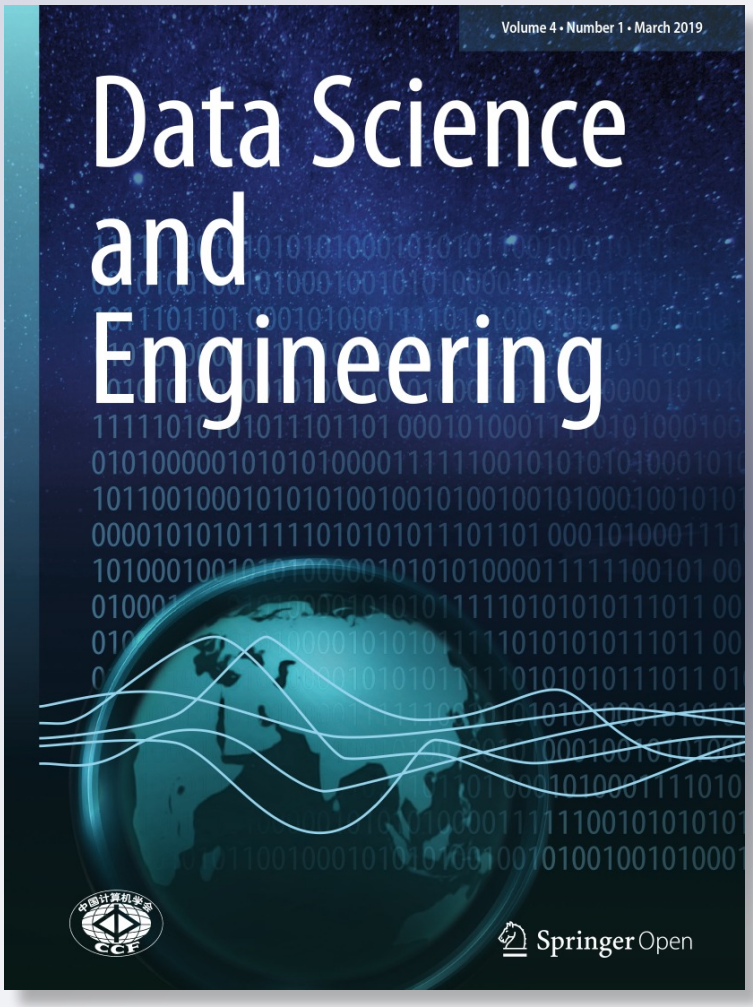
Novel Parallel Algorithms for Fast Multi-GPU-Based Generation of Massive Scale-Free Networks

Maksudul Alam, Kalyan S. Perumalla & Peter Sanders

Data Science and Engineering

ISSN 2364-1185
Volume 4
Number 1

Data Sci. Eng. (2019) 4:61-75
DOI 10.1007/s41019-019-0088-6



Springer

Data Science and Engineering (2019) 4:61–75
https://doi.org/10.1007/s41019-019-0088-6

Novel Parallel Algorithms for Fast Multi-GPU-Based Generation of Massive Scale-Free Networks

Maksudul Alam[1] · **Kalyan S. Perumalla**[1] · **Peter Sanders**[2]

Received: 21 March 2018 / Revised: 4 December 2018 / Accepted: 20 March 2019 / Published online: 30 March 2019

Abstract
A novel parallel algorithm is presented for generating random scale-free networks using the preferential attachment model. The algorithm, named **cuPPA**, is custom-designed for "single instruction multiple data (SIMD)" style of parallel processing supported by modern processors such as graphical processing units (GPUs). To the best of our knowledge, our algorithm is the first to exploit GPUs, and also the fastest implementation available today, to generate scale-free networks using the preferential attachment model. A detailed performance study is presented to understand the scalability and runtime characteristics of the **cuPPA** algorithm. Also another version of the algorithm called **cuPPA-Hash** tailored for multiple GPUs is presented. On a single GPU, the original **cuPPA** algorithm delivers the best performance, but is challenging to port to multi-GPU implementation. For multi-GPU implementation, **cuPPA-Hash** has been used as the parallel algorithm to achieve a perfect linear speedup up to 4 GPUs. In one of the best cases, when executed on an NVidia GeForce 1080 GPU, the original **cuPPA** generates a scale-free network of two billion edges in less than 3 s. On multi-GPU platforms, **cuPPA-Hash** generates a scale-free network of 16 billion edges in less than 7 s using a machine consisting of 4 NVidia Tesla P100 GPUs.

Keywords GPU · Preferential attachment · Random networks · Scale-free networks

1 Introduction

Networks are prevalent in many complex systems such as circuits, chemical compounds, protein structures, biological networks, social networks, the Web, and XML documents. Recently, there has been substantial interest in the study of a variety of random networks to serve as mathematical models of complex systems. Various network theories, metrics, topology, and mathematical models have been proposed to understand the underlying properties and relationships of these systems. Among the proposed network models, the first and the most studied model is the Erdős–Rényi model [14]. However, the Erdős–Rényi model does not exhibit the characteristics observed in many real-world complex systems [8]. Barabási and Albert [8] discovered a class of inhomogeneous networks, called scale-free networks, characterized by a power-law degree distribution $P(k) \propto k^{-\gamma}$, where k represents the degree of a vertex and γ is a constant. While high degree vertices are improbable in Erdős–Rényi networks, they do occur with statistically significant probability in scale-free networks. Furthermore, the work of Albert et al. [6] suggests these high degree vertices appear to play an important role in the behavior of scale-free systems, particularly with respect to their resilience [11]. For example, the Barabasi–Albert model can be used for evaluating the North American electric grid with high reliability [11].

As these complex systems of today grow larger, the ability to generate progressively large random networks becomes all the more important. It is well known that the structure of

✉ Kalyan S. Perumalla
perumallaks@ornl.gov

Maksudul Alam
alamm@ornl.gov

Peter Sanders
sanders@kit.edu

[1] Oak Ridge National Laboratory, Oak Ridge, TN, USA

[2] Karlsruhe Institute of Technology, Karlsruhe, Germany

larger networks is fundamentally different from that of small networks, and many patterns such as communities emerge only in massive networks [20]. Although various random network models have been used and studied over the last several decades, even efficient sequential algorithms for generating such networks were nonexistent until recently. The efficient sequential algorithms are able to generate networks with millions of edges in a reasonable amount of time; however, generating networks with billions of edges can take a prohibitively large amount of time. This motivates the need for efficient parallel algorithms for generating such networks. Naïve parallelization of the sequential algorithms for generating random networks may not work due to the dependencies among the edges and the possibility of creating duplicate (parallel) edges.

One of the earliest known parallel algorithms for the preferential attachment model is given by Yoo and Henderson [26]. Although useful, the algorithm has two weaknesses: (1) for ease of handling dependencies and avoid the required complex synchronization, they adopted an approximation algorithm rather than an exact algorithm; and (2) to correctly generate the network, the algorithm needs manual adjustment of several control parameters. An exact distributed-memory parallel algorithm was presented in [4]. A literature review of the recent developments is presented in Sect. 6.

Graphics processors (GPUs) are a cost-effective, energy-efficient, and widely available parallel processing platform. GPUs are highly parallel, multi-threaded, many-core processors that have greatly expanded beyond graphics operations and are now widely used for general purpose computing. The use of GPUs is prevalent in many areas such as scientific computation, complex simulations, big data analytics, machine learning, and data mining. However, there is a lack of GPU-based graph/network generators, especially for scale-free networks such as those based on the preferential attachment model. There exist GPU-based network generators for Erdős–Rényi networks [23] and small-world model [18]. However, until recently we found no GPU-based algorithm to generate scale-free networks [3]. In this paper, we present **cuPPA**, a novel GPU-based algorithm for generating networks conforming to the preferential attachment model. The algorithm adopts the copy model [17] and employs a simpler synchronization technique suitable for GPUs. With cuPPA, one can generate a network with two billion edges using a modern NVidia GPU in less than 3 s. To the best of our knowledge, this is the first GPU-based algorithm to generate networks using the exact preferential attachment model. Although cuPPA works really well on a single GPU, generating bigger networks with multiple GPUs is a challenging issue due to the complex synchronization and message communication required among the GPUs. We present another algorithm called **cuPPA-Hash** to generate networks using preferential attachment model on multiple GPUs. The algorithm uses hashing instead of pseudorandom number generators and does not require any communication among the GPUs. With cuPPA-Hash, we generated a network of 16 billion edges in less than 7 s using a machine consisting of 4 NVidia Tesla P100 GPUs.

The rest of the paper is organized as follows: In Sect. 2, background material is provided in terms of preliminary information, notations, an outline of the network generation problem, and two leading sequential algorithms. In Sect. 3, our parallel **cuPPA** algorithm for the GPU is presented. In Sect. 4, we present a multi-GPU algorithm called **cuPPA-Hash**. The experimental study and performance results using **cuPPA** are described in Sect. 5. We present a review of related works in Sect. 6. Finally, Sect. 7 concludes with a summary and an outline of future directions.

2 Background

2.1 Preliminaries and Notations

In the rest of this paper, we use the following notations. We denote a network $G(V, E)$, where V and E are the sets of vertices and edges, respectively, with $m = |E|$ edges and $n = |V|$ vertices labeled as $0, 1, 2, \ldots, n-1$. For any $(u, v) \in E$, we say u and v are *neighbors* of each other. The set of all neighbors of $v \in V$ is denoted by $N(v)$, i.e., $N(v) = \{u \in V | (u, v) \in E\}$. The degree of v is $d_v = |N(v)|$. If u and v are neighbors, sometimes we say that u is *connected* to v and vice versa.

We develop parallel algorithms using the CUDA (Compute Unified Device Architecture) framework on the GPU. A GPU contains multiple streaming multiprocessors (SMs). An SM is a group of core processors. Each core processor executes only one thread at a time. All core processors can execute their corresponding threads simultaneously. If some threads perform operations that have to wait for data fetches with high latencies, those are put into the waiting state and other pending threads are executed. Therefore, GPUs increase throughput by keeping the processors busy. All thread management, including the creation and scheduling of threads, is performed entirely in hardware with virtually zero overhead and requires negligible time for launching work on the GPU. For these advantages, modern supercomputers such as Summit and Titan, two of the largest supercomputers in the USA, are built using GPUs in addition to conventional central processing units (CPUs).

We use K, M, and B to denote thousand, million, and billion, respectively, e.g., 2 B stands for two billion.

2.2 Preferential Attachment-Based Models

The preferential attachment model is a model for generating randomly evolved scale-free networks using a preferential attachment mechanism. In a preferential attachment

mechanism, a new vertex is added to the network and connected to some existing vertices that are chosen preferentially based on some properties of the vertices. In the most common method, preference is given to vertices with larger degrees: The higher the degree of a vertex, the higher is the probability of choosing it. In this paper, we study only the degree-based preferential attachment, and in the rest of the paper, by preferential attachment (PA) we mean degree-based preferential attachment.

Before presenting our parallel algorithms for generating PA networks, we briefly discuss the sequential algorithms for the same. Many preferential attachment-based models have been proposed in the literature. Two of the most prominent models are the Barabási–Albert model [8] and the copy model [17] as discussed below.

2.3 Sequential Algorithm: Barabási–Albert Model

One way to generate a random PA network is to use the Barabási–Albert (BA) model. Many real-world networks have two important characteristics: (1) They are evolving in nature and (2) the network tends to be scale-free [8]. In the BA model, a new vertex is connected to an existing vertex that is chosen with probability directly proportional to the current degree of the existing vertex.

The BA model works as follows: Starting with a small clique of $\hat{d}$ vertices, in every time step, a new vertex t is added to the network and connected to $d \leq \hat{d}$ randomly chosen existing vertices: $F_\ell(t)$ for $1 \leq \ell \leq d$ with $F_\ell(t) < t$; that is, $F_\ell(t)$ denotes the ℓth vertex which t is connected. Thus, each phase adds d new edges $(t, F_1(t)), (t, F_2(t)), \ldots, (t, F_d(t))$ to the network, which exhibits the evolving nature of the model. Let $\mathbb{F}(t) = \{F_1(t), F_2(t), \ldots, F_d(t)\}$ be the set of outgoing vertices from t. Each of the d end points in the set $\mathbb{F}(t)$ is randomly selected based on the degrees of the vertices in the current network. In particular, the probability $P_i(t)$ that an outgoing edge from vertex t is connected to vertex $i < t$ is given by $P_i(t) = \frac{d_i}{\sum_j d_j}$, where d_j represents the degree of vertex j.

The networks generated by the BA model are called the BA networks, which bear the aforementioned two characteristics of a real-world network. BA networks have power-law degree distribution. A degree distribution is called power law if the probability that a vertex has degree d is given by $\Pr[d] \propto d^{-\gamma}$, where $\gamma \geq 1$ is a positive constant. Barabási and Albert showed that the preferential attachment method of selecting vertices results in a power-law degree distribution [8].

A naïve implementation of network generation based on the BA model takes $\Omega(n^2)$ time where n is the number of vertices. Batagelj and Brandes give an efficient algorithm with a running time of $\mathcal{O}(m)$ where m is the number of edges [9]. This algorithm maintains a list of vertices such that each vertex i appears in this list exactly d_i times. The list can easily be updated dynamically by simply appending u and v to the list whenever a new edge (u, v) is added to the network. Now, to find $F(t)$, a vertex is chosen from the list uniformly at random. Since each vertex i occurs exactly d_i times in the list, we have the probability $\Pr[F(t) = i] = \frac{d_i}{\sum_j d_j}$.

2.4 Sequential Algorithm: Copy Model

As it turns out, the BA model does not easily lend itself to an efficient parallelization [4]. Another algorithm called the *copy model* [16, 17] preserves preferential attachment and power-law degree distribution. The copy model works as follows: Similar to the BA model, it starts with a small clique of $\hat{d}$ vertices and in every time step, a new vertex t is added to the network to create $d \leq \hat{d}$ connections to existing vertices $F_\ell(t)$ for $1 \leq \ell \leq d$ with $F_\ell(t) < t$. For each connection $(t, F_\ell(t))$ from vertex t, the following steps are executed:

Step 1: First, a random vertex $k \in [0, t-1]$ is chosen with uniform probability.

Step 2: Then, $F_\ell(t)$ is determined as follows:

$$F_\ell(t) = k \text{ with prob. } p \quad \text{(Direct edge)} \tag{1}$$

$$= F_l(k) \text{ with prob. } (1-p) \quad \text{(Copy edge)} \tag{2}$$

where l is a random outgoing connection from vertex k.

We also denote $\mathbb{F}(t) = \{F_1(t), F_2(t), \ldots, F_d(t)\}$ to be the set of outgoing vertices from vertex t.

It can be easily shown that a connection from vertex t to vertex i is made with probability $\Pr[i \in \mathbb{F}(t)] = \frac{d_i}{\sum_j d_j}$ when $p = \frac{1}{2}$. Thus, when $p = \frac{1}{2}$, this algorithm follows the Barabási–Albert model as shown in [2, 4].

Thus, the copy model is more general than the BA model. It has been previously shown [17] that the copy model produces networks with degree distribution that follows a power-law $d^{-\gamma}$, where the value of the exponent γ depends on the choice of p. Further, it is easy to see the running time of the copy model is $\mathcal{O}(m)$. The copy model has been used to develop efficient parallel algorithms for generating preferential attachment networks in distributed-memory and shared-memory machines [4, 7]. In our work presented in this paper, we adopt the copy model as a starting point to design and develop our GPU-based parallel algorithm.

3 GPU-based Parallel Algorithm: cuPPA

The PA model imposes a critical dependency that every new vertex needs to have the state of the previous network to compute its edges. This poses a major challenge in

parallelizing preferential attachment algorithms. In phase v, to determine $F(v)$, it requires that F_i is known for each $i < v$. As a result, any algorithm for preferential attachment apparently seems to be highly sequential in nature: Phase v cannot be executed until all previous phases are completed.

In [4], a distributed-memory-based algorithm was proposed that exploits the copy model to relieve this sequentiality and run in parallel. We re-examined that exploitation and designed **cuPPA**, an efficient parallel algorithm for generating preferential attachment-based networks on a single GPU as described next. Here, we assume that the entire network can be stored in the GPU memory.

Let T be the number of threads in the GPU. The set of vertices V is partitioned into T disjoint subsets of vertices $V_0, V_1, \ldots, V_{T-1}$; that is, $V_i \subset V$, such that for any i and j, $V_i \cap V_j = \emptyset$ and $\bigcup_i V_i = V$. Thread $\mathcal{T}_i$ is responsible for computing and updating $F(v)$ for all $v \in V_i$. The algorithm starts with an initial network, which is a clique of the first d vertices labeled $0, 1, 2, \ldots, d-1$. For each vertex v, the algorithm computes d edges $(t, F_1(v)), (t, F_2(v)), \ldots, (t, F_d(v))$ and ensures that such edges are distinct without any parallel edges. We denote the set of vertices $\{F_1(v), F_2(v), \ldots, F_d(v)\}$ by $\mathbb{F}(v)$. The algorithm works in two phases. In the first phase of the algorithm (called *execute copy model*), we execute the copy model for all vertices in parallel (using all threads). This phase creates all the direct edges and some of the "copy" edges (Eq. 2). However, many copy edges might not be fully processed due to the dependencies. The incomplete copy edges are put in a waiting queue called $\mathcal{Q}$. In the second phase of the algorithm (called *resolve incomplete edges*), we resolve the incomplete edges from the waiting queue $\mathcal{Q}$ and finalize the copy edges. The pseudocode of cuPPA is given in Algorithm 1. A list of symbols used in the paper is presented in Table 1.

Algorithm 1: cuPPA

n	Number of vertices
d	Number of outgoing edges from each vertex
p	Probability of creating a direct edge
V_i	The set of vertices processed by thread $\mathcal{T}_i$
$\mathbb{F}(u)$	The set of outgoing ends of edges from vertex u
$F_i(u)$	The i-th outgoing edge from vertex u
$\mathcal{Q}$	A queue for the current set of unfinished edges
$\mathcal{Q}'$	A queue for the next set of unfinished edges

```
1  with T threads do in parallel      /* Each thread T_i executes the following in parallel: */
       // Phase 1: Execute Copy Model
2      foreach v ∈ V_i do
3          for ℓ = 1 to d do
4              u ← a uniform random vertex in [0, v − 1]
5              c ← a uniform random number in [0, 1]
6              if c < p then                                   // i.e., with prob. p
7                  if u ∉ F(v) then
8                      F_ℓ(v) ← u
9              else
10                 l ← a uniform random integer in [1, d]
11                 if F_l(u) ≠ NULL and F_l(u) ∉ F(v) then     // Resolved
12                     F_ℓ(v) ← F_l(u)
13                 else                                        // Unresolved edge into Q
14                     F_ℓ(v) ← NULL
15                     Add ⟨u, l⟩ to Q
16
       // Phase 2: Resolve Incomplete Edges
17     while Q ≠ ∅ do
18         foreach ⟨u, l⟩ ∈ Q do
19             if F_l(u) ≠ NULL then
20                 F_ℓ(v) = F_l(u)
21             else
22                 Append ⟨u, l⟩ to Q'
23         Swap Q and Q'
24         Q' ← ∅
```

Table 1 Symbols used in this paper

Symbol	Description
n	The number of vertices
V	The set of vertices
m	The number of edges
E	The set of edges
T	The number of threads
d	The number of outgoing edges generated from each new vertex
p	The probability of creating a direct edge in the copy model
$N(v)$	The set of neighbors of vertex v
d_v	The degree of vertex v
$F_t(k)$	The outgoing end of kth edge from vertex t
$\mathbb{F}_t$	The set of outgoing ends of edges from vertex t
$\mathcal{Q}$	A queue for the current set of unfinished edges
$\mathcal{Q}'$	A queue for the next set of unfinished edges

In the first phase (lines 3–21), the algorithm executes the copy model for all of its vertices. The edges that could not be completed are stored in a queue $\mathcal{Q}$ to be processed later. We call the queue a waiting queue. Each of the other vertices from d to $n-1$ generates d new edges. There are fundamentally two important issues that need to be handled: (1) how we select $F_\ell(v)$ for vertex v where $1 \leq \ell \leq d$, and (2) how we avoid duplicate edge creation. Multiple edges for a vertex v are created by repeating the same procedure d times (line 4), and duplicate edges are avoided by simply checking if such an edge already exists—such a check is done whenever a new edge is created.

For the ℓth edge of a vertex v, another vertex u is chosen from $[1, v-1]$ uniformly at random (line 5, 6). Edge (v, u) is created with probability p (line 7). However, before creating such an edge (v, u) in line 8, the existence of such an edge is checked immediately before creating them in line 9. If the edge already exists at that time, the edge is discarded and the process is repeated again (line 5). With the remaining $1-p$ probability, v is connected to some vertex in $\mathbb{F}(u)$; that is, we make an edge $(v, F_\ell(u))$, such that ℓ is chosen from $[1, d]$ uniformly at random.

After the first phase is completed, the algorithm starts to resolve all incomplete edges by processing the waiting queue (lines 22–29). If an item in the current queue $\mathcal{Q}$ could not be resolved during this step, it is subsequently placed in another queue $\mathcal{Q}'$. After all incomplete edges on the queue $\mathcal{Q}$ are processed, the queues $\mathcal{Q}$ and $\mathcal{Q}'$ are swapped and $\mathcal{Q}'$ is cleared. We repeat this process until both the queues are empty. Note that there is no circular dependency in the copy edges. For two vertices $u > v$, copy edges from u may depend on edges from v but not vice versa. Therefore, there would be no circular waiting and no deadlock situation for the waiting queue to complete.

3.1 Graph Representation

We use one array G of nd elements to represent and store the entire graph. Each vertex u connects to d existing vertices. The neighbors of u are stored between the indices inclusive from ud to $(u+1)d-1$ that represents the other end-point vertices. We call these indices the *outgoing vertex list* for vertex u. The initial network consists of the d^2 vertices from the start of the array. For any edge u, v where $u > v$ and $u, v > d$, the edge is represented by storing v in one of the d items in the *outgoing vertex list* of u. Note that the graph G contains exactly nd edges as defined by the Barabási–Albert or the copy model. Any vertex with the index $0 \leq i < nd$ of the array G denotes the $(t \mod d)$th end point of the vertex $\frac{i}{d}$.

3.2 Partitioning and Load Balancing

Recall that we distribute the computation among the threads by partitioning the set of vertices $V = \{0, 1, \ldots, n-1\}$ into T subsets $V_0, V_1, \ldots, V_{T-1}$ as described at the beginning of Sect. 3, where T is the number of available threads. Although several partitioning schemes are possible, our study suggests that the *round-robin partitioning* (RRP) scheme best suits our algorithm. In this scheme, vertices are distributed in a round-robin fashion among all threads. Partition V_i contains the vertices $\langle i, i+T, i+2T, \ldots, i+kT \rangle$ such that $i + kT \leq n < i + (k+1)T$; that is, $V_i = \{j | j \mod T = i\}$. In other words, vertex i is assigned to set $V_{i \mod T}$. Therefore, the number of vertices in the sets is almost equal., i.e., the number of vertices in a set is either $\left\lceil \frac{n}{T} \right\rceil$ or $\left\lfloor \frac{n}{T} \right\rfloor$. The round-robin partitioning scheme is illustrated in Fig. 1.

3.3 Segmented Round-Robin Partitioning

However, the naïve round-robin scheme discussed above also has some technical issues. As described in Sect. 3, the first phase of Algorithm 1 executes the copy model for every vertex assigned to it and stores any unresolved copy edge in the waiting queue. In the second phase, the algorithm takes out each unresolved edge from the waiting queue and tries to resolve them. To reduce the memory latency accessing the waiting queue, we store the waiting queue $\mathcal{Q}$ in the GPU *shared memory* that offers many folds faster memory access than the global GPU memory. Note that this memory is limited in capacity and is shared among all threads running within the same block. Modern GPUs such as NVidia GeForce 1080 have 48 KB of ultra-fast shared memory per block. Since the amount of the shared memory is very limited, it can only store a limited number of unresolved items in the queue. Let $\mathcal{C}$ denote the total capacity of the waiting

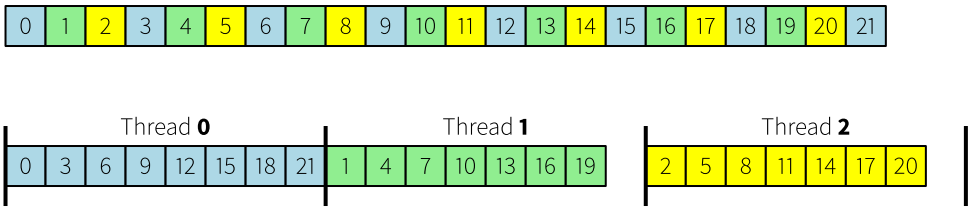


Fig. 1 Distributing 21 vertices among 3 threads using round-robin partitioning

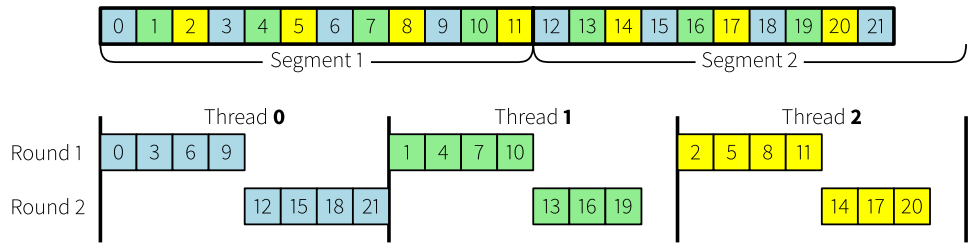


Fig. 2 Distributing 21 vertices among 3 threads using segmented round-robin partitioning with 2 rounds

queue. For example, with a 48 KB of shared memory, we have a total capacity to store $C = \frac{48\times1024}{8} = 6144$ items in the waiting queue where each item takes 8 bytes of memory. If we use τ threads per block, each thread will have a capacity of $\frac{C}{\tau}$ items to be placed in the waiting queue. Therefore, if the number of vertices assigned to a thread is too large, it may generate a large number of unresolved copy edges to be placed in the waiting queue, essentially forcing the algorithm to use a large amount of GPU memory instead of the available shared memory.

In order to exploit the faster shared memory without overflowing the waiting queue capacity, we use a modified round-robin partitioning scheme called, *segmented round-robin partitioning* (SRRP). In this scheme, the entire set of vertices V is first partitioned into some k consecutive subsets $S_1, S_2, S_3 \ldots S_k$ called *segments*. From the definition of the copy model, it is clear that vertices on a segment S_i may only depend on vertices on segment S_j where $i \geq j$ but not vice versa. Therefore, the segments have to be processed in a consecutive fashion. Let $B_i = |S_i|$ denote the number of elements (also called the segment size) in segment S_i where $1 \leq i \leq k$. Next, the parallel algorithm is executed in k consecutive rounds where round i executes the parallel algorithm for all the vertices in segment S_i. In round i, the B_i vertices in segment S_i are further partitioned into T subsets $V_0(S_i), V_1(S_i), \ldots V_{T-1}(S_i)$, using the round-robin scheme discussed above and executed in parallel using the T threads. The technique is illustrated in Fig. 2.

Next, we need to determine the best segment size to avoid overflow while using the shared memory. From the copy model, it is easy to see that the lower the probability p is, the more likely it is to be in the waiting queue. In the worst case, when $p = 0$, all generated edges consist of copy edges. Therefore, at most d unresolved copy edges could be placed in the waiting queue per vertex. Additionally, as the value of d gets bigger, the number of copy edges increases and hence, the waiting queue size increases. Therefore, p and d both have a significant impact on the required size of the waiting queue. Having that in mind, we use two approaches for the segment size:

- **Fixed Segment Size:** The simplest way is to use a fixed sized segments in each round. From the previous discussion, it is clear that in the worst case we need d items per vertex to be placed on the waiting queue. Therefore, we can use up to $\tau = \min\left(\frac{C}{d}, \theta\right)$ threads per block where C is the total queue capacity and θ is the maximum number of threads per block. Then, the segment size is $\frac{C}{d\tau}$ vertices per segment. Note that we can exploit the shared memory for $d \leq C$; otherwise, we need to use the global memory. However, in almost all practical scenarios we have $d \ll C$; hence, we can take advantages of the shared memory.
- **Dynamic Segment Size:** Although the fixed segment size scheme ensures that the queue will not overflow in any round, it may not be the most efficient implementation. We use another scheme where the segment size is determined dynamically between two rounds based on the current state of the algorithm. In this scheme, we start with the number of threads per block τ and the segment size $\frac{C}{d\tau}$ vertices per segment as was done in the fixed segment size scheme. However, at the end of each round, we determine the maximum number of items that were placed in the waiting queue per thread. If the number of items placed in the waiting queue in the round is less than some f factor of the waiting queue capacity per thread $\frac{C}{\tau}$, we increase the total capacity C by a factor of f (typically, we set $f = 2$). Before the next round, we recompute the required number of threads per block and update the segment size accordingly.

3.4 CUDA-Specific Deadlock Scenario

In the round-robin scheme, completion of a copy edge of a vertex in a thread $\mathcal{T}_i$ may depend on some other thread $\mathcal{T}_j$ where $i \neq j$. Due to the nature of dependency, $\mathcal{T}_j$ also may have a copy edge that depends on another vertex that belongs to $\mathcal{T}_i$. Therefore, if any of these threads are not running simultaneously on the GPU, the other thread will not be able to complete and a deadlock situation may arise. To avoid such a situation, we must ensure that either all the GPU threads are running concurrently or the dependent threads are put to sleep for a while. In the current CUDA framework, the runtime engine schedules each kernel block to a streaming multiprocessor, and the blocks of running threads are non-preemptible. Therefore, to ensure that threads are running concurrently to avoid deadlock situation, we cannot use more blocks than the number of available streaming multiprocessors. Note that the upcoming CUDA runtime supports *cooperative groups*. On such future systems, the deadlock situation could be avoided using block sizes larger than the number of shared multiprocessors.[1]

4 Generating Networks Using Multiple GPUs

So far we have presented an algorithm to generate the preferential attachment-based model using a single GPU. Our algorithm works well if the entire network can be stored in the GPU memory. However, the size of the generated network is limited by the amount of GPU memory. Nowadays, it is very common to have multiple GPUs in a computing cluster, even on commodity machines. Therefore, we could potentially use multiple GPUs to generate even larger networks. In this section, we discuss how **cuPPA** can be extended for multiple GPUs.

Similar to distributing the works of generating edges into multiple threads as discussed at the beginning of the section, we need to distribute the vertices into multiple GPUs. Let the vertices V be partitioned into g subsets $\mathbb{V}_1, \mathbb{V}_2, \ldots, \mathbb{V}_g$ where g is the number of available GPUs. GPU i processes the edges generated by the vertices $\mathbb{V}_i$. Next, we execute the cuPPA algorithm on each of the GPUs with their set of vertices. We can immediately see that computing phase 1 of the algorithm can be done independently in all GPUs. However, resolving the incomplete edges in phase 2 of the algorithm requires careful attention. Due to the nature of the dependency, an incomplete edge on a GPU may require the information resident on the memory of another GPU. Therefore, a synchronization among the GPUs is required. Such a synchronization technique between CPUs with distributed memory was presented in [4]. Although the technique

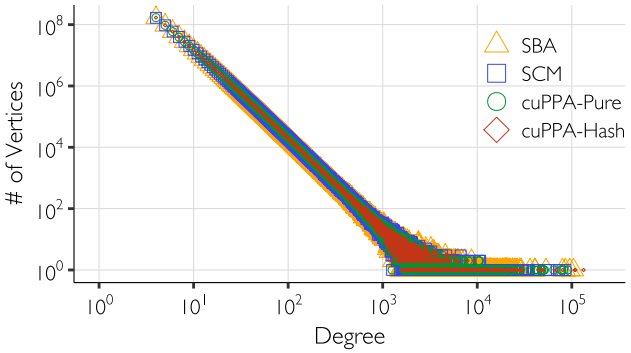


Fig. 3 The degree distributions of the PA Networks ($n = 500M$, $d = 4$). In log–log scale the degree distribution is a straight line validating the scale-free property. Further, all four models produce almost identical degree distributions showing that both versions of cuPPA produce networks with accurate degree distributions

can be adapted for synchronization between the GPUs, it requires complex and intricate communications between the GPUs. NVidia CUDA offers another scheme for accessing memory across multiple GPUs called the "unified memory addressing." In this case, a single memory address space accessible from any processor in a system is available from the CUDA runtime application programming interface. Therefore, any GPU can access the memory of other GPUs. However, due to the nature of random and sparse memory access, the approach would not yield the desired benefit. In the next section, we present **cuPPA-Hash**, an alternative algorithm to generate networks using the preferential attachment model using hash functions instead of pseudorandom number generators.

4.1 cuPPA-Hash: A Hash Function-Based Implementation

Notice that the dependency of generating an edge on other vertices only arises while creating a copy edge, i.e., when a vertex u tries to connect to an random end point of another vertex v. We adapt an idea previously used for communication-free parallel generation of BA graphs [24] to a GPU setting. Consider a vertex u copying an end point from a vertex v. Rather than looking up this value from a memory cell that is filled when vertex v is generated, the end point is recomputed independently. This is possible using a hash function to generate the random numbers instead of using pseudorandom numbers. This approach has the additional benefit that the exact same graph can be reproduced using the same hash functions. We also extend [24] by developing a more general preferential attachment-based algorithm using the copy model called **cuPPA-Hash**.

[1] https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/.

Algorithm 2: cuPPA-Hash

```
1  Function cuPPA-Hash(𝕍_k)
2      with T threads do in parallel /* Each thread 𝒯_i executes the following in
       parallel: */
3          foreach v ∈ V_i do
4              for ℓ = 1 to d do
5                  F_ℓ(v) ← Calculate–Edge(v, ℓ)
6  Function Calculate–Edge(v, ℓ)
7      e ← vd + ℓ                          /* The number of edges up to vertex v − 1 */
8      r ← hash(e) % e
9      u ← ⌊r/d⌋,    l ← r % d             /* r denotes the l-th edge from vertex u */
10     if u < d then
11         return u                                  // Connect to an initial vertex
12     else
13         r ← hash_f(e)
14         if r < p then
15             return u                          // i.e., Direct Edge with prob. p
16         else
17             return Calculate–Edge(u, l)   // i.e., Copy Edge with prob. 1 − p
```

A simplified pseudocode of **cuPPA-Hash** is presented in Algorithm 2. The initial set of vertices is first divided into g mutually exclusive subsets $\mathbb{V}_1, \ldots, \mathbb{V}_g$ where g is the number of GPUs. Next each GPU k processes the vertices in set $\mathbb{V}_k$ using the procedure **cuPPA-Hash** ($\mathbb{V}_k$). We further partition the set of vertices $\mathbb{V}_k$ into T subsets $V_0, V_1, \ldots, V_{T-1}$ where T is the number of threads on GPU k. Thread i executes the copy model on the vertices in set V_i (line 3). For each of the d outgoing edge of a vertex v, the function Calculate-Edge calculates the end-point vertex using the copy model. The ℓth outgoing edge of the vertex v is uniquely denoted by the index $e = vd + \ell$ (line 7). The edge index e is used to generate a hash value r using a hash function. We used a 64-bit CRC64 as our hash function. Note that r denotes the index of the lth outgoing edge of vertex u calculated in line 9. If $u < d$ then u denotes an initial vertex and we connect $F_\ell(v)$ to u (line 11). Otherwise, we compute a floating point number r using a floating point version of the hash function (line 14). If $r < p$ (i.e., with probability p), we connect $F_\ell(v)$ to u. Otherwise, we calculate the outgoing edge of $F_l(u)$ (line 17) recursively. In the actual implementation, we use an iterative function instead of the recursive one.

Note that the algorithm does not require to access any GPU memory pertaining to other GPUs. Instead all copy edges are essentially recomputed. Therefore, this approach requires more computation than the original cuPPA algorithm. However, due to the independent computations, the algorithm scales very well to multiple GPUs as shown in the experimental section.

5 Experimental Results

In this section, we evaluate our algorithm and its performance by experimental analysis. In the following sections, we denote our first algorithm using pseudorandom number generators as **cuPPA-Pure** and the second algorithm using hash function as **cuPPA-Hash**. We demonstrate the accuracy of our algorithm by showing that our algorithm produces networks with power-law degree distribution as desired. We also compare the runtime of our algorithm using other sequential and parallel algorithms.

5.1 Hardware and Software

We used a computer consisting of 6 AMD Phenom(tm) II 6174 processor with 3.3 GHz clock speed and 64 GB system memory. The machine also incorporates a NVidia 1080 GPU with 8 GB memory. The operating system is `Ubuntu 16.04 LTS`, and all software on this machine was compiled with GNU `gcc 4.6.3` with optimization flags `-O3`. The `CUDA compilation tools V8` were used for the GPU code along with `nvcc` compiler. In additional experiments, we used another system consisting of 4 NVidia Tesla P100 GPUs with 16 GB memory each.

5.2 Degree Distribution

To demonstrate the accuracy of **cuPPA-Pure** and **cuPPA-Hash**, we compared those with the sequential Barabási–Albert (SBA) [9] and the sequential copy model (SCM) algorithms. The degree distributions of the networks

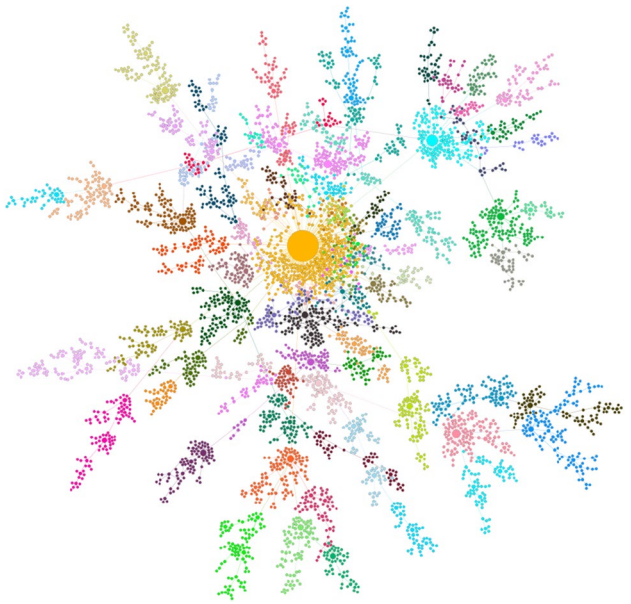


Fig. 4 Visualization of networks generated by **cuPPA** using $n = 10{,}000$, $p = 0.5$ and $d = 1$

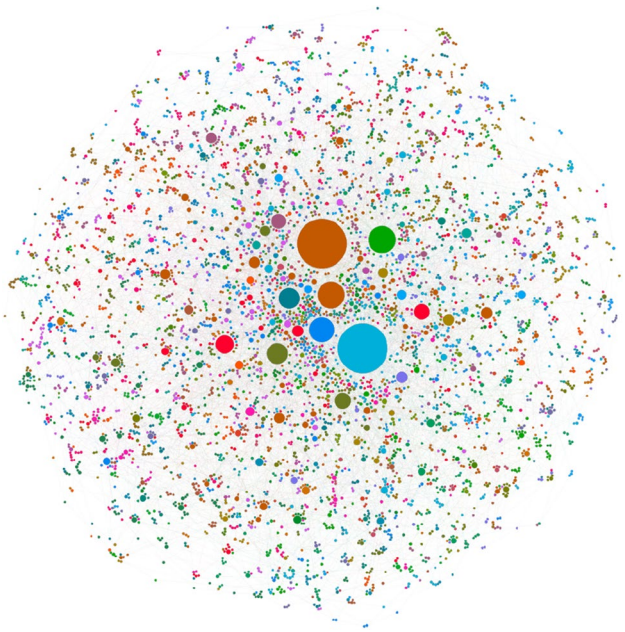


Fig. 5 Visualization of networks generated by **cuPPA** using $n = 10{,}000$, $p = 0.5$ and $d = 2$

generated by SBA, SCM, **cuPPA-Pure**, and **cuPPA-Hash** are shown in Fig. 3 in a log–log scale. We used $n = 500M$ vertices, each generating $d = 4$ new edges with a total of two billion (2×10^9) edges. The distribution is heavy-tailed, which is a distinct feature of power-law networks. The exponent γ of the power-law degree distribution is measured to be 2.7. This supports the fact that for the finite average degree of a scale-free network, the exponent should be $2 < \gamma < \infty$ [12]. Also notice that the degree distributions of SBA and SCM are quite identical. The degree distributions of both **cuPPA** algorithms are also similar to both SBA and SCM.

5.3 Visualization of Generated Graphs

In order to gain an idea of the structure and degree distributions, we obtained a visualization of some of the networks generated by our algorithm. We generated the visualizations using a popular network visualization tool called `Gephi`. Bearing aesthetics in mind and to minimize undue clutter, we focused on a few small networks by choosing $n = 10{,}000$, $p = 0.5$, and $d = 1, 2, 4$. The visualizations are shown in Figs. 4, 5, and 6.

5.4 Effect of Edge Probability on Degree Distribution

As mentioned earlier, the strength of the copy model is the capability of generating other preferential attachment networks by simply varying one parameter, namely, the probability p. In Fig. 7, we display the degree distribution of the

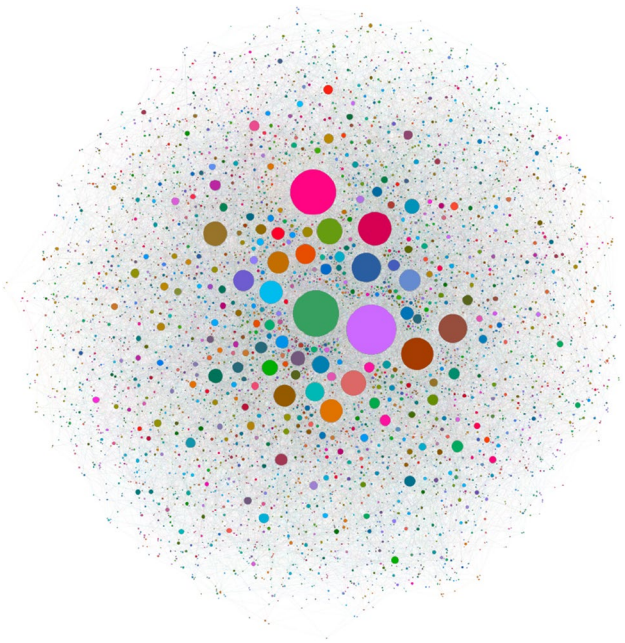


Fig. 6 Visualization of networks generated by **cuPPA** using $n = 10{,}000$, $p = 0.5$ and $d = 4$

generated networks by varying p using both **cuPPA-Pure** and **cuPPA-Hash**. When $p = 0$, all edges are produced by copy edges, and thus, the network becomes a star network where all additional vertices connect to the d initial vertices. With a small value of p ($p = 0.01$), we can generate a

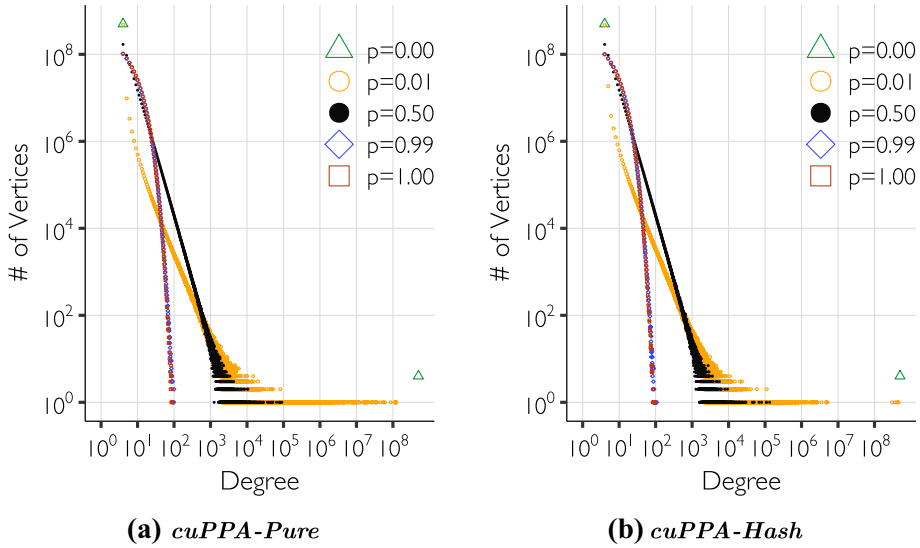


Fig. 7 The degree distributions of the networks by cuPPA ($n = 500M$, $d = 4$) with varying p

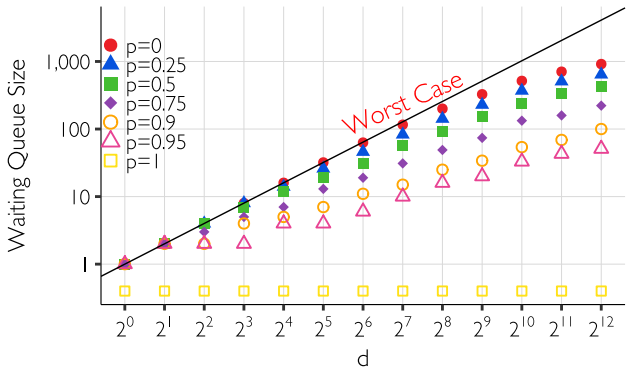


Fig. 8 Maximum size of the waiting queue per thread for different values of p and d (both axes in log scale). In the worst case ($p = 0$), the maximum size increases linearly with d for smaller values ($d \leq 64$). For larger d, the actual maximum size of waiting queue is comparatively smaller than the worst case

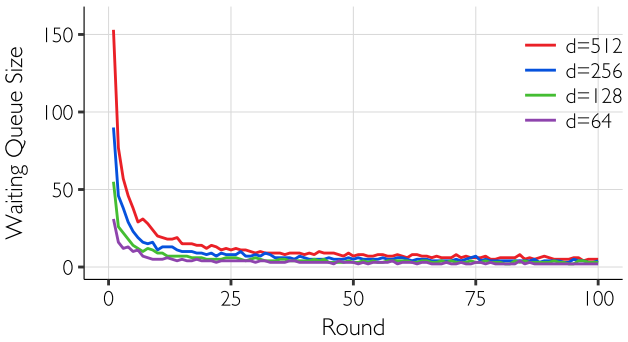


Fig. 9 Size of the waiting queue decreases significantly with rounds in SRRP scheme

network with a very long tail. When we set $p = 0.5$, we get the Barabási–Albert networks which exhibit a straight line in log–log scale. When we increase p to 1, we get a network consisting entirely of direct edges that do not form any tail.

5.5 Waiting Queue Size of cuPPA-Pure

As mentioned in Sect. 3.3, the waiting queue size depends on p and d. To evaluate the impact of p and d, we ran simulations using 1280 CUDA threads (20 blocks and 64 threads per block) where each thread only executed one vertex. The value of p is varied from 0 to 1 with different probability values. We also varied the value of d from 1 to 4096 as increasing powers of 2. In Fig. 8, we show the number of items placed in the waiting queue per vertex for different combinations of p and d. We also added the worst case value as a line in the plot. As seen from the figure, in the worst case with $p = 0$, the maximum size of the waiting queue increases linearly with d for smaller values of d (up to 64) and afterward, it does not increase much compared to d. Therefore, for smaller values of d, we need to have provisions for at least d items per vertex in the waiting queue.

However, as the round progresses, the maximum size of the waiting queue decreases significantly as shown in Fig. 9. For this figure, we also ran cuPPA using 1280 CUDA threads (20 blocks and 64 threads per block) to generate networks with $d = 512, 256, 128, 64$ and $p = 0.5$. Each CUDA thread processes exactly one vertex per round. Only the first 100 rounds are shown for brevity. From Fig. 9, we can see that as the round progresses, the size of the waiting queue per round decreases dramatically for all different values of d. This indicates that we could process more vertices in later rounds using the same amount of queue memory. Therefore, we can dynamically change the size of the segments between two consecutive rounds to increase parallelization. Based on these observations

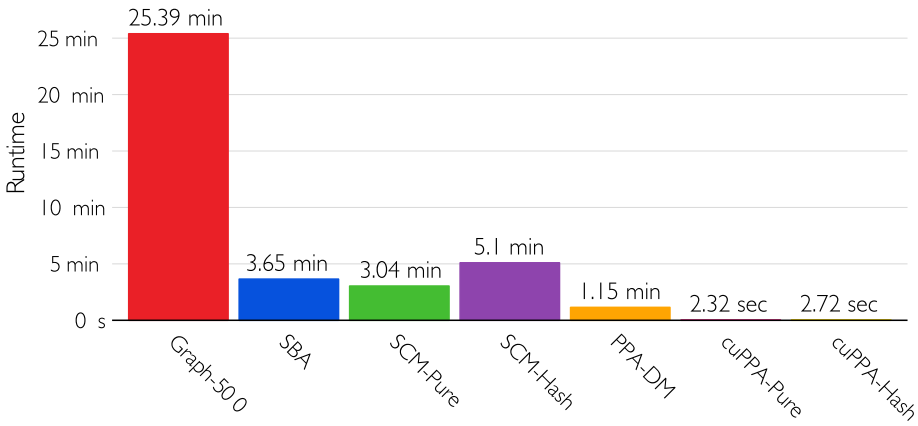


Fig. 10 Runtime of Graph500 generator, SBA, SCM, PPA-DM, and cuPPA for generating two billion edges ($n = 500M$, $d = 4$). Both of our cuPPA algorithms can generate the network in less than 3 s

regarding the size of the waiting queue, we designed an adaptive version of **cuPPA-Pure** that monitors the maximum size of the waiting queue and manages the segment size accordingly as discussed in Sect. 3.3. We call this version **cuPPA-Dynamic** and use it for all other experiments.

5.6 Runtime Performance

In this section, we analyze the runtime and performance of **cuPPA-Pure** and **cuPPA-Hash** relative to other algorithms and show the variation of performances against various parameters.

5.6.1 Runtime Comparison with Existing Algorithms (Single GPU)

To the best of our knowledge, our algorithm is the first GPU-based parallel algorithm to generate preferential attachment networks. Therefore, it is not possible to compare the runtime with other GPU-based algorithms. Instead, we compare with the existing non-GPU algorithms. The total run times of both versions of cuPPA and the existing algorithms are shown in Fig. 10 for generating two billion edges ($n = 500M$, $d = 4$). In this experiment, we used a single NVidia GeForce 1080 GPU with 8 GB memory.

– **Sequential Algorithms:** We compare cuPPA with two efficient sequential algorithms: SBA [9] and SCM [17]. For SCM, we used two implementations: one with the pseudorandom number generators (called SCM-Pure) and the other with hash functions (called SCM-Hash). We also compared our algorithm with a reference sequential graph generation library from the Graph500 [1] reference code that uses SKG to generate networks.

  As shown in Fig. 10, SKG from Graph500 takes the longest time to generate 2B scale-free networks—25.39 minutes. In comparison, our GPU-based algorithm is 650× faster.

  We also found that SCM-Pure is slightly faster than the SBA algorithm. The hash-based SCM-Hash essentially recomputes all the copy edges and therefore takes approximately 70% more time than the SCM-Pure algorithm. However, the hashing technique is shown to scale to a large number of processors making it a viable candidate for large network generation using many processors [24]. On the other hand, the GPU-based **cuPPA-Pure** generates the network in just 2.32 s on the NVidia 1080 GPU with 78× to 94× speedup. Also note that **cuPPA-Hash** is slightly slower than **cuPPA-Pure** on a single GPU due to more computation.

– **Parallel Algorithms:** We also compared cuPPA with a distributed-memory (PPA-DM) [4] and a shared-memory (PPA-SM) [7] parallel algorithms. As shown in Fig. 10, both of the cuPPA algorithms outperform PPA-DM on a system with 24 processors. The main reason is that unlike PPA-DM, cuPPA algorithms do not require complex synchronizations and message communications.

  Due to the unavailability of the PPA-SM code, we compared the runtime to generate the largest graph ($n = 10^7, d = 10$) reported in [7] with the corresponding runtime of cuPPA. PPA-SM generates the network using 16 cores of Intel Xeon CPU E5-2698 2.30 GHz in approximately 7.5 s, whereas **cuPPA-Pure** generates the same network in just 0.3 s.

5.6.2 Runtime Versus Number of Vertices (Single GPU)

First, we examine the runtime performance of **cuPPA-Pure** (fastest of the two algorithms in a single GPU) with increasing number of vertices n. Here, we examine two cases. In the first case, we set $d = 4$, vary $p = \{0, 0.001, 0.25, 0.5, 0.75, 1\}$, and vary $n = \{1.9M, 3.9M, 7.8M, 15.6M, 31.25M, 62.5M, 125M, 250M, 500M\}$ to see how the runtime changes with increasing number of vertices for different p. The corresponding runtime is shown in Fig. 11. In the second case, we set $p = 0.5$, vary

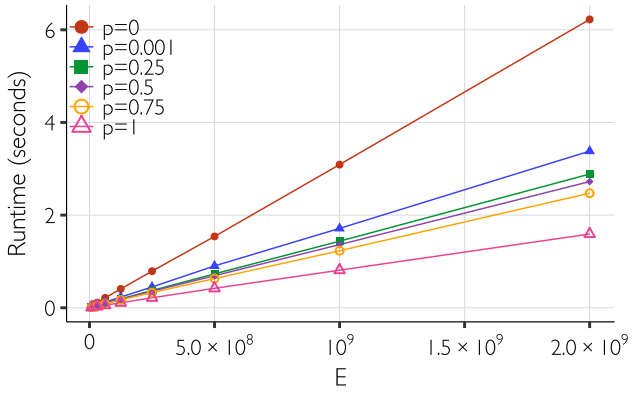


Fig. 11 Runtime versus number of edges suggests that **cuPPA** is very scalable with increasing n for different values of p with a fixed value of $d = 4$

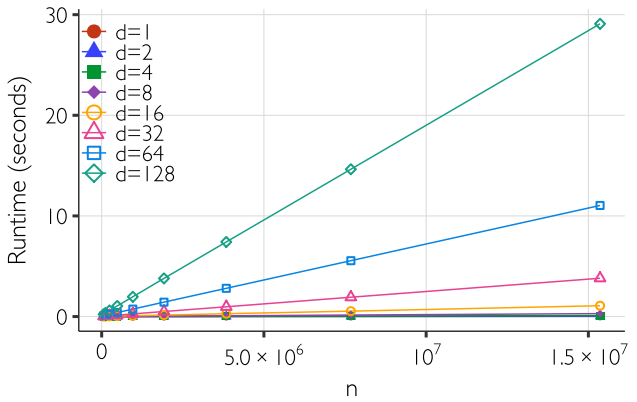


Fig. 12 Runtime versus number of vertices suggests that **cuPPA** is very scalable with increasing n for different values of d with a fixed value of $p = 0.5$

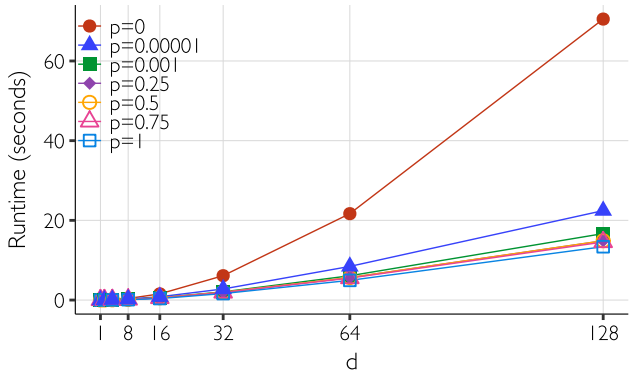


Fig. 13 Runtime versus d for generating networks with $n = 7812500$ with varying $d = 1, 2, 4, 8, 16, 32, 64, 128$ for different values of p. The runtime almost increases linearly

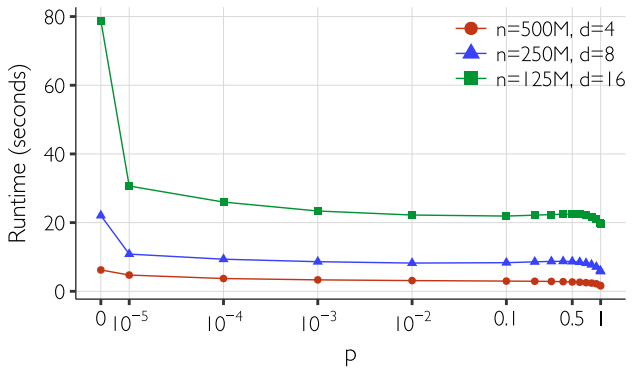


Fig. 14 Runtime versus p for three sets of values for n and d (x-axis in log scale). At $p = 0$ the runtime is the largest which reduces significantly with a slight increase. As p increases, the runtime reduces

$d = \{1, 2, 4, 8, 16, 32, 64, 128\}$, and vary $n = \{60K, 120K, 240K, 480K, 960K, 1.92M, 3.84M, 7.68M\}$ to see how the runtime changes with increasing number of vertices for different d. The corresponding runtime is shown in Fig. 12.

From Figs. 11 and 12, we can observe that for any fixed set of values for p and d, with increasing n, the runtime increases linearly, indicating that the algorithm scales very well with increasing value of n.

5.6.3 Runtime Versus Degree of Preferential Attachment (Single GPU)

Next, we examine the runtime performance of **cuPPA** with increasing d. The runtime is shown in Fig. 13. Here, we set $n = 7812500$, vary $p = \{0, 0.00001, 0.001, 0.25, 0.5, 0.75, 1\}$, and vary $d = \{1, 2, 4, 8, 16, 32, 64, 128\}$ to see how the runtime changes for increasing value of d for different p. As seen from the figure, with increasing d, the runtime increases almost linearly. Therefore, the algorithm is observed to scale well for increasing value of d. Note that higher values of d are typically unlikely. However, we included higher values of d for performance measurement purpose. Also notice that the runtime is the largest for $p = 0$. With a small value of $p = 0.00001$, the runtime drops significantly and does not change much for higher values of p. Since the typical values of p are much larger than 0, this observation suggests that cuPPA performs well for real-world scenarios.

5.6.4 Runtime Versus Probability of Copy Edge (Single GPU)

Next we examine the runtime performance of cuPPA with increasing p. The runtime is shown in Fig. 14. Here, we used three different sets of values for n and d ($\langle n = 500M, d = 4\rangle$, $\langle n = 125M, d = 16\rangle$, and $\langle n = 31.25M, d = 64\rangle$), and vary $p = \{0, 0.00001, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99, 1.00\}$. As seen from the figure, the runtime reduces dramatically with a slight increase of $p = 0$ to

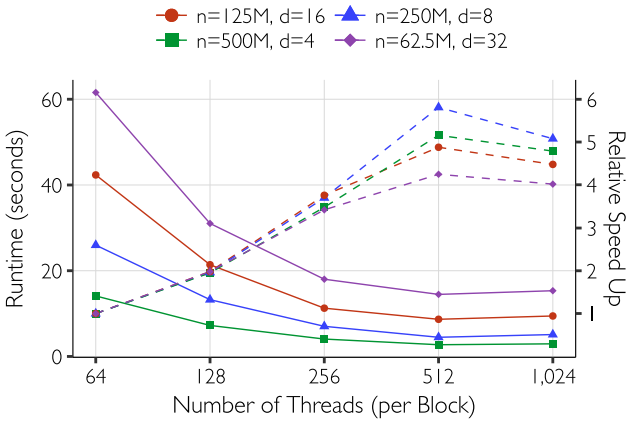


Fig. 15 Runtime (solid lines) and relative speedup (dashed lines) versus number of threads. Best performance is observed with 512 threads per block

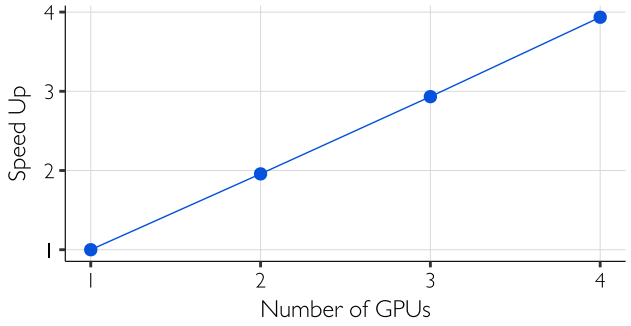


Fig. 16 Strong scaling of **cuPPA-Hash** for 4B edges ($n = 1B$ and $d = 4$). **cuPPA-Hash** exhibits perfect linear speedup

$p = 0.00001$ up to $p = 0.1$ in all of these three cases. Then, the runtime reduces almost linearly up to $p = 0.9$ and then reduces sharply toward $p = 1$. With lower values of p, most of the edges are produced by copy edges. Therefore, the size of the waiting queue increases, thereby increasing the runtime. As the value of p increases toward 1, most of the edges are created using direct edges, and therefore, fewer items are stored in the waiting queue.

5.6.5 Runtime Varied with the Number of Threads (Single GPU)

To understand how the performance of **cuPPA** depends on the number of threads, we set $p = 0.5$ and used four different sets of n and d to generate networks. We also varied the number of CUDA threads per block from 64, 128, 256, 512, to 1024. The runtime (solid lines) and the relative speedup (dashed lines) of the experiments are shown in Fig. 15. Let t_T be the runtime of cuPPA using T threads. Then, the relative speedup is defined as $\frac{t_T}{t_{64}}$ in this experiments, i.e., the speedup gained compared to the runtime of cuPPA using 64 threads. Figure 15 is shown in two y-axes, the left and right axis correspond to the runtime and relative speedup, respectively. From the figure, the best performance is observed with 512 threads per block for all cases. Therefore, in our final algorithm, we use up to 512 threads per block.

5.7 Runtime Performance of cuPPA-Hash (Multiple GPUs)

Next, we evaluate the performance of **cuPPA-Hash** for multiple GPUs. In this experiment, we used a machine consisting of 4 NVidia Tesla P100 GPUs with 16 GB memory each.

5.7.1 Strong Scaling

To study the strong scaling of the algorithm, we generated a network of 4B edges using $n = 1B$ and $d = 4$. We used 1 to 4 GPUs for the experiment. The strong scaling is presented in Fig. 16. From the figure, we can clearly see that **cuPPA-Hash** achieves perfect linear speedup by the virtue of being an embarrassingly parallel algorithm.

5.7.2 Generating Large Networks

Using **cuPPA-Hash** with 4 GPUs, we are able to generate a network of 16B edges ($n = 2B$ and $d = 8$) in just 7 s. That represents a rate of 2.29 billion edges per second, which is unprecedented in this domain.

6 Related Work

Although the concepts of random networks have been used and well studied over the last several decades, efficient algorithms to generate the networks were not available until recently. The first efficient sequential algorithm to generate Erdős–Rényi and Barabási–Albert networks was proposed in [9]. A distributed-memory-based algorithm to generate preferential attachment networks was proposed in [26]. However, their algorithm was not exact, rather an approximate algorithm and required manually adjusting several control parameters. The first exact distributed-memory-based parallel algorithm using the copy model was proposed in [4]. Another distributed-memory-based parallel algorithm using the Barabási–Albert model was proposed in [22, 24]. However, instead of using pseudorandom number generators, they used hash functions to generate the networks. A shared-memory-based parallel algorithm using the copy model was proposed in [7].

Several other theoretical studies were done on the preferential attachment-based models. Machta and Machta [21] described how an evolving network can be generated

in parallel. Dorogovtsev et al. [13] proposed a model that can generate graphs with fat-tailed degree distributions. In this model, starting with some random graphs, edges are randomly rewired according to some preferential choices. There exists other popular network models to generate networks with power-law degree distribution. R-MAT [10] and stochastic Kronecker graph (SKG) [19] models can generate networks with power-law degree distribution using matrix multiplication. Due to its simpler parallel implementation, the Graph500 group [1] choose the SKG model in their supercomputer benchmark. Highly scalable generators for Erdős-Renyi, 2D/3D random geometric graphs, 2D/3D Delaunay graphs, and hyperbolic random graphs are described in [15]. The corresponding software library release also includes an implementation of the algorithm described in [24]. An efficient and scalable algorithmic method to generate Chung–Lu, block two-level Erdős–Renyi (BTER), and stochastic blockmodels was also presented in [5].

There is a lack of GPU-based network generators in the literature. A GPU-based algorithm for generating Erdős–Rényi networks was presented in [23]. Another GPU-based algorithm for generating networks using the small-world model [25] was presented in [18]. However, until recently no GPU-based algorithm existed for the preferential attachment model. We introduced **cuPPA** as the first preferential attachment-based algorithm on the GPU using the copy model [3].

7 Conclusion

A novel GPU-based algorithm, named **cuPPA**, has been presented, with a detailed performance study, and its combination of its scale and speed has been tested by achieving the ability to generate networks with up to two billion edges in under 3 s of wall clock time. The algorithm is customizable with respect to the structure of the network by varying a single parameter, namely, a probability measure that captures the preference style of new edges in the preferential attachment model. Also, a high amount of concurrency in the generator's workload per thread or processor is observed when that probability is at very small fractions greater than zero. In future work, we intend to exploit code profiling tools for further optimization of the runtime and memory usage on the GPU. Also, the algorithm needs to be extended to exploit multiple GPUs that may be colocated within the same node. This would require periodic data synchronization across GPUs, which can be efficiently achieved using the NVidia Collective Communication Library (NCCL). Additional future work involves porting to GPUs spanning multiple nodes, and also hybrid CPU-GPU scenarios in order to utilize unused cores of multi-core CPUs. Methods to incorporate other network generator models can also be explored with our **cuPPA** as a starting point. Finally, future work is needed in converting our internal, GPU-based graph representation to other popular network formats for usability.

Acknowledgements Funding was provided by Oak Ridge National Laboratory (Grant No. 3X012DCS).

References

1. (2010) Graph 500. http://www.graph500.org/
2. Alam M (2016) HPC-based parallel algorithms for generating random networks and some other network analysis problems. Ph.D. thesis, Virginia Tech
3. Alam M, Perumalla KS (2017) Gpu-based parallel algorithm for generating massive scale-free networks using the preferential attachment model. In: IEEE international conference on Big Data (Big Data), pp 3302–3311. https://doi.org/10.1109/BigData.2017.8258315
4. Alam M, Khan M, Marathe MV (2013) Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. In: International conference for high performance computing, networking, storage and analysis. https://doi.org/10.1145/2503210.2503291
5. Alam M, Khan M, Vullikanti A, Marathe M (2016) An efficient and scalable algorithmic method for generating large: Scale random graphs. In: Proceedings of the international conference for high performance computing, networking, storage and analysis. IEEE Press, Piscataway, NJ, USA, SC '16, pp 32:1–32:12. http://dl.acm.org/citation.cfm?id=3014904.3014947
6. Albert R, Jeong H, Barabási AL (2000) Error and attack tolerance of complex networks. Nature. https://doi.org/10.1038/35019019
7. Azadbakht K, Bezirgiannis N, de Boer FS, Aliakbary S (2016) A high-level and scalable approach for generating scale-free graphs using active objects. In: Proceedings of the annual ACM symposium on applied computing
8. Barabási AL, Albert R (1999) Emergence of scaling in random networks. Science. https://doi.org/10.1126/science.286.5439.509
9. Batagelj V, Brandes U (2005) Efficient generation of large random networks. Phys Rev E. https://doi.org/10.1103/PhysRevE.71.036113
10. Chakrabarti D, Zhan Y, Faloutsos C (2004) R-mat: A recursive model for graph mining. In: SIAM international conference on data mining, pp 442–446. https://doi.org/10.1137/1.9781611972740.43
11. Chassin DP, Posse C (2005) Evaluating North American electric grid reliability using the Barabási–Albert network model. Physica A. https://doi.org/10.1016/j.physa.2005.02.051
12. Dorogovtsev S, Mendes J (2002) Evolution of networks. In: Advances in physics, vol 51. https://doi.org/10.1080/00018730110112519

13. Dorogovtsev S, Mendes J, Samukhin A (2003) Principles of statistical mechanics of uncorrelated random networks. Nucl Phys B. https://doi.org/10.1016/S0550-3213(03),00504-2
14. Erdős P, Rényi A (1960) On the evolution of random graphs. Publications of the Mathematical Institute of the Hungarian Academy of Sciences, pp 17–61
15. Funke D, Lamm S, Sanders P, Schulz C, Strash D, von Looz M (2018) Communication-free massively distributed graph generation. In: 32nd IEEE international parallel & distributed processing symposium (IPDPS), to appear, preprint arXiv:1710.07565
16. Kleinberg J, Kumar R, Raghavan P, Rajagopalan S, Tomkins A (1999) The Web as a graph: measurements, models, and methods. In: Annual international conference on computing and combinatorics
17. Kumar R, Raghavan P, Rajagopalan S, Sivakumar D, Tomkins A, Upfal E (2000) Stochastic models for the web graph. In: Annual symposium on foundations of computer science, IEEE Comput. Soc. https://doi.org/10.1109/SFCS.2000.892065
18. Leist A, Hawick K (2011) Graph generation on GPUs using dynamic memory allocation. In: International conference on parallel and distributed processing techniques and applications
19. Leskovec J (2010) Kronecker graphs: an approach to modeling networks. J Mach Learn Res 11:985–1042
20. Leskovec J, Horvitz E (2008) Planetary-scale views on a large instant-messaging network. In: International conference on World Wide Web, ACM Press. https://doi.org/10.1145/1367497.1367620
21. Machta B, Machta J (2005) Parallel dynamics and computational complexity of network growth models. Phys Rev E 71(2):26704. https://doi.org/10.1103/PhysRevE.71.026704
22. Meyer U, Penschuck M (2016) Generating massive scale-free networks under resource constraints. In: Proceeding of the workshop on algorithm engineering and experiments (ALENEX)
23. Nobari S, Lu X, Karras P, Bressan S (2011) Fast random graph generation. In: International conference on extending database technology, p 331. https://doi.org/10.1145/1951365.1951406
24. Sanders P, Schulz C (2016) Scalable generation of scale-free graphs. Info Proc Lett
25. Watts DJ, Strogatz SH (1998) Collective dynamics of 'small-world' networks. Nature (6684). https://doi.org/10.1038/30918
26. Yoo A, Henderson K (2010) Parallel generation of massive scale-free graphs. arXiv CoRR