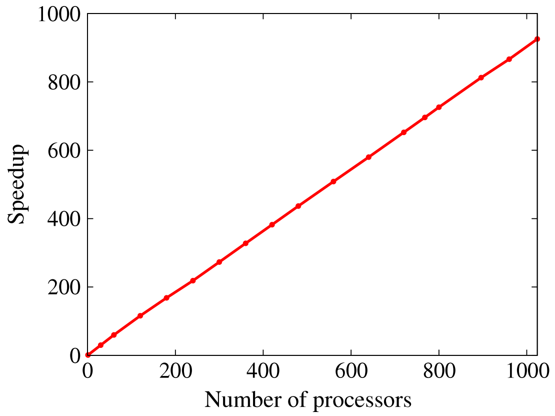


Fig. 28. Strong scaling of the parallel algorithm of multinomial distribution using $N = 10\,000$B, $\ell = 20$ and $q_i = 1/\ell$.

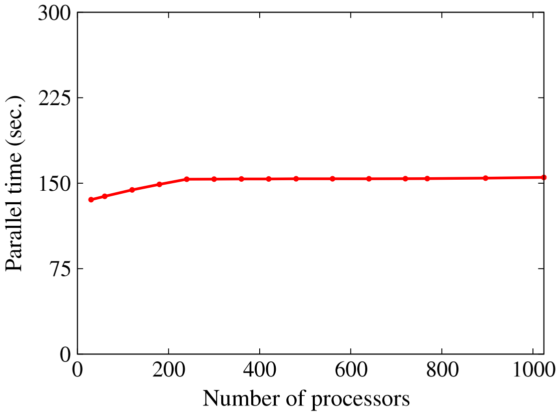


Fig. 29. Weak scaling of the parallel algorithm of multinomial distribution using $N = p \times 20$B, $\ell = p$ and $q_i = 1/\ell$.

5.3. Performance analysis of the parallel algorithm

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

Strong scaling. The strong scaling of the parallel algorithm is illustrated in Fig. 28. We keep the problem size fixed ($N = 10\,000$B, $\ell = 20$ and $q_i = 1/\ell$), and achieve a speedup of 925 using 1024 processors. The speedup increases almost linearly with the increase of processors. The parallel algorithm can compute a multinomial distribution of 10 000B in 71 s using 1024 processors.

Weak scaling. Fig. 29 shows the weak scaling of our parallel algorithm. We use $\ell = p$ (i.e., total number of processors), $N = p \times 20$B (i.e., 20B per processor), and equal probability values, $q_i = 1/\ell$. The parallel runtime is almost constant indicating a very good weak scaling.

6. Conclusion

We presented parallel algorithms for switching edges in massive networks. They can be used in studying various properties of large dynamic networks as well as in generating massive scale random graphs. The algorithms scale well to a large number of processors and exhibit good speedup. We also presented the trade-offs of several partitioning schemes. We demonstrated an application of our parallel algorithms to generate assortative networks. In addition, we developed a parallel algorithm for generating multinomial random variables that is almost perfectly parallelized. This algorithm can be of independent interest and prove useful in parallelizing many other stochastic processes. We believe that the parallel algorithms will contribute significantly when dealing with big data, one of the most challenging problems in today's research world.

Acknowledgments

We thank our external collaborators, members of the Network Dynamics and Simulation Science Laboratory (NDSSL), and anonymous reviewers for their suggestions and comments. We are grateful to Anil Vullikanti for interesting discussions and helpful comments on a draft of this paper. We also sincerely thank Maureen Lawrence-Kuether, Lenwood S. Heath, and Jim Walke for proof-reading this paper.

References

[1] I. Adler, S. Oren, S. Ross, The coupon-collector's problem revisited, J. Appl. Probab. 40 (2) (2003) 513–518.

[2] A. Barabási, R. Albert, Emergence of scaling in random networks, Science 286 (5439) (1999) 509–512.

[3] C. Barrett, R. Beckman, M. Khan, V. Kumar, M. Marathe, P. Stretz, T. Dutta, B. Lewis, Generation and analysis of large synthetic social contact networks, in: Proceedings of the 2009 Winter Simulation Conference, WSC, 2009, pp. 1003–1014.

[4] H. Bhuiyan, J. Chen, M. Khan, M. Marathe, Fast parallel algorithms for edge-switching to achieve a target visit rate in heterogeneous graphs, in: Proceedings of the 43rd International Conference on Parallel Processing, ICPP, IEEE, 2014, pp. 60–69.

[5] J. Blitzstein, P. Diaconis, A sequential importance sampling algorithm for generating random graphs with prescribed degrees, Internet Math. 6 (4) (2011) 489–522.

[6] B. Bollobás, Random Graphs, Springer, 1998.

[7] C. Cooper, M. Dyer, C. Greenhill, Sampling regular graphs and a peer-to-peer network, Combin. Probab. Comput. 16 (4) (2007) 557–593.

[8] T. Cormen, Introduction to Algorithms, MIT press, 2009.

[9] C. Davis, The computer generation of multinomial random variates, Comput. Stat. Data Anal. 16 (2) (1993) 205–217.

[10] S. Eubank, A. Vullikanti, M. Khan, M. Marathe, C. Barrett, Beyond degree distributions: Local to global structure of social contact graphs, in: Proceedings of the Third International Conference on Social Computing, Behavioral Modeling, and Prediction, SBP, 2010, p. 1.

[11] T. Feder, A. Guetz, M. Mihail, A. Saberi, A local switch markov chain on given degree graphs with application in connectivity of peer-to-peer networks, in: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, FOCS, 2006, pp. 69–76.

[12] C. Gkantsidis, M. Mihail, E. Zegura, The markov chain simulation method for generating connected power law random graphs, in: Proceedings of the Fifth Workshop on Algorithm Engineering and Experiments, ALENEX, Vol. 111, SIAM, 2003, pp. 16–25.

[13] D. Gregor, A. Lumsdaine, The parallel BGL: A generic library for distributed graph computations, Parallel Object-Oriented Sci. Comput., POOSC 2 (2005) 1–18.

[14] A. Hagberg, P. Swart, D. Schult, Exploring network structure, dynamics, and function using NetworkX, in: Proceedings of the 7th Python in Science Conference, SciPy, 2008, pp. 11–15.

[15] S. Hakimi, On realizability of a set of integers as degrees of the vertices of a linear graph, J. Soc. Ind. Appl. Math. 10 (3) (1962) 496–506.

[16] M. Jerrum, A. Sinclair, Fast uniform generation of regular graphs, Theoret. Comput. Sci. 73 (1) (1990) 91–100.

[17] V. Kachitvichyanukul, B. Schmeiser, Binomial random variate generation, Commun. ACM 31 (2) (1988) 216–222.

[18] R. Kannan, P. Tetali, S. Vempala, Simple markov-chain algorithms for generating bipartite graphs and tournaments, Random Struct. Algorithms 14 (4) (1999) 293–308.

[19] G. Karypis, K. Schloegel, V. Kumar, ParMETIS: Parallel graph partitioning and sparse matrix ordering library, Version 1.0, Dept. of Computer Science, University of Minnesota, 1997.

[20] J. Kim, V. Vu, Sandwiching random graphs: universality between random graph models, Adv. Math. 188 (2) (2004) 444–469.

[21] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection, (Jun. 2014). http://snap.stanford.edu/data.

[22] M. Newman, The structure and function of complex networks, SIAM Rev. 45 (2) (2003) 167–256.

[23] M. Newman, Assortative mixing in networks, Phys. Rev. Lett. 89 (20) (2002) 208701.

[24] M. Newman, Mixing patterns in networks, Phys. Rev. E 67 (2) (2003) 026126.

[25] J. Ray, A. Pinar, C. Seshadhri, Are we there yet? When to stop a markov chain while generating random graphs, in: Proceedings of the 9th Workshop on Algorithms and Models for the Web Graph, WAW, Springer, 2012, pp. 153–164.

[26] I. Stanton, A. Pinar, Constructing and sampling graphs with a prescribed joint degree distribution, J. Exp. Algorithmics, JEA 17 (3) (2012) 3–5.

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

[28] A. Steger, N. Wormald, Generating random regular graphs quickly, Combin. Probab. Comput. 8 (04) (1999) 377–396.

[29] L. Tabourier, C. Roth, J. Cointet, Generating constrained random graphs using multiple edge switches, J. Exp. Algorithmics, JEA 16 (1) (2011) 1–7.

[30] D. Watts, S. Strogatz, Collective dynamics of 'small-world' networks, Nature 393 (6684) (1998) 440–442.

[31] N. Wormald, Models of random regular graphs, in: London Mathematical Society Lecture Note Series, 1999, pp. 239–298.

[32] R. Xulvi-Brunet, I. Sokolov, Reshuffling scale-free networks: From random to assortative, Phys. Rev. E 70 (6) (2004) 066102.

Hasanuzzaman Bhuiyan is a Ph.D. student in the Department of Computer Science, Virginia Tech and a Graduate Research Assistant in the Network Dynamics and Simulation Science Laboratory at the Biocomplexity Institute of Virginia Tech. He received his B.S. in Computer Science and Engineering from Bangladesh University of Engineering and Technology and his M.S. in Computer Science from Virginia Tech. His research interests include high-performance computing, parallel and distributed computing, graph algorithms and data mining with real-world applications in network science. His current research includes designing and developing efficient and scalable parallel algorithms for big data analytics.

Maleq Khan is currently an Assistant Professor in the Department of Electrical Engineering and Computer Science at Texas A&M University—Kingsville. He received his Ph.D. in Computer Science from Purdue University. His research interests are parallel and distributed computing, big data analytics, high performance computing, data mining, and in the design and analysis of algorithms, specifically distributed algorithms, parallel algorithms, randomized algorithms, and graph algorithms. He has published a large number of papers in these areas. One of his papers received a best paper award at the Symposium on Distributed Computing (DISC), a flagship conference on distributed computing, and another of his papers was a best paper award finalist at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC16).

Jiangzhuo Chen is a Research Associate Professor in the Network Dynamics and Simulation Science Laboratory at the Biocomplexity Institute of Virginia Tech. He received his B.A. in Economics from Nanjing University, his M.A. in Economics from Boston College, and his Ph.D. in Computer Science from Northeastern University. His research interests include big data analytics; model based forecasting; modeling, simulation, and analysis of large scale social networks; computational epidemiology; computational economics; and approximation algorithms for network optimization problems. His current research includes high performance simulation of social network dynamics; modeling of synthetic population and social network; and forecasting of epidemics.

Madhav Marathe is the Director of the Network Dynamics and Simulation Science Laboratory at the Biocomplexity Institute of Virginia Tech and Professor of Computer Science at Virginia Tech. He has over ten years of experience in project leadership and technology development, specializing in high performance computing algorithms and software environments for simulating and analyzing socio-technical network science. He is the recipient of the Distinguished Copyright award for TRANSIMS software, Los Alamos National Laboratory's achievement award, a recipient of the University at Albany Distinguished Alumni Award and 2010 Award for Research Excellence, Virginia Bioinformatics Institute. He is the 2011 Inaugural George Michael Distinguished Scholar at the Lawrence Livermore National Laboratory. In 2013 he became an ACM Fellow and IEEE Fellow. In 2014, he was named an AAAS Fellow.