Fast Random Graph Generation

Sadegh Heyrani Nobari Xuesong Lu Panagiotis Karras Stéphane Bressan
National University of Singapore
{snobari,xuesong,karras,steph}@nus.edu.sg

ABSTRACT

Today, several database applications call for the generation of random graphs. A fundamental, versatile random graph model adopted for that purpose is the Erdős-Rényi $\Gamma_{v,p}$ model. This model can be used for directed, undirected, and multipartite graphs, with and without self-loops; it induces algorithms for both graph generation and sampling, hence is useful not only in applications necessitating the generation of random structures but also for simulation, sampling and in randomized algorithms. However, the commonly advocated algorithm for random graph generation under this model performs poorly when generating large graphs, and fails to make use of the parallel processing capabilities of modern hardware. In this paper, we propose PPreZER, an alternative, data parallel algorithm for random graph generation under the Erdős-Rényi model, designed and implemented in a graphics processing unit (GPU). We are led to this chief contribution of ours via a succession of seven intermediary algorithms, both sequential and parallel. Our extensive experimental study shows an average speedup of 19 for PPreZER with respect to the baseline algorithm.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and networks

General Terms

Algorithms, Design, Experimentation

Keywords

Erdős-Rényi, Gilbert, random graphs, parallel algorithm

1. INTRODUCTION

Several data management applications call for the generation of random graphs. For example, such generation is needed for the synthesis of data sets aiming to evaluate efficiency and effectiveness of algorithms [2], for simulating processes [21], and at the heart of randomized algorithms [34, 31]. Furthermore, a random-graph generation process can be leveraged for sampling. Thanks to the versatility of graphs as data representation model, random graph generation processes are also relevant in applications ranging from physics and biology to sociology.

Two simple, elegant, and general mathematical models are instrumental in random graph generation. The former, noted[1] as $\Gamma_{v,e}$ [8], chooses a graph uniformly at random from the set of graphs with v vertices and e edges. The second model, noted as $\Gamma_{v,p}$ [11], chooses a graph uniformly at random from the set of graphs with v vertices where each edge has the same independent probability p to exist. Paul Erdős and Alfréd Rényi proposed the $\Gamma_{v,e}$ model [8], while E. N. Gilbert proposed, at the same time, the $\Gamma_{v,p}$ model [11]. Nevertheless, both models are commonly referred to as Erdős-Rényi models.

To date, this model has been widely utilized in many fields. Examples include communication engineering [6, 10, 27], biology [28, 30] and social network studies [9, 19, 32]. In [6], the authors study the dissemination of probabilistic information in random graphs. They propose a generic method and apply it in Erdős-Rényi graphs to get upper and lower bounds for the probability of the global outreach. The authors of [10] investigate the dispersal of viruses and worms in Erdős-Rényi networks because its epidemic dispersing behavior determines the dispersing behavior of graphs following a power law. In [27], uniform random graphs based on the Erdős-Rényi model are utilized to explore data search and replication in peer-to-peer networks. They conclude that uniform random graphs perform the best. In the field of biology, the Erdős-Rényi model is accepted as a basic model to study biological networks; [28] assesses the similarity of the topologies between biological regulatory networks and monotone systems, and uses the Erdős-Rényi model is used as a model to compare against, when studying the properties of loops in three intracellular regulatory networks. In [30] this model is again applied to explore how activating and inhibiting connections influence network dynamics. The Erdős-Rényi model is also used as a classic model in social studies [19]. In [9], the heritability in degree and transitivity is tested on Erdős-Rényi networks together with other models in order to investigate genetic variation in human social networks. In [32], a class of models that are generalizations of the Erdős-Rényi model is analyzed and applied to

[1]We adopt the original notation of [8]. Some other authors write $\mathcal{G}(v,e)$ or $G_{v,e}$. Similarly, we write $\Gamma_{v,p}$ where some other authors write $\mathcal{G}(v,p)$ or $G_{v,p}$.

EDBT 2011, March 22–24, 2011, Uppsala, Sweden.

model social networks. Besides, these models are also used for sampling. Namely, $\Gamma_{v,e}$ is a uniform random sampling of e elements from a set of v. $\Gamma_{v,p}$ is a fixed, independent, and equiprobable random sampling of edges. Intuitively, generation corresponds to sampling the complete graph, while sampling usually chooses elements from an existing graph or data set [40]. Still, a sampling process can be effectively simulated using the random generation process as a component. These two models can be easily adapted to model directed and undirected graphs with and without self loops, as well as bi- and multipartite graphs.

Despite the omnipresence of random graphs as a data representation tool, previous research has not paid due attention to the question of efficiency in random graph generation. A naïve implementation of the basic Erdős-Rényi graph generation process, as given in the $\Gamma_{v,p}$ model, does not scale well to very large graphs. In this paper we propose *PPreZER*, a novel, parallel algorithm for random graph generation under the $\Gamma_{v,p}$ model[2]. For the sake of presentation, we outline our full algorithm via a progression of seven algorithmic proposals, bringing in one additional feature at each step. Each intermediate step in this progression forms a random graph generation algorithm in its own right, and the effect of each additional feature can be evaluated via a comparison of the algorithm that includes it to its predecessors. All our algorithms can be tuned to a specific type of graph (directed, undirected, with or without self-loops, multipartite) by an orthogonal decoding function.

We first propose a succession of four sequential algorithms. Our first algorithm, *ER*, is the naïve implementation of the random process of Gilbert's model. In our second algorithm, *ZER*, we exploit the availability of an analytical formula for the expected number of edges that can be skipped, in a geometric approach. Next, we improve this algorithm further to *PreLogZER*, which avoids an expensive computation of logarithms via pre-computation. Still, the skipping element in the algorithm can be implemented even more efficiently, using an acceptance/rejection [35] or Ziggurat [29] method. We add this additional feature in our *PreZER* algorithm.

Then we devise data parallel versions for the ER, ZER and PreZER algorithms. We refer to these data parallel versions as *PER*, *PZER* and *PPreZER*, respectively. We eschew developing a parallel version of PreLogZER, as the benefits this intermediary step brings are sufficiently understood in the sequential version. These three algorithms are implemented using the application programming interface CUDA from the C language, and executable on an off-the-shelve graphics card with a graphics processor unit (GPU). We do not use any specific synchronization mechanism proprietary to more sophisticated and more expensive hardware.

The remainder of this paper is structured as follows. The next section introduces the principles and properties of the Erdős-Rényi $\Gamma_{v,p}$ model along with a baseline generation algorithm. Section 3 describes our three successive enhancements over the baseline sequential algorithm, namely ZER, PreLogZER and PreZER. Section 4 presents our design of the data parallel algorithms PER, PZER and PPreZER. We conduct an extensive empirical comparison of the performance of all seven algorithms in Section 5. We conclude and discuss our plans for future work in Section 7.

[2]For the sake of simplicity, henceforward we refer to the $\Gamma_{v,p}$ model as the Erdős-Rényi model.

2. BASELINE ALGORITHM

A graph in the $\Gamma_{v,p}$ model has v vertices, while each possible edge, i.e., any edge between a pair of vertices, appears with probability p. A particularly simple random process constructs at random a graph with an edge probability distribution corresponding to this model. The process only has to iteratively consider each possible edge and select it with probability p. In case of a directed graph with self-loops, this process selects each of the v^2 possible edges with probability p. This process can be naïvely implemented by two loops over the v vertices, selecting an edge if a uniformly drawn random number between 0 and 1 is smaller than p. Algorithm 1 illustrates the pseudocode for this algorithm.

Algorithm 1: Original ER

Input: v : number of vertices, indexed 0 to $v-1$;
p : inclusion probability
Output: G : Erdős-Rényi graph

```
1 G = ∅;
2 for i = 0 to v − 1 do
3 |   for j = 0 to v − 1 do
4 |   |   Generate a uniform random number θ ∈ [0, 1);
5 |   |   if θ < p then
6 |   |   |   G ← (i, j);
```

Algorithm 1 is designed for a particular desired type of graph, i.e., a directed graph of v vertices with self-loops. Still, it is possible to implement an algorithm for the $\Gamma_{v,p}$ model without prejudice to the particular desired graph type. Given the number of possible edges E, in a certain graph, it suffices to generate indices between 0 and $E-1$. Such indices can be decoded with regard to the particular desired graph type.

For example, the edge corresponding to the index 13 for a directed graph of 5 vertices (labeled 0 to 4) with self-loops is the edge between vertex 2 and vertex 3. Algorithm 2 provides the pseudocode that produces this decoding. Similar coding and decoding functions are available for directed and undirected graphs, with and without self loops, as well as for multipartite graphs.

Algorithm 2: Decoding

Input: ind : edge index; v : number of vertices
Output: the decoded edge (i, j) where the edge starts from vertex i and ends at vertex j

```
1 i = ⌊ind / v⌋;
2 j = ind mod v;
```

The decoding aspect is orthogonal to the performance of the graph generation algorithm. Thus, any decoding can be relegated to the stage after graph generation. In the case of sampling, the decoding is rendered irrelevant, as a particular graph to be sampled is a given of the problem. In effect, without loss of generality, we omit the presentation of particular decoding algorithms. Such algorithms are *the same* for all sequential algorithms and could be parallelized in the same way for all data parallel algorithms, yielding the same further speedup. They are inconsequential to the performance study we are concerned with here.

We now rewrite the baseline ER algorithm using a single loop from 0 to $E-1$ in Algorithm 3.

Algorithm 3: ER

Input: E : maximum number of edges;
p : inclusion probability
Output: G : Erdős-Rényi graph

```
1 G = ∅;
2 for i = 0 to E − 1 do
3 |   Generate a uniform random number θ ∈ [0, 1);
4 |   if θ < p then
5 |   |   G ← e_i;
```

3. SEQUENTIAL ALGORITHMS

In this section we successively introduce three sequential algorithms for random graph generation in the $\Gamma_{v,p}$ model, namely ZER, PreLogZER and PreZER. These algorithms exploit the geometric distribution corresponding to the Bernoulli process constituted by the ER algorithm as well as the idea of pre-computing and sampling from a distribution.

3.1 Skipping Edges

The ER algorithm employs a process of successively selecting edges. This is a Bernoulli process with probability p and sequence length E, the total possible edges in the type of graph considered. The number of selected edges has a binomial distribution $B(E, p)$, hence the mean of the number of edges selected is:

$$\mu = p \times E \tag{1}$$

and its standard deviation is:

$$\sigma = \sqrt{p \times (1-p) \times E} \tag{2}$$

At any step of the sequence, let k be the number of edges that are skipped before the next edge is selected. The value of k has a geometric distribution with parameter p. Its probability mass distribution, i.e., the probability that exactly k edges are skipped at any step of the sequence, is:

$$f(k) = (1-p)^k \times p. \tag{3}$$

The respective cumulative distribution function, expressing the probability that any number of edges from 0 to k is skipped, is:

$$F(k) = \sum_{i=0}^{k} f(i) = 1 - (1-p)^{k+1} \tag{4}$$

with mean $\frac{1-p}{p}$ and standard deviation $\frac{\sqrt{1-p}}{p}$.

Following the above analysis, as argued in [2], we can avoid the per se computation of each skipped edge during the Bernoulli process. Instead, at the beginning of the process, and at any point at which an edge has been selected, we can randomly generate the number of skipped edges k and hence directly select the next $(k+1)^{th}$ edge. In order to generate the value of k, we reason as follows. Let α be a number chosen uniformly at random in $(0, 1]$. Then, the probability that α falls in interval $(F(k-1), F(k)]$ is exactly $F(k) - F(k-1) = f(k)$. In other words, the probability that k is the smallest positive integer such that $\alpha \leq F(k)$ is $f(k)$. Then, in order to assure that each possible value of k is generated with probability $f(k)$, it suffices to generate α and then calculate k as the *smallest* positive integer such that $F(k) \geq \alpha$, hence $F(k-1) < \alpha \leq F(k)$ (or zero, if no such positive integer exists). Setting $\varphi = 1 - \alpha$, this condition is equivalently written as:

$$1 - (1-p)^k < \quad \alpha \quad \leq 1 - (1-p)^{k+1} \Leftrightarrow \tag{5}$$

$$(1-p)^{k+1} \leq \quad \varphi \quad < (1-p)^k \tag{6}$$

where φ is chosen uniformly at random in $[0, 1)$. By Equation 6, k is computed as

$$k = \max(0, \lceil \log_{1-p} \varphi \rceil - 1) \tag{7}$$

Then the expected value for k is:

$$\mathcal{E}(k) = \lim_{l \to +\infty} \sum_{i=0}^{l} i \times (1-p)^i \times p = \frac{1-p}{p} \tag{8}$$

As we skip k edges, the offset of the next selected edge is $k+1$, with expected value $\mathcal{E}(k+1) = \frac{1-p}{p} + 1 = \frac{1}{p}$. If n skips are required to process all E vertices, then $\sum_{i=1}^{n} \mathcal{E}(k+1) = n \times \frac{1}{p} = E$. Thus, the expected number of skips required to process all E vertices is $\mathcal{E}(n) = p \times E$.

3.2 ZER

According to the preceding discussion, we can eschew the drawing of a random number for each single edge; it suffices to compute the offsets of edge-skipping steps instead. An algorithm that implements this idea is given in [2]; the edge-skipping process is reminiscent of a similar process in the Z Reservoir sampling algorithm [41], hence we name this algorithm ZER; its pseudocode is given in Algorithm 4.

Algorithm 4: ZER

Input: E : maximum number of edges;
p : inclusion probability
Output: G : Erdős-Rényi graph

```
1 G = ∅;
2 i = −1;
3 while i < E do
4 |   Generate a uniform random number φ ∈ [0, 1);
5 |   Compute the skip value k = max(0, ⌈log_{1−p} φ⌉ − 1);
6 |   i = i + k + 1;
7 |   G ← e_i;
8 Discard the last edge;
```

As argued in [2], a random sampling algorithm that exploits a skipping process is expected to be faster than the algorithm that explicitly considers each candidate sample; accordingly, ZER is expected to be faster than ER, as it considers only a fraction of the total number of edges. However, as we show in Section 5.2.1, this turns out not to be always the case, due to the logarithm computation overhead (Step 5 of ZER).

3.3 PreLogZER

We now turn our attention to this logarithm computation overhead that prevents ZER from achieving its full potential.

If we generate a 16-bit random number $\varphi \in (0, 1]$, then φ can assume $2^{16} = 65536$ different values. This is likely to be a small number relative to the number of times the logarithm function (Step 5 of ZER) is called. For instance, assume we generate a directed graph of $v = 10,000$ vertices, hence $E = 100,000,000$ candidate edges, under probability $p = 0.1$. Then, we expect to calculate $\mathcal{E}(n) = p \times E = 10,000,000$ logarithm calls.

In effect, it appears to be more economical to pre-calculate and store the logarithm values of all possible 65536 random numbers. Such preprocessing can be especially useful in the generation of multiple random graphs, and it is likely to bring a benefit in the generation of a single random graph alone. We call the algorithm that makes use of this preprocessing step PreLogZER; its pseudocode is presented in Algorithm 5.

Algorithm 5: PreLogZER

Input: E : maximum number of edges;
p : inclusion probability
Output: G : Erdős-Rényi graph

```
1  G = ∅;
2  c = log(1 − p);
3  for i = 1 to RAND_MAX do
4  |  log[i] = log(i/RAND_MAX);
5  i = −1;
6  while i < E do
7  |  Generate a uniform random number
   |  φ ∈ [0, RAND_MAX);
8  |  Compute the skip value k = max(0, ⌈log[φ]/c⌉ − 1);
9  |  i = i + k + 1;
10 |  G ← e_i;
11 Discard the last edge;
```

We have based our argument on the benefit brought about by PreLogZER on the assumption that we use 16-bit random numbers. Still, this benefit does not hold any more if we use random numbers of higher precision. Indeed, in the experiments of Section 5 we use 32-bit random numbers, which render the PreLogZER algorithm competitive only for very large graphs and relatively high probability values p, for which the expected number of logarithm calls exceeds 2^{32}. Still, we wish our generation algorithm to be of general utility for reasonably-sized graphs and high random-number precision.

3.4 PreZER

Our first effort at avoiding the logarithm computation overhead was based on pre-computing all logarithm values we may need, so that we shall never need to compute one more than once. Still, the cost reduction would be much more effective if we avoided logarithm computation altogether. Indeed, instead of pre-computing the logarithm values themselves, it suffices to pre-compute the breakpoints of the cumulative distribution function $F(k)$, as follows.

According to Equation 3, the probability $f(k)$ that k edges are skipped decreases as a function of k. Figure 1 illustrates the $f(k)$ function for several values of p. In effect, the value of k is likely to be lower than some, sufficiently large, fixed integer m. Then, instead of computing the value of k as a function of $\varphi = 1 - \alpha$ at each iteration, we can simply pre-compute the $m+1$ breakpoints of the intervals in which random number α is most likely to fall, from which the value of k is directly determined.

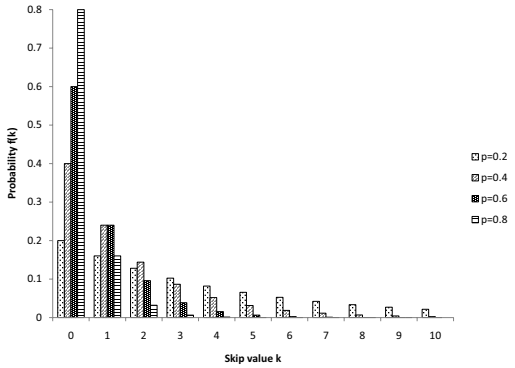


Figure 1: f(k) for varying probabilities p.

It then suffices to generate α uniformly at random in the interval $[0, 1)$ and compare it with $F(k)$ for $k = 0$ to m. We can then set the value of k to the smallest value such that $F(k) > \alpha$, or otherwise, if $F(m) \leq \alpha$, compute k by invoking an explicit logarithm computation as in ZER; such computations should be invoked rarely for sufficiently large values of m. The exact value of m is to be decided based on the requirement of the application at hand. Algorithm 6 gives the pseudocode for this PreZER algorithm.

Algorithm 6: PreZER

Input: E : maximum number of edges;
p : inclusion probability
Output: G : Erdős-Rényi graph

```
1  G = ∅;
2  for i = 0 to m do
3  |  Compute the cumulative probability F[i];
4  i = −1;
5  loop:
6  while i < E do
7  |  Generate a uniform random number α ∈ (0, 1];
8  |  j = 0;
9  |  while j ≤ m do
10 |  |  if F[j] > α then
11 |  |  |  Set the skip value k = j; Break;
12 |  |  j = j + 1;
13 |  if j = m + 1 then
14 |  |  Compute the skip value k = ⌈log_{1−p}(1−α)⌉ − 1;
15 |  i = i + k + 1;
16 |  G ← e_i;
17 Discard the last edge;
```

4. PARALLEL ALGORITHMS

In this section we leverage the parallel-processing capabilities of a Graphics Processing Unit to develop three successive data parallel algorithms for random graph generation in the $\Gamma_{v,p}$ model. These algorithms are the data parallel counterparts of ER, ZER and PreZER. For the economy of the presentation, we leave PreLogZER out of the discussion, as the benefits it confers are overshadowed by those of PreZER.

4.1 GPU Essentials

Graphics Processing Units (GPUs) are intended to be used for tasks related to graphics applications, as their name suggests. However, their performance, general availability, and

cost render them excellent candidates for hosting general-purpose algorithms as well. We exploit this potential, aiming to provide fast random graph generation algorithms that can be easily deployed on hardware that is likely to be available in most application scenarios.

4.1.1 GPU Architecture

Modern GPUs are multiprocessors with multi-threading support. Currently, standard GPUs do not offer efficient synchronization mechanisms. However, their single-instruction, multiple-threads architecture can leverage thread-level parallelism. The chief function of a multiprocessor GPU is to execute hundreds of threads running the same function concurrently on different data. Data parallelism is achieved by hardware multi-threading that maximizes the utilization of the available functional units.

As our implementation platform we use nVidia graphic cards, which consist of an array of Streaming Multiprocessors (SMs). Each SM can support a limited number of co-resident concurrent threads, which share the SM's limited memory resources. Furthermore, each SM consists of multiple (usually eight) Scalar Processor (SP) cores. The SM performs all thread management (i.e., creation, scheduling, and barrier synchronization) entirely in hardware with zero overhead for a group of 32 threads called *warp*. The zero overhead of lightweight thread scheduling, combined with fast barrier synchronization, allow the GPU to efficiently support fine-grained parallelism.

4.1.2 Programming the GPU

In order to program the GPU, we use the C-language Compute Unified Device Architecture (CUDA) [1] parallel-computing application programming interface. CUDA is provided by nVidia and works on nVidia graphic cards. The CUDA programming model consists of a sequential host code combined with a parallel kernel code. The former prepares the input data, orchestrates the invocation of the parallel code, manages the data transfer between main memory and GPU memory, and collects and processes the results; the latter is the actual code executed on the GPU.

4.1.3 Parallel Pseudo-Random Number Generator

In order to fully transpose our algorithm to parallel versions, we should generate random numbers on the GPU. Unfortunately, there is no programmatically accessible source of entropy that can be used to generate true random numbers on a GPU. There is no readily available GPU-based pseudo-random number generator either. While solutions as the one in [39] are available for older-generation GPUs, we wish our pseudo-random number generator to be fast, exploit the features of state-of-the-art programmable GPUs, and be able to generate independent uniformly distributed random numbers in every thread with a single seed. Most existing pseudo-random number generators designed for CPUs are difficult to adapt to a GPU in a way that satisfies the above requirements [17].

To satisfy our requirements, we use Langdon's pseudo-random number generator [24, 25], a data-parallel version of Park-Miller's pseudo-random number generator [33], also used for random sampling in the context of genetic programming [26]. Park-Miller's algorithm is a linear congruential generator (LCG) [23, 4] with increment equal to zero and transition function of the form $x_{n+1} = (a \times x_n) \mod m$ with $a = 16807$ and $m = 2147483647$. Park and Miller show that these settings generate a full period in the range from 1 to $m - 1$, with equal probability for each number in this range [33]. For a correct implementation, multiplications in Park-Miller's random number generator must be on 46 bits. In [25], Langdon compared his GPU version of Park-Miller's pseudo-random number generator with the original implementation, and validated both using the method suggested by Park and Miller [33].

Algorithm 7: PLCG

Input: $MasterSeed$, $ItemsPerThread$
Output: $PRNG$: list of random numbers of one thread

```
1 a = 16807;
2 m = 2147483647;
3 reciprocalm = 1.0/m;
4 seed = MasterSeed + ThreadNumber;
5 for i = 0 to ItemsPerThread do
6 |   temp = seed × a;
7 |   φ = (temp − m × floor(temp × reciprocalm));
8 |   seed = φ;
9 |   PRNG ← φ;
```

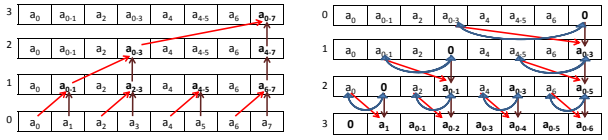
Algorithm 7 shows the pseudocode for our implementation of Langdon's pseudo-random generator. For our purposes, we produce a block of B random numbers. Each streaming multiprocessor concurrently generates a vector of random numbers of size $\frac{B}{|threads|}$. Each thread is seeded differently. Then, the vectors are concatenated to produce the final block of random numbers. It suffices to initialize the pseudo-random generator with different seed on each stream [24], using a single master seed from which individual thread seeds are derived. A simple way to create an individual seed for each thread is to add the thread number to the master seed. Langdon shows that, despite the dependence of neighboring thread numbers, for most practical purposes, it suffices to discard the first 3 random numbers to obtain independent random numbers thereafter [24]. We adopt this method.

4.1.4 Parallel Prefix Sum

Another fundamental primitive we need for parallelizing our algorithms is an algorithm that creates a sequence of partial sums from an existing sequence of numbers. Such an algorithm has been proposed by Iverson [20], known as *Prefix Sum* or *Scan with Sum Operation*. Prefix sum comes in two variants: *inclusive* and *exclusive*. Given an input sequence a_i, *inclusive* prefix sum generates the sequence $b_i = a_0 + a_1 + \cdots + a_i$, while *exclusive* prefix sum generates the sequence $b_i = a_0 + a_1 + \cdots + a_{i-1}$.

A GPU-based implementation of prefix sum is given by Horn [16] and optimized by Hensley et al. [15]. Both these implementations have $O(nlogn)$ complexity, while a sequential scan requires only $O(n)$. Sengupta et al. [38] propose a $O(n)$ GPU-based data-parallel prefix sum algorithm, which is further tuned for CUDA [37, 36]. This algorithm consists of two phases: the up-sweep (also known as reduce) and the down-sweep phase, illustrated in Figure 2. In Figure 2a, sequence 0 is the original sequence. The elements of the subsequent sequences are computed by adding values of elements of the preceding one, as shown in the figure. For instance, a_{0-1} in sequence 1 is the sum of a_0 and a_1 in se-

quence 0, as indicated by the arrows. In Figure 2b, sequence 0 (on top) is the result of the upsweep operation. Straight arrows indicate sum, as previously, while curly arrows indicate a swap of elements. The eventual result (bottom of Figure 2b) is the desired prefix sum sequence.



(a) Up-sweep (b) Down-sweep

Figure 2: Phases of Parallel Prefix Sum [36].

On a GPU, this algorithm suffers from multiple threads accessing the same bank. It requires artificial padding to prevent such conflicts. Sengupta et al. propose an approach based on the hierarchy model of threads in CUDA that avoid padding [36]. We use Sengupta et al.'s implementation of both inclusive and exclusive prefix sums.

4.1.5 *Parallel Stream Compaction*

Last, in our parallelization scheme we employ stream compaction, a method that compresses an input array A into a smaller array B by keeping only the elements that verify a predicate p, while preserving the original relative order of the elements [16].

Stream compaction consists of two phases. A *scan phase* computes an exclusive parallel prefix sum on the given predicate list (see Section 4.1.4). In effect, the result of this scan can provide the addresses of those edges that passes the predicate, as those locations in the parallel prefix sum list that differ from their successor. Then, a *scatter phase* organizes the edge indices into the final edge in a parallel-processing manner, using the list generated by the scan phase [16]. The scatter operation is natively accessible in recent GPUs, rendering stream compaction considerably more efficient. Figure 3 illustrates its working for an array of 10 elements.

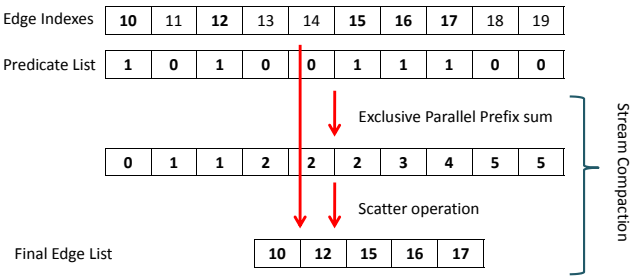


Figure 3: Stream Compaction for 10 elements.

4.2 PER

The ER algorithm naturally lends itself to parallelization. A straightforward way for rendering it parallel is to spread predefined portions of a complete graph's edge set E among the available processing units. The thread running in each unit checks independently each edge in its assigned group against a uniform random number, independently generated as in Section 4.1.3, and sets a corresponding edge predicate to 1 if the condition is satisfied, otherwise to 0.

The resulting predicate list can be used to represent the generated graph. However, such a representation would be inefficient in most practical cases, where the probability p is relatively low and, therefore, the generated graph relatively sparse, resulting into a predicate list of many 0 entries. All such 0 entries would have to be kept in the processing units' memory and later transferred to the master CPU. We devise a more efficient method in our PER algorithm, the data parallel version of ER, as follows.

Instead of explicitly maintaining the unselected edges, we discard edges with predicate value 0 from the edge list. We perform this discarding operation in a parallel manner. We first compute the total number of selected edges, in parallel, via an exclusive parallel prefix sum on the predicate list (see Section 4.1.4). Then, we proceed to extract the location of each selected edge via the scatter phase of the parallel stream compaction operation, using the result of the parallel prefix sum, as in Section 4.1.5. The pseudocode for the overall PER algorithm is given in Algorithm 8.

Algorithm 8: PER

Input: E : maximum number of edges;
p : inclusion probability
Output: G : Erdős-Rényi graph

```
1  G = ∅;
2  B = |threads|;
3  Iterations = E/B;
4  for i = 0 to Iterations − 1 do
5  |  T = ∅;
6  |  R = ∅;
7  |  In parallel : Generate B random numbers θ ∈ [0, 1)
   |  in an array R;
8  |  PL = ∅;
9  |  In parallel : foreach θ ∈ R do
10 |  T ← θ's position in R + i × B;
11 |  if θ < p then
12 |  |  PL ← 1;
13 |  else
14 |  |  PL ← 0;
15 |  In parallel : Perform Parallel Stream Compaction
   |  on T wrt PL;
16 |  G ← T;
```

The main parameter of the PER algorithm is the number of threads per block from which we derive the number of edges per thread in order to process the total number of edges. Figure 3 illustrates, in effect, the working of PER for a set of ten edges.

4.3 PZER

We proceed to develop a data parallel version of ZER, which we call PZER. The two building blocks of PZER are (i) a data parallel pseudo-random number generator for the computation of the skip values, and (ii) an inclusive parallel prefix sum for the computation of absolute edge indices. PZER iteratively computes blocks of edges until the maximum index is reached.

We employ the pseudo-random generator presented in Section 4.1.3. Again, the generated random numbers are used to compute the skip value as for the ZER algorithm (Equation 7). These values can indeed be calculated independently and concurrently. However, in order to correctly derive the

absolute edge indices from these skip values, each processing unit needs to know the location of its starting edge. This starting edge location can only be disclosed, for each processing unit, once all edges preceding the group of edges handled by that unit have been processed.

In order to address this problem, we separate skip value generation from edge index computation. In effect, each processing unit first generates sufficient skip values. Then, once all units have stored their generated skip values, we compute the overall absolute index values via a parallel prefix sum operation (see Section 4.1.4) over the full skip value list. The pseudocode for this PZER algorithm is given in Algorithm 9.

Algorithm 9: PZER

Input: E : maximum number of edges;
p : inclusion probability;
λ : parameter of the block size
Output: G : Erdős-Rényi graph

$G = \emptyset$;
$L = 0$;
$\sigma = \sqrt{p \times (1-p) \times E}$;
$B = (p \times E) + (\lambda \times \sigma)$;
while $L < E$ **do**
  $R = \emptyset$;
  In parallel : Generate B random numbers $\varphi \in [0, 1)$ in an array R;
  $S = \emptyset$;
  In parallel : **foreach** $\varphi \in R$ **do**
    Compute the skip value $k = \max(0, \lceil \log_{1-p} \varphi \rceil - 1)$;
    $S \leftarrow k$;
  In parallel : Perform Parallel Prefix Sum on S;
  $G = G \cup S$;
  $L =$ Last Edge Index in G;

Figure 4 illustrates how PZER generates the absolute edge index list with probability $p = 0.1$ and block size of 10.

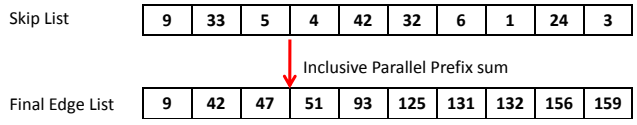


Figure 4: Generating edge list via skip list in PZER.

The main parameter of the PZER algorithm is the block size it works with. If that size is too large, then too many threads will perform unnecessary computations. If the same is too small, then there will be too many iteration steps of the main while loop of the algorithm, attenuating its parallel character. Thus, a tradeoff between unnecessary computations and potential loss of parallelism emerges.

To strike a balance in this tradeoff, we biased the algorithm slightly towards the side of excessive computations, using a block size that is a number of λ standard deviations larger than the mean number of edges as given in Equations 1 and 2. Thus, we use block size $B = (E \times p) + (\lambda \times \sqrt{p(1-p) \times E})$, where E is the maximum number of edges in the graph, and λ the number of standard deviations added to the mean value of the number of skips.

4.4 PPreZER

Eventually, we devise a data parallel version of PreZER, which we call PPreZER. This algorithm forms the culmination of our effort. PPreZER shares the structure of PZER, but includes a pre-computation of m probability values, which are then used to guide the sampling from the logarithmic distribution, as in PreZER. The pseudocode for PZER is given in Algorithm 10.

Algorithm 10: PPreZER

Input: E : maximum number of edges;
p : inclusion probability;
m : number of pre-computations;
λ : parameter of the block size
Output: G : Erdős-Rényi graph

$G = \emptyset$;
for $i = 0$ *to* m **do**
  Compute the cumulative probability $F(i)$;
$L = 0$;
$\sigma = \sqrt{p \times (1-p) \times E}$;
$B = (p \times E) + (\lambda \times \sigma)$;
while $L < E$ **do**
  $R = \emptyset$;
  In parallel : Generate B random numbers $\alpha \in (0, 1]$ in an array R;
  $S = \emptyset$;
  In parallel : **foreach** $\alpha \in R$ **do**
    $j = 0$;
    while $j \leq m$ **do**
      if $F[j] > \alpha$ **then**
        Set the skip value $k = j$;
        Break
      $j = j + 1$;
    if $j = m + 1$ **then**
      Compute the skip value $k = \lceil \log_{1-p}(1-\alpha) \rceil - 1$;
    $S \leftarrow k$;
  In parallel : Perform Parallel Prefix Sum on S;
  $G = G \cup S$;
  $L =$ Last Edge Index in G;

The main parameters of PPreZER are the block size B, which is set, as in PZER, and m, the largest value of skip whose probability is pre-computed.

5. PERFORMANCE EVALUATION

In this section we evaluate the performance of all the algorithms we have presented and introduced to each other.

5.1 Setup

We ran the sequential algorithms on a 2.33GHz Core 2 Duo CPU machine with 4GB of main memory under Windows XP. The parallel algorithms ran on the same machine with a GeForce 9800 GT graphics card having 1024MB of global memory, 14 streaming processors (i.e., 112 scalar processors) and a PCI Express ×16 bus. All algorithms were implemented in Visual C++ 9.0.

In the default settings, we set the algorithms to generate directed random graphs with self loops, having 10,000 vertices, hence at most 100,000,000 edges. We measure execution time as user time, averaging the results over ten runs. The m parameter for PreZER and PPreZER is set to 9, a

value which is empirically determined to yield the best performance, as discussed in subsection 5.2.6. The block size for PZER and PPreZER is set to $(p \times E) + (\lambda \times \sqrt{p(1-p) \times E})$, with $E = 10^8$ and $\lambda = 3$, which is found to be the value that yields best performance, as discussed in subsection 5.2.6.

5.2 Results

5.2.1 *Overall Comparison*

We first turn our attention to the execution time of the six main compared algorithms, namely ER, ZER, PreZER, PER, PZER and PPreZER, as a function of inclusion probability p. Figure 5 shows the results, with the best parameter settings in those algorithms where they are applicable.

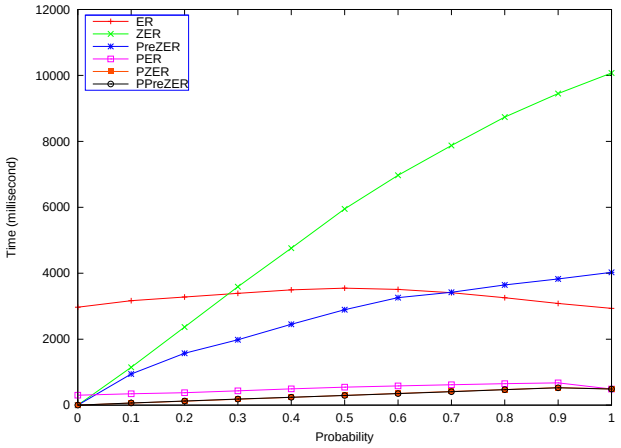


Figure 5: Running times for all algorithms.

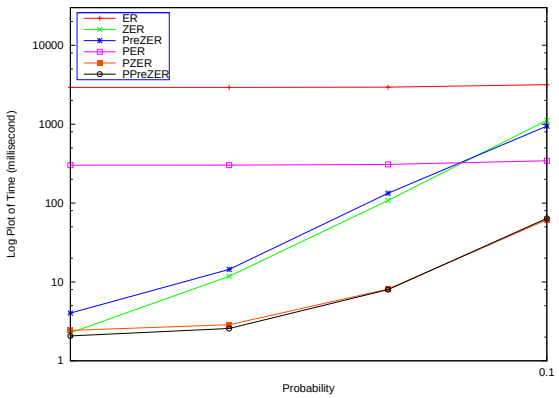


Figure 6: Running times for small probability.

We observe that both ZER and PreZER become faster as the value of p decreases. This is expected, given that smaller values of p offer more opportunities to skip edges. Still, these two relatively naive edge-skipping algorithms are *not* consistently faster than ER. ZER remains faster than ER only for probability values up to $p = 0.3$; this poor performance is explained by the overhead due to logarithm computations. PreZER is more efficient than ER for probability values up to $p = 0.7$; after that point, the advantage of having precomputed values does not suffice to achieve better performance.

All three parallel algorithms are significantly faster than the sequential ones for all values of p; the only exception to this observation is that PER is slightly slower than ZER and PreZER for very small values of p, as it has to generate E random numbers whatever the value of p. Figure 6 illustrates the effect of such small values of p on execution time on logarithmic axes.

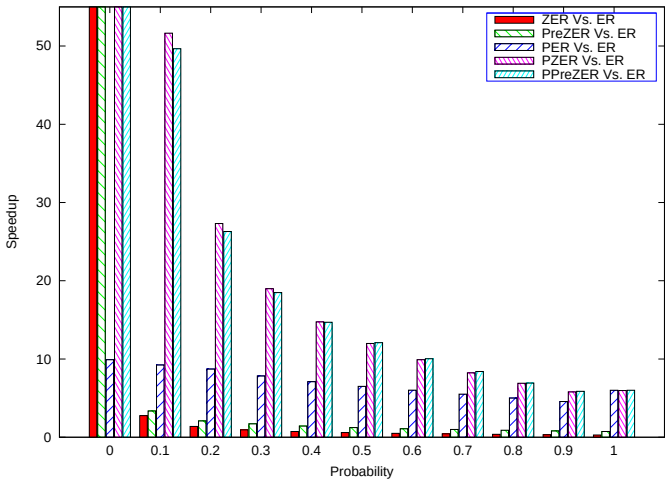


Figure 7: Speedup for all algorithms over ER.

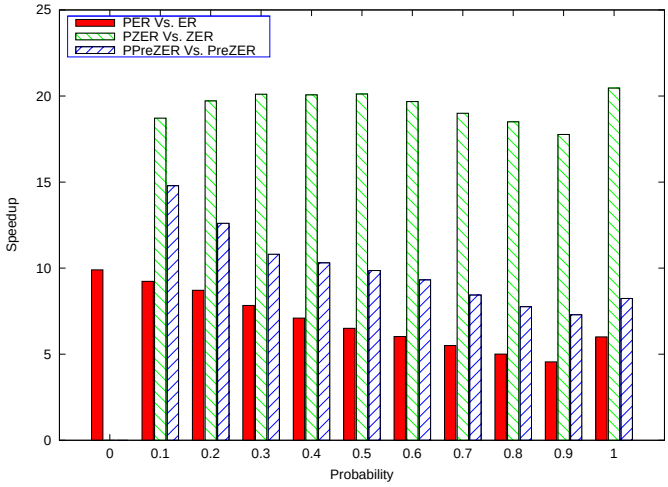


Figure 8: Speedup for parallel algorithms over their sequential counterparts.

5.2.2 *Speedup Assessment*

Next, Figure 7 presents the average speedup over the baseline ER algorithm, for the other five algorithms in the comparison, as a function of p. The average speedup for PreZER, PER, PZER and PPreZER are 1, 1.5, 7.2, 19.3, 19.2, respectively. Besides, for $p \leq 0.5$, the average speedup for ZER, PreZER, PER, PZER and PPreZER are 1.3, 2, 8.4, 29.9 and 29.4, respectively.

In addition, in Figure 8 we gather the average speedup of each parallel algorithm over its sequential counterparts. The average speedup for PER, PZER and PPreZER over their

sequential version are 7.2, 22.8 and 11.7, respectively. For $p \leq 0.5$ the average speedup for PER, PZER and PPreZER are 8.4, 23.4 and 13.7, respectively.

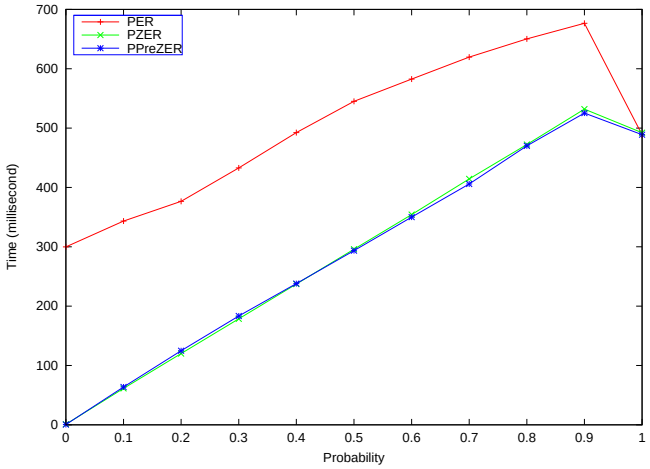


Figure 9: Running times for parallel algorithms.

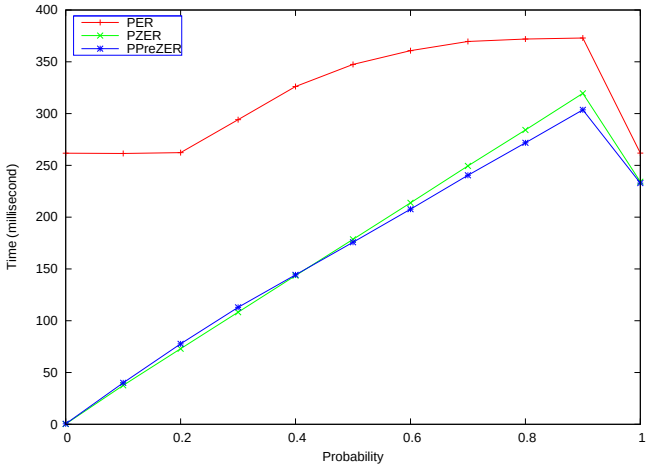


Figure 10: The times for pseudo-random number generator with skip for PZER and PPreZER and check for PER.

5.2.3 *Comparison among Parallel Algorithms*

Now we focus our attention on the comparison among the three parallel algorithms in our study. Figure 9 shows the overall execution times for these three algorithms only. For all probability values, PZER and PPreZER are faster than PER. PPreZER is slightly faster than PZER for probabilities greater than 0.4 and slightly slower or identical for the rest. This result arises from the handling of branching conditions by the GPU, as concurrent threads taking different execution paths are serialized. In order to verify this supposition, we compared only the time spent for random number generation by the three algorithms. Figure 10 shows the results, which verify the same performance pattern. We conclude that PZER and PPreZER are the fastest algorithms for random graph generation in our study.

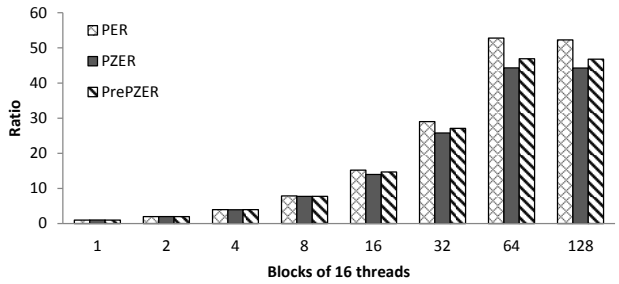


Figure 11: Speedup of the parallel algorithms against themselves for $\Gamma_{v=10K,p=0.1}$ graphs

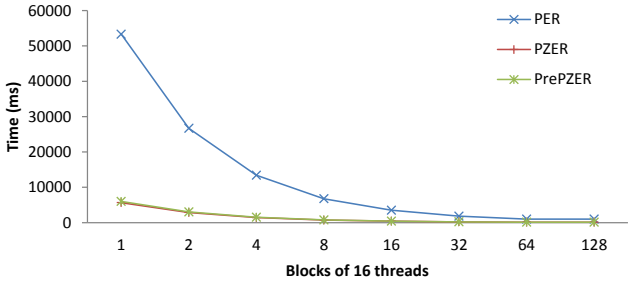


Figure 12: Runtime of the parallel algorithms for varying thread-blocks for $\Gamma_{v=10K,p=0.1}$ graphs

5.2.4 *Parallelism Speedup*

Now we evaluate the performance speedup gained solely by virtue of parallelism. For that purpose, we assess, for each parallel algorithm, its performance speedup while varying the number of thread-blocks it employs, while fixing the number of threads per block to half warp, i.e. 16. In this set of experiments, we generate 100 $\Gamma_{v=10K,p=0.1}$ graphs and we report the average time. Figures 11 and 12 depict two different aspect of our results.

Figure 11 presents the speedup of each parallel algorithm against the version of itself running with one thread-block. The runtime charts, in Figure 12, show the total amount of time spent for each parallel algorithm. The results show that the greater the number of thread-blocks is, the faster the algorithm. This result holds for up to 64 thread-blocks, as the hardware configuration does not afford more benefits from parallelism under the specific v and p values in this experiment.

5.2.5 *Size Scalability*

In all hitherto experiments, we have used a constant graph size (i.e., number of vertices) and tuned probability p or the number of thread-blocks we employ. Still, the question of scalability of our algorithms to larger graph sizes also needs to be addressed. Figures 13, 14, and 15, show the execution time results for three different probability values, as a function of increasing number of vertices.

The results reconfirm our previous findings and carry them forward to larger graph structures in terms of vertices. They verify that the difference between ZER and PreZER is at-

tenuated for smaller values of p, while the advantages of skip-based parallel algorithm are amplified for such smaller probability values.

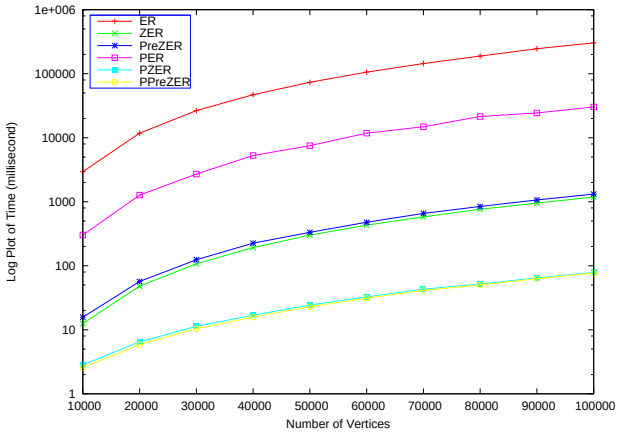


Figure 13: Runtime for varying graph size, $p = 0.001$

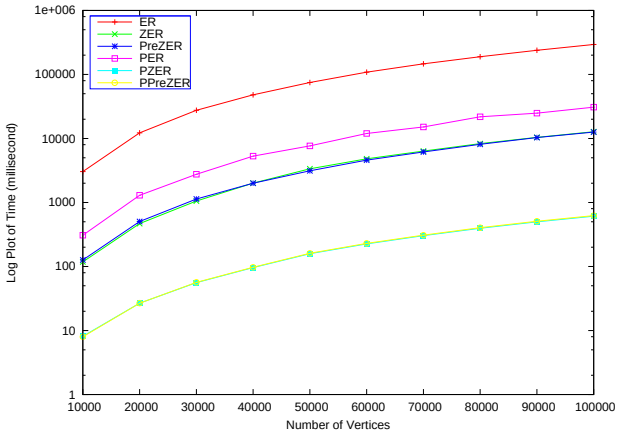


Figure 14: Runtime for varying graph size, $p = 0.01$

5.2.6 *Performance Tuning*

We also conducted an experimental study to aid our tuning decisions. We shortly report our findings here.

CUDA's threads are organized in a two-level hierarchy: a batch of threads forms a thread-block and a collection of thread-blocks makes a grid. A warp is a group of 32 threads. A thread-block must have at least four warps, i.e. 128 threads, to fully utilize the hardware resources and hide the latencies [1]. Our experiments suggest that the optimal thread-block configuration strongly depends on the particular requirements of the running algorithm. In Section 5.2.4 we have seen an instance where a given thread-block configuration did not confer more benefits from parallelism after crossing a threshold.

We have experimented with different values of the m parameter, the largest value of skip whose probability is precomputed in PreZER and PPreZER. Our experiments have shown that there are only marginal benefits to be gained for m values larger than 9, due to the tradeoff discussed in Section 4.3. We used the value $m = 9$ in our experiments.

Furthermore, we have experimented with different values of the size-of-block parameter λ in the PZER and PPreZER

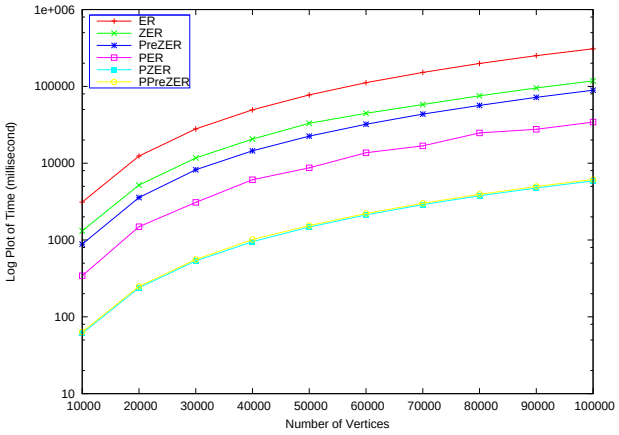


Figure 15: Runtime for varying graph size, $p = 0.1$

implementation. Our experiments have shown that performance improves with increasing λ up to $\lambda = 3$, with not much to gain thereafter, due to the over-computation overhead (see Section 4.3). Thus, we have used the value $\lambda = 3$ in our experiments.

5.3 Discussion

A possible objection to the significance of our results is the fact that the speedup achieved is gained over an algorithm which is already relatively fast compared to time needed to store the results in secondary memory. Indeed the time to write individual or buffered results to disk dominates the generation. However, this is not the case when the graph remains in main memory. Besides, the relevance and applicability of our results is enhanced by the increasing availability of flash memory storage and the proliferation of graph-based techniques, methods, and simulation requirements in modern data management tasks.

Our algorithms are efficient for generating graphs that fit in main memory for further processing but also for larger graphs to be further processed concurrently, in parallel, or in a streaming fashion. This is even more the case when the graphs remain in the memory of the GPU for further processing. Still, for the sake of a fair comparison, our experiments have been conducted with all results written to main memory.

6. RELATED WORK

Graphs constitute such a versatile data model that they are used in almost every domain of science and humanities.

Random graphs [3] are used for the generation of synthetic data sets for the evaluation of effectiveness and efficiency of graph processing methods. They are also used in simulation [21], randomized algorithms [13], and sampling [40]. In the domain of information and knowledge management, they are used in such applications as web mining and analysis [22, 5], data mining [18], and social network analysis [32], among a multitude of other examples.

It has been properly argued that Erdős-Rényi random graphs are generally not representative of real-life graphs, such as communication networks, social networks, or molecular structures, which exhibit specific degree distributions and resulting properties. However, if real-life graphs significantly differ from Erdős-Rényi graphs, one still needs Erdős-

Rényi models to characterize these differences and quantify their significance. The generation of random graphs with prescribed constraints generally requires different and more complex machinery. The prevalent approach is the usage of mixing Markov chains, as found in [12], for the generation of random structures as bipartite graphs of transactions and items with prescribed degree constraints.

The authors of [42] distinguish between matching-based models and switching-based models. Mixing-Markov-chains-based generation leverages switching-based models. The two models referred to as Erdős-Rényi models were proposed in 1959 by Paul Erdős and Alfréd Rényi, for $\Gamma_{v,e}$ in [8], and E.N. Gilbert in [11], for $\Gamma_{v,p}$, respectively. They are matching-based models. The two models indeed coincide when v, the number of vertices of the total graph, becomes sufficiently large and $e \approx E \times p$, where E is the maximum number of possible edges.

The Erdős-Rényi models can be used to sample existing graphs or other data structures. To this extent, Reservoir sampling [41] can be seen as a continuous sampling algorithm in the $\Gamma_{v,e}$ model, where e is the size of the Reservoir. The authors of [40] propose a sampling algorithm for counting triangles in graph based on the $\Gamma_{v,p}$ model. They discuss a MapReduce [7] implementation of their algorithm that shares the conceptual design of our PER data parallel algorithm: generate and test vertices in parallel and compact the result.

As we have discussed, the said model induces a simple generation algorithm. To our knowledge, the authors of [2] were the first to discern that this algorithm can be improved by leveraging the geometric distribution of the number of elements skipped. This idea is similar to the one used in Reservoir algorithm Z of [41] for reservoir sampling of streams. We refer to this algorithm as ZER. Still, unfortunately, the theoretical improvement announced in [2] cannot be achieved in practice because of the overhead of the logarithm computation.

We are not aware of any other attempt to enhance the efficiency of random graph generation. In our effort, the availability of a GPU gave us the motivation and the opportunity to successfully tackle this task. The absence of readily available random generators on graphic processors [24] and the data structures of the processor and of the CUDA framework [1] posed additional challenges that we were able to address.

7. CONCLUSIONS

7.1 Summary of Contribution

This paper has led to the proposal of two new algorithms for the generation of random graphs in the Erdős-Rényi model $\Gamma_{v,p}$, namely PreZER and PPreZER. In order to motivate, explain, and evaluate this work, we have outlined a succession of algorithm leading to our two flagship contributions. The baseline algorithm, ER, and the three enhancements thereupon, namely ZER, PreLogZER and PreZER, are sequential algorithms. The three algorithms PER, PZER and PPreZER are data parallel versions of their sequential counterparts designed for graphics cards and implemented in CUDA. To our knowledge, PreZER is the fastest known sequential algorithm, while PZER and PPreZER can both claim the title of the fastest known parallel algorithms for a GPU. They yield average speedups of 1.5 and 19 over the baseline algorithm, respectively. These results enable significant efficiency gains in modern data management tasks, simulations, and applications, whenever they call for the generation of a random graph.

7.2 Future Work

Currently, we are studying ways to build efficient stream sampling algorithms using the techniques developed for PZER and PPreZER. Such an alteration involves the careful design of the communication protocol between the incoming stream of data managed by the central processor and the graphic card memory, on the one hand, as well as of the streamed production of results, on the other hand. We are also studying the generation of other random structures with prescribed constraints. We study alternatives to the mixing Markov chain approach that would allow us to devise and cast new algorithms in both the random generation and random sampling frameworks at the same time. The main applications that we are contemplating are in the fields of data mining and social networks anonymization in the spirit of [12] and [14].

8. REFERENCES

[1] CUDA Zone: Toolkit & SDK. http://www.nvidia.com/object/what_is_cuda_new.html.

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

[3] B. Bollobas. *Random graphs*. Academic Press, 2nd edition, 2001.

[4] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation*, page 397. Springer, 2nd edition, 1987.

[5] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309–320, 2000.

[6] S. Crisóstomo, U. Schilcher, C. Bettstetter, and J. Barros. Analysis of probabilistic flooding: How do we choose the right coin. In *IEEE ICC*, 2009.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[8] P. Erdős and A. Rényi. On random graphs I. *Publicationes Mathematicae*, 6:290–297, 1959.

[9] J. H. Fowler, C. T. Dawes, and N. A. Christakis. Model of genetic variation in human social networks. *PNAS*, 106(6):1720–1724, 2009.

[10] A. J. Ganesh, L. Massoulié, and D. F. Towsley. The effect of network topology on the spread of epidemics. In *INFOCOM*, 2005.

[11] E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[12] A. Gionis, H. Mannila, T. Mielikäinen, and P. Tsaparas. Assessing data mining results via swap randomization. *TKDD*, 1(3), 2007.

[13] S. Hanhijärvi, G. C. Garriga, and K. Puolamäki. Randomization techniques for graphs. In *SDM*, 2009.

[14] M. Hay, G. Miklau, D. Jensen, D. F. Towsley, and P. Weis. Resisting structural re-identification in anonymized social networks. *PVLDB*, 1(1):102–114, 2008.

[15] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast summed-area table generation and

its applications. *Computer Graphics Forum*, 24(3):547–555, 2005.

[16] D. Horn. Stream reduction operations for GPGPU applications. In M. Pharr, editor, *GPU Gems 2*. Addison Wesley, 2005.

[17] L. Howes and D. Thomas. Efficient random number generation and application using CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, 2007.

[18] A. Inokuchi, T. Washio, and H. Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50(3):321–354, 2003.

[19] Y. M. Ioannides. Random graphs and social networks: An economics perspective. Technical Report 0518, Department of Economics, Tufts University, 2005.

[20] K. E. Iverson. *A programming language*. Wiley, 1962.

[21] M. Kaiser, R. Martin, P. Andras, and M. P. Young. Simulation of robustness against lesions of cortical networks. *European Journal of Neuroscience*, 25(10):3185–3192, 2007.

[22] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The web as a graph: Measurements, models, and methods. In *COCOON*, 1999.

[23] D. E. Knuth. The linear congruential method. In *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, pages 10–26. Addison Wesley, 3rd edition, 1997.

[24] W. B. Langdon. A fast high-quality pseudo-random number generator for graphics processing units. In *IEEE Congress on Evolutionary Computation*, 2008.

[25] W. B. Langdon. A fast high-quality pseudo-random number generator for nVidia CUDA. In *GECCO (Companion)*, 2009.

[26] W. B. Langdon. A many-threaded CUDA interpreter for genetic programming. In *EuroGP*, 2010.

[27] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *SIGMETRICS*, 2002.

[28] A. Ma'ayan, A. Lipshtat, R. Iyengar, and E. Sontag. Proximity of intracellular regulatory networks to monotone systems. *Systems Biology, IET*, 2(3):103–112, 2008.

[29] G. Marsaglia and W. W. Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7, 2000.

[30] D. McDonald, L. Waterbury, R. Knight, and M. Betterton. Activating and inhibiting connections in biological network dynamics. *Biology Direct*, 3(1):49, 2008.

[31] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[32] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. *PNAS*, 99(Suppl 1):2566–2572, 2002.

[33] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[34] S. Pettie and V. Ramachandran. Randomized minimum spanning tree algorithms using exponentially fewer random bits. *ACM Transactions on Algorithms*, 4(1):5:1–5:27, 2008.

[35] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer, 2005.

[36] S. Sengupta, M. Harris, and M. Garland. Efficient parallel scan algorithms for GPUs. Technical Report NVR-2008-003, NVIDIA Corporation, 2008.

[37] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 2007.

[38] S. Sengupta, A. Lefohn, and J. D. Owens. A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures (EDGE)*, 2006.

[39] M. Sussman, W. Crutchfield, and M. Papakipos. Pseudorandom number generation on the GPU. In *ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 2006.

[40] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. DOULION: counting triangles in massive graphs with a coin. In *KDD*, 2009.

[41] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[42] X. Ying and X. Wu. Graph generation with prescribed feature constraints. In *SDM*, 2009.