2012 ASE/IEEE International Conference on Social Computing and 2012 ASE/IEEE International Conference on Privacy, Security, Risk and Trust

Parallelizing Preferential Attachment Models for Generating Large-Scale Social Networks that Cannot Fit into Memory

Yi-Chen Lo, Cheng-Te Li, Shou-De Lin
Dept. Of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
anauma3@gmail.com, d98944005@csie.ntu.edu.tw, sdlin@csie.ntu.edu.tw

***Abstract*—Social network generation is an important problem in social network analysis. The goal is to produce artificial networks that preserve some real world properties of social networks. As one of most popular social network generation algorithms, the Barabási–Albert (BA) model is a method that can generate random social networks with power-law degree distribution. This paper discusses the situation of generating large-sized social network that cannot fit into the memory. We design a parallel framework to tackle this problem. The challenge lies in the fact that the *preferential attachment* mechanism used in the BA model has direct conflict with the concept of parallelism. To achieve the preferential attachment, during the generation processes the degree information of nodes needs to be known, which prohibits the parallelism that allows nodes to generate edges independently. To handle this issue, this paper proposes a method to generate the expected accumulated degree of vertices for the parallel BA model. We further propose several novel techniques to reduce the complexity of generating N vertices with P processes to $O(N\log N/P)$. We implement the model using MapReduce and the experiment results show that our model can produce billion-sized scale-free networks in minutes.**

Keywords: graph mining; large-scaled social networks; social network generation; Barabási–Albert model; preferential attachment; MapReduce

I. INTRODUCTION

The goal of social network generation is to generate artificial random graphs to model real world social networks properties. While the target size of social network graph data (usually in terms of the number of vertices in a network) to be generated scales up, the execution time to generate a social network grows drastically. This paper aims to efficiently generate realistic social networks whose sizes are too large to be loaded into the memory of a single machine.

Many social network generation algorithms have been proposed to fit real world social networks properties such as low average path length, high clustering coefficient, and power-law degree distribution. The Erdős-Rényi model [3], the Watts–Strogatz model [4], and the Barabási–Albert model [5] (abbreviated as ER, WS, and BA model, respectively) are the most famous social network generation algorithms that fulfill the above properties. All three models can produce networks with low average path length. The WS model produces networks with high clustering coefficient while the BA model generates networks with power-law degree distribution. To handle the limited memory issue in large-scale social network generation, we can divide the generation task of different sets of nodes to different processors in parallel. The ER and WS model can be easily implemented in parallel since the generation processes of vertexes are independent of each other. Unfortunately, such nice properties do not apply to the BA model. The BA model adds vertices with the *preferential attachment (PA)* mechanism, which states that vertices with higher degree values are more likely to be attached to the new vertices. That is, while adding a new vertex into an existing network, it is required to know the degree information of existing vertices to satisfy the PA property. In this sense, we are facing a dilemma while trying to parallelize a PA model. To add a node using PA, the degree information of all other nodes is required; however, the parallelization framework prohibits the spread of degree information. Because each vertex generates its edges in parallel, it is impossible to know the degree information of all existing nodes without engaging significant communication burden on the processors. Therefore, the original BA model can hardly be parallelized.

In order to parallelize a PA model, we propose a method to calculate the *expected accumulated degree* of each node independently, and show that such degree information can be utilized to perform PA. With an accurate estimation of the degree change over time for each node, it is possible to break the degree dependency among nodes to allow parallelism.

We first propose a naïve method to compute the expected accumulated degree which takes $O(N^2)$ time to add an edge (the space complexity is $O(N)$). A binary search method is then proposed to reduce it to $O(N\log N)$. We further decompose the degree-generation function into three components, and propose constant solutions for each of them. Eventually, adding a PA edge requires only $O(\log N)$ of computation time and the overall time complexity for generating the network is $O(m*(N/P)*logN)$. Moreover, our memory usage is reduced to $O(N/P)$ for each processor, which means we can generate a network with any size as long as there are sufficient processors for parallelization.

Although the proposed algorithm is applicable on different parallel computing frameworks such as MPI [12] and CUDA [15], we implemented our model using MapReduce[1], which is a distributed computing framework proposed by Google, as the distributed framework to generate social networks. MapReduce provides enhancement of efficiency (less execution time) and scalability in terms of network size (flexible memory usage).Our experiment shows that the algorithm can preserve the properties of power-law degree distribution while improving the efficiency and scalability. A billion-sized network can be created in minutes using relatively small amount of processors.

The main contributions of this paper are:

1. Strategically, we propose to use 'expected cumulative degree' of nodes to facilitate the parallelism of PA models when the target size of networks is too large to

978-0-7695-4848-7/12 $26.00 © 2012 IEEE
DOI 10.1109/SocialCom-PASSAT.2012.28

fit into a single processor's memory. Such idea, to our knowledge, has not yet been mentioned in the area of social network generation.

2. Tactically, our contributions are threefold. First, we decompose the computation of the expected cumulative degree into three sub-tasks, and propose a constant-time algorithm to solve two of them. Second, for the remaining task, we propose a constant-time approximation algorithm and prove the theoretical error bound of it. Third, we propose a binary search model for PA and exploiting caching mechanism to further reduce the time complexity.
3. Empirically, we implemented a competitor's model for comparison, and show that our model is more consistent in producing social networks that satisfy PA property. Furthermore, our model has much fewer parameters and therefore can be used without significant amount of efforts spending on parameter-tuning.

II. BACKGROUND AND RELATED WORK

In this section, we will first provide the background about MapReduce framework and the BA model. Then we will introduce related works in two categories: 1) existing MapReduce graph libraries, and 2) existing parallel algorithm to generate scale-free networks, which share a similar goal with our work.

A. MapReduce, Hadoop

MapReduce [1] is a distributed computing framework on computer clusters proposed by Google. MapReduce provides simple API of functional programming: user can execute a MapReduce job by only defining two functions: map function and reduce function. It also has the advantage that the underlying framework can automatically handle complicated distributed computing issues such as machine failures, load balancing, data distribution and replication. With MapReduce, users can run parallel jobs without tackling these difficulties.

In MapReduce, only one machine node is considered as the master, and the others are worker nodes. The master would assign map task or reduce task to the worker nodes. Input data are first split into pieces. The number of pieces equals to the number of worker nodes to be assigned with map task. Then worker nodes assigned with map task read in corresponding input split, parsing them into key/value pairs, and passing them to map function. Intermediate key/value pairs are written to local disk as outputs of map function. These pairs then are divided into partitions by the partition function which can be defined by user. The number of partitions is the same as number of reduce tasks. Each partition would be assigned to be remotely read by a corresponding worker node assigned with reduce task. Applying modulo operation by the number of reduce tasks on hash value of key is one common example of partition function. After a reduce worker read in all intermediate key/value pairs for its partition, the pairs are sorted by key and pairs with the same key are grouped together. Then keys and corresponding set of values are passed to the reduce function. The outputs of the reduce function from all reduce workers are appended as the final output. When all map tasks and reduce tasks are finished, the MapReduce job is completed.

Hadoop[2] is the open-source implementation of MapReduce from Apache. The methods we proposed in this work are implemented and tested on Hadoop.

B. Existing Graph Libraries in MapReduce

There are already many graph libraries in MapReduce released on the Internet. PEGASUS [6] is a state-of-the-art open source MapReduce library implementing many graph algorithms and proposing primitive functions based on matrix-vector multiplication for the algorithms. Lai's work [7] categorized graph mining algorithms, designed solutions respectively, and proposed many optimization techniques. Their experiments showed their approaches can be faster in execution time over PEGASUS. There are other MapReduce graph libraries such as Cloud9 [10]. Cloud9 is a MapReduce library implementing design patterns and algorithms introduced in the book Data-Intensive Text Processing [11] with MapReduce by Lin and Dyer, and being served as the teaching and research supporting tool of the book.

C. Barabási–Albert model (BA model)

Social network generation algorithms are designed to satisfy properties held by real world social networks such as short average path length, high clustering coefficient, and power-law degree distribution. A network having a property of power-law degree distribution is called a scale-free network. The ER model [3], the WS model [4], and the BA model [5] are frequently used social network generation algorithm. ER and WS model can be easily implemented in parallel while it is not the case for the BA model due to its *preferential attachment* (*PA*) mechanism.

The BA model is a model for generating random scale-free network. If the degree distribution, i.e. probability density function of degree corresponds to the form $p(x) = Cx^{\alpha}$, the graph is said to satisfy power-law degree distribution. In the BA model, the α value of degree distribution is proven to be -3.

The BA model adds vertices into the graph serially with the mechanism of PA. A new vertex is added to the graph one at a time till the graph grows to the target size. The probability of a new coming vertex forms a link with an existing vertex is proportional to degree of the existing vertex. It means new vertices prefer to link with high degree vertices. It is an intuitive criterion because in real world these high degree vertices stand for popular people in a social network or hot websites on the Internet.

Here is one simple example of how the BA model generates a graph. In Figure 1, the graph starts with m_0=3 fully connected vertices. New vertex 4 is added with m=2 edges. Since the degrees of the three vertices are the same, probability of selecting vertex1, 2, and 3 to connect the new vertex to are equally 1/3. Suppose vertex 2 and 3 are chosen to be attached with the new vertex 4.

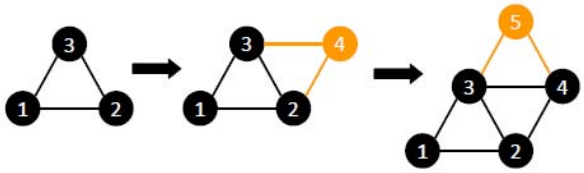

Figure 1. A example of the BA model generation

Next iteration, a new vertex 5 is added with $m = 2$ edges. Now the degree of vertex 1, 2, 3, and 4 is 2, 3, 3, and 2, respectively. The total degree is now 10 and the probability of each vertex being chosen are 2/10, 3/10, 3/10, 2/10 respectively. Vertices 2 and 3 have higher probability to be chosen than vertex 1 and 4. Here vertex 3 and 4 are chosen to be attached with the new vertex 5 as a result. Since m edges are added into the graph each iteration, the total degree increases by 2m each iteration. If we define the iteration t=0 as the initial state, the total degree of iteration t is $total(t) = 2mt + 2\,C_2^{m0}$ where the notation C_k^n stands for number of k-combinations of n elements.

One possible strategy to implement PA utilizes the accumulated degree. The accumulated degree of the ith vertex is the sum of all degrees of the vertexes that come before it (i.e. the sum of degree from v_1 to v_{i-1}). In the example above, the accumulated degrees of vertex 1, 2, 3, and 4 are 0, 2, 5, and 8 respectively, and the total degree is 10. To select a vertex to link to the new vertex, the model generates a random real number between 0 and the total degree (10), and establishes a link with the vertex whose accumulated degree is closest to but less than the random number. For instance, if the number generated is 2.7, then the newly added vertex will establish a link with vertex 2. If the next number generated as 7.13, then another edge is attached to vertex 3.

That says, with the accumulated degree of each vertex, we will be able to perform PA using a linear scan algorithm, without having to know the exact degree of each individual vertex.

D. *Existing Parallel Algorithms to Generate Scale-free Network*

The parallel version of R-MAT graph algorithm [13] in Parallel BGL[8] and the parallel BA algorithm by Yoo et al. [9] share the same goal of generating scale-free network in parallel. Below we will introduce these methods and discuss potential issues in their works.

In Parallel BGL [8] graph generators are implemented including ER model, WS model, and R-MAT graph [13] in MPI [12]. R-MAT algorithm generates configurable scale-free networks. To add an edge into an R-MAT graph, it recursively divides the adjacency matrix into four areas with user defined probability a, b, c, d till the area is left 1, i.e. pointing an element in the adjacency matrix, the edge is added with the corresponding index. This process loops till the target number of edges is satisfied[1]. To achieve this goal, it first assigns each vertex to one of the processes, and then in each process they add an edge if one of the vertex pair is assigned to the process, otherwise, they discard the edge. This design keeps the edges from being duplicated in multiple processes but leaves a problem of redundant iterations which generating discarded edges. As the number of processes increases, it would be more difficult for edges to hit corresponding processes, i.e. the overhead of redundant iterations would be more significant. Therefore their implementation is not well scalable.

Yoo et al. [8] designed two parallel graph generation algorithms to generate scale-free networks, the parallel BA model and parallel Kronecker model [14]. The parallel Kronecker algorithm has higher degree of parallelism than the parallel BA algorithm, but the structure of generated graph is heavily dependent on the initial seed graph, thus it is more difficult to configure desired properties. In the parallel BA algorithm, each vertex is assigned to a process and each process is assigned to one or more factions. A faction contains multiple processes and a process may share multiple factions with any other process. This algorithm is composed of two phases. In phase 1, each vertex in a process determines which remote processes to connect to, and send requests to the remote processes. The more factions a remote process shares with local process, the more edges the local process would link with it. Then in phase 2, each process receives the request of establishing edges from remote processes and determines which local vertices would be connected based on PA. The number of factions and which processes belong to which factions are left as degrees of freedom for users to determine. Since the edges added in phase 1 and 2 are all intra-faction, there is another parameter to establish some inter-faction edges with remote processes.

The main concern for this paper is that there are complicated parameters to tune in this system, including setting the number of factions, the variable size of each faction, the evolved factions for each processor, incorporating inter-faction edges. However in their paper they did not provide a suitable configuration that can produce graphs with power-law distribution. As will be shown in the experiment section, we re-implemented their parallel BA algorithm and tried several configurations, but found that power-law degree distribution property cannot be attained in most of the configurations.

III. PARALLEL BA MODEL GENERATION ALGORITHM

In this section we introduce our method to parallelize the BA model. We use the same notations as in Section2.C to describe parameters in the BA model: m_0 is the number of initial fully connected vertices, and m is the number of edges emerged from a vertex when it first joins the network. We define t_i as the i-th iteration and v_i as the single vertex joining the network in t_i. The total degree of the network at t_i is denoted as $total(t_i) = 2mi+2\,C_2^{m0}$

To parallelize a BA model task of generating N vertices with P processes, our goal is to assign N/P vertices to each of the processors in sequence. Figure 2 shows an example of generating a graph with 9 vertices divided into 3 partitions for 3 processors. Process 1, 2, and 3 are responsible of generating the first three vertices (v_1 to v_3), the next three vertices (v_4 to v_6), and the last three (v_7 to v_9), respectively, while our goal is to enable the parallelism of these three processors so they can

[1] Note that, the distributed implementation of R-MAT in Parallel BGL is not described in the corresponding paper. We obtained knowledge about their parallelization schema by tracing their open source code.

create edges independently. v_{xy} represents vertex x, v_x, at time y, t_y. Note that the newly added vertex in each iteration is marked on the right-most position of each row. The original BA model relies on the concept of PA, which requires the later vertex (e.g. vertex 9) to obtain the degree information of all other vertices (e.g. vertex 1~8) before determining which vertices it should attach to (i.e. the higher degree the better chance). In general, the PA property conflicts with the parallelism framework. For instance, to perform PA for v_6 when it first joins the network, we need to know the degree of vertices v_1 *to* v_5. Unfortunately such information is not available because those vertices themselves are being processed in parallel. Even if some vertices finish the generation process earlier, their degree information can hardly be passed to the other processors due to high communication cost between processors during parallelism.

A. The parallel BA Approach Overview

In this section we propose a method that allows independency among link generation of each vertex. Our goal is to use only $O(m*N/P)$ of memory and adding an edge with $O(\log N)$of time

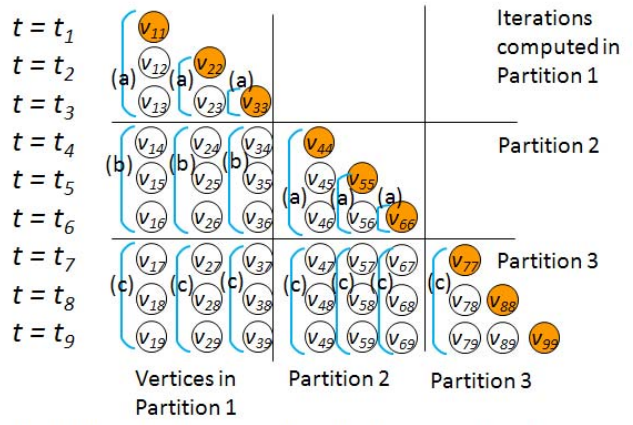

Figure 2. Generating a graph of 9 vertices divided into 3 partitions in the BA model, the colored vertices are new vertices

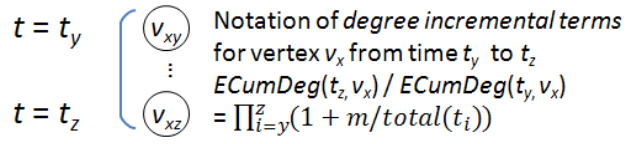

Figure 3. Explanation of graphic notation in Figure 2.

As mentioned in section 2.C, PA depends on knowing the the accumulated degree of each vertex is known. Unfortunately, similar to the degree information, the accumulated degree of vertices cannot be obtained when PA is conducted in parallel.

Given the accumulated degree is unknown, we propose to generate the "expected accumulated degree" for each vertex in each processor. The expected accumulated degree for a vertex v_k at iteration t_j is denoted as $ECumDeg(t_j, v_k)$.

In Section 3.B we will prove that

$$ECumDeg(t_j, v_k) = total(t_j) * \prod_{i=k}^{j} (1+m/total(t_i)) \quad (1)$$

With this equation we can calculate the expected accumulated degree of any vertex at any iteration in any graph partition, independent of degree information from other partitions.

In the example shown in figure 2, to introduce v_9 into the network, we can naively apply (1) to calculate accumulated degree of v_1 to v_8 at iteration t_9, and then for each new edge, we have to generate a random degree and run linear scan to find the vertex to attach it to. Unfortunately, to compute $ECumDeg(t_9, v_1)$, the term $(1+m/total(t_i))$ would have to be multiplied for 8 times (each with different t_i), and similarly, 7 times for $ECumDeg(t_9, v_2)$, and so on. That says, in order to add a link, we need to calculate the estimated accumulated degree for all N vertices requires the time complexity $O(N^2)$ and space complexity $O(N)$. It becomes computationally intractable when N becomes larger

Trying to solve the efficiency problem when identifying the vertex for PA, we found that it is not necessary to generate the *ECumDeg* for every vertex. Instead of applying linear scan to find the vertex to attach a new node to, we use binary search. Take the example in Figure 2, suppose the vertex with accumulated degree closest to but smaller than a random number R is v_2, we can first generate $ECumDeg(t_9, v_4)$ and check whether R is larger or less than it. If $R < ECumDeg(t_9, v_4)$, we can than generate $ECumDeg(t_9, v_2)$ for comparison, and the process continues until the target vertex is found. The target vertex can be found in O(log$N$) steps. By applying binary search, we can reduce the time complexity of PA from $O(N^2)$ to $O(NlogN)$. The space complexity is $O(1)$ since we compute $ECumDeg(t_j, v_k)$ on the fly and do not store it.

Next, we further reduce the complexity of PA by categorizing the multiplications into three types: *initial product*, *internal product, and external product*, and cache the preprocessed values. Figure 2 presents a diagram showing how each vertex's degree changes when new vertices are being added. As will be shown in section 3.C, the *internal* and *external products* can be generated in constant time using cache technique. In section 3.D, we further propose an approximating algorithm that reduces the *initial product* to $O(1)$. Eventually, we can significantly reduce the complexity from $O(N^2)$ to $O(\log N)$. Below we provide detail description and formulae about the proposed methods.

B. Expected Value of Accumulated Degree

We propose to estimate the accumulated degree of nodes for PA because it can be generated without knowing the attachment outcome from other vertices.

$$ExpDeg(t_j,v_k) = ExpDeg(t_k,v_k) \prod_{i=k}^{j-1} (1+m/total(t_i)) \quad (2)$$

(2) is the recursive formula to express the expected degree for a specific vertex with id v_k (i.e. the k-th joined vertex) in a specific iteration t_j. The total degree of at j-th iteration in the BA model is $total(t_j)= 2mj+2\,C_2^{m0}$, since the total degree is

$2C_2^{m0}$ at the initial state ($t = 0$) and every new vertex comes with degree of m. A vertex's degree gain on average m multiplies the proportion of the vertex's to the current total degree. The expected degree of a vertex k at iteration j is notated as $ExpDeg(t_j, v_k)$, and we can construct the recursive equation as:

$$\begin{aligned} & ExpDeg(t_j, v_k) \\ &= ExpDeg(t_{j-1}, v_k) + \frac{m * ExpDeg(t_{j-1}, v_k)}{total(t_{j-1})} \\ &= ExpDeg(t_{j-1}, v_k)\left(1 + m / total(t_{j-1})\right) \\ &= ExpDeg(t_k, v_k)\prod_{i=k}^{j-1}\left(1 + m / total(t_i)\right) \end{aligned}$$

where $ExpDeg(t_k, v_k) = m_0 - 1$ if $k = 0$, or m otherwise.

As we have mentioned previously, PA can be achieved by the accumulated degree of vertices as well, and the expected accumulated degree of vertex v_k at iteration t_j is defined as

$$ECumDeg(t_j, v_k) = \sum_{i=1}^{k} ExpDeg(t_j, v_k) \quad (3)$$

The reason we prefer using accumulated degree than the original degree of vertices is that then we do not really need to generate the expected degree of each individual vertex. We can use the equation we prove in *Lemma1* to generate the accumulated degree.

Lemma1. $ECumDeg(t_j, v_k) = total(t_k) * \prod_{i=k}^{j-1} (1+m/total(t_i))$

Proof.

From (2)(3),

$$\begin{aligned} & ECumDeg(t_j, v_k) \\ &= \sum_{i=1}^{k} ExpDeg(t_j, v_i) \\ &= \sum_{i=1}^{k} ExpDeg(t_k, v_i)\prod_{i=k}^{j-1}\left(1 + m / total(t_i)\right) \\ &= total(t_k) * \prod_{i=k}^{j-1}\left(1 + m / total(t_i)\right) \end{aligned}$$

Note that Lemma 1 tells us that the *ECumDeg* values can be generated using only the total vertex information in each iteration. In the extreme case, to generate $ECumDeg(t_j, v_1)$, we will need to execute multiplication j-1 times. That says, when the total number of vertices N grows, the complexity of generating the accumulated degree for the 1st vertex approaches $O(N)$. From here on we define a *query* as the computation of the accumulated degree of a node. Therefore, for an iteration i, we need to query i-1 times (from v_1 to v_{i-1}) in order to perform one PA attachment, while performing binary search allows us to reduce the complexity from $O(N)$ to $O(\log N)$. That is, overall it takes $O(N \log N)$ time to create a link through PA in our parallel framework. In the next section we will further discuss how to reduce the complexity to $O(\log N)$.

C. Decomposing Expected Accumulated degree

To avoid heavy computation from calculating the *expected accumulated degree*, we exploited two ideas. The first idea lies in reusing the result of computation. For example, we may query the accumulated degree of the same vertex in different iteration, $ECumDeg(t_j, v_k)$ and $ECumDeg(t_{j+d}, v_k)$. Since both requires the value of $\prod_{i=1}^{j} (1+m/total(t_i))$, we can cache such intermediate values to avoid re-calculating every term in every query.

The second idea is to exploit the parallelism to calculate the product of the term $1+m/total(t_i)$. Here we use Figure 2 as an example. Assuming the vertex v_9 is just joining the network and we need to establish its link. We need to generate $ECumDeg(t_9, v_1)$, which is represented as v_{19} in Figure 2.

We decompose $\prod_{i=1}^{k} (1+m/total(t_i))$ into three parts:

1) *Initial product*: the part of product in the partition where the queried vertex is generated. In this example, $\prod_{i=1}^{2} (1+m/total(t_i))$.

2) *Internal product*: the part of product of terms within the currently generated partition. In this example, $\prod_{i=6}^{8} (1+m/total(t_i))$.

3) *External product*: the part of product from other partitions which is multiplied across the whole partition. In this example, $\prod_{i=3}^{5} (1+m/total(t_i))$.

In Figure 2 we marked the three products in the example of querying expected acculumuted degree at t_9.

Internal product can be easily calculated and cached while going through generation iterations in one partition. Note that the *internal product* terms to be multiplied in each process is identical (e.g. for process 3, $ECumDeg(t_9, v_1) / ECumDeg(t_6, v_1) = ECumDeg(t_9, v_2) / ECumDeg(t_6, v_2) = \ldots = ECumDeg(t_9, v_7) / ECumDeg(t_6, v_7) = \prod_{i=7}^{9}(1 + m/total(t_i))$, therefore we can calculate it once and reuse it using cache. Eveutally it takes only $O(1)$ to generate this number.

The *external product* represents the multiplications associated with vertices of other partitions. For example, for v_{19}, the multiplication of v_{14} to v_{16} is considered as an external product. In general, for partitions that join later, the *external product* represents more terms. Take Figure 4 for example, for partition 5, its *external product* equals to *E2*E3*E4* and for partition 4 it is *E3*E4*. Here we have realized that there are total of $P^2/2$ such E_k blocks. If we can pre-calculate in parallel and store them in the cache of each partition, the *external product* values can be generated in constant time as well. Note for each query, every block except the top one (i.e. initial produce) and bottom one (internal block) are considered as part of the external block.

The time complexity to cache *external product* is $O(P^2)$ while space complexity is $O(P)$. With this cache, we can obtain *external product* in $O(1)$ while being queried.

Unlike *internal* and *external products*, *initial product* for every vertex being queried in the partition has different product value. Neither caching *initial product* for every vertex, nor calculating *initial product* for every query is tractable . Therefore, we choose to apply approximation to *initial products* which will be shown in the next section.

Our approximation generates the *initial products* in $O(1)$ time. In a nutshell, by using caching and approximation techniques, we are allowed to reduce the time complexity of each query to $O(1)$.

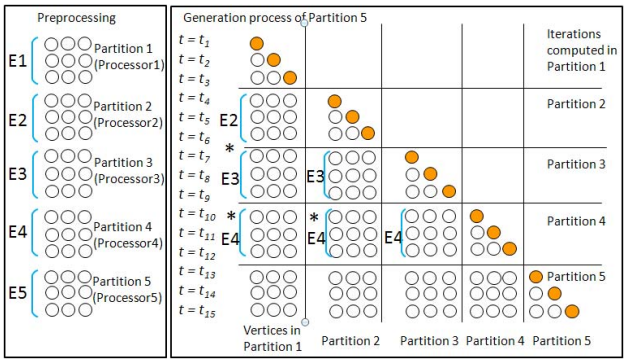

Figure 4. Computing *external product* in process of *partition 5*

D. Approximation and Partial Caching of Initial Product

For *initial product*, every vertex in the partition results in different product value. The first vertex of each partition receives N/P products, and N/P-1 for the next vertex, finally the value for the last vertex of each partition is one. If we cache *initial product* of all vertices in the same partition, the computation time is only linear of the partition size (number of vertices of the whole graph divides by number of partitions). However, to cache the *initial product*, we have to store the *initial product* of all vertices in each partition, which takes $O(N)$, which is not very realistic as we assume each partition has only $O(N/P)$ caching space.

Therefore we propose a fast approximation for the value of *initial product.* The product of terms of $(1+m/total(t_i))$ starts from t_j to t_{j+d}, respectively, representing the start and end iteration of the *initial product*.

$$\prod_{i=j}^{j+d}\left(1+m/total\left(t_i\right)\right)\approx\sqrt{1+d/(t_i+C_2^{m0}/m)}$$

The following is the process we obtain such approximation:

$$\prod_{i=j}^{j+d}\left(1+m/total(t_i)\right)=\prod_{i=j}^{j+d}\left(1+m/(2mi+2C_2^{m_0})\right)$$

$$=\frac{2(j+C_2^{m_0}/m)+1}{2(j+C_2^{m_0}/m)}*\ldots*\frac{2(j+d+C_2^{m_0}/m)+1}{2(j+d+C_2^{m_0}/m)}$$

$$=\frac{2t_{start}+1}{2t_{start}}*\ldots*\frac{2t_{end}+1}{2t_{end}} \quad (4)$$

$$=\frac{3/2*5/4*7/6*\ldots*(2t_{end}+1)/2t_{end}}{3/2*5/4*7/6*\ldots*2t_{start}/(2t_{start}-1)} \quad (5)$$

$$\approx\frac{\sqrt{2/\pi}\sqrt{2t_{end}}}{\sqrt{2/\pi}\sqrt{2t_{start}}} \quad (6)$$

$$=\sqrt{t_{end}/t_{start}}=\sqrt{1+d/(j+C_2^{m_0}/m)}$$

In (4) we apply the following substitution to make it simple

$$t_{start}=j+C_2^{m_0}/m$$

$$t_{end}=j+d+C_2^{m_0}/m$$

In (5) we multiply the term below which is equal to 1

$$\frac{3/2*5/4*7/6*\ldots*2t_{start}/(2t_{start}-1)}{3/2*5/4*7/6*\ldots*2t_{start}/(2t_{start}-1)}$$

Then (6) we apply the following equation of *Wallis product* [16] for π.

This equation is true if j is infinite, or the error is known to be $O(1/j)$ from *Stirling's approximation*. [17]

$$\prod_{i=1}^{j}\left(\frac{2i}{2i-1}\cdot\frac{2i}{2i+1}\right)=\frac{2}{1}\frac{2}{3}\frac{4}{3}\frac{4}{5}\frac{6}{5}\frac{6}{7}\frac{8}{7}\frac{8}{9}\cdots=\frac{\pi}{2}$$

The following are detailed induction for *Wallis product* substitution in (6).

$$\frac{3}{2}\frac{5}{4}\frac{7}{6}*\ldots*\frac{(2t_{end}+1)}{2t_{end}}$$

$$=\sqrt{\frac{1}{2}\frac{3}{2}\frac{3}{4}\frac{5}{4}\frac{5}{6}\frac{7}{6}*\ldots\frac{(2t_{end}-1)}{2t_{end}}*\frac{(2t_{end}+1)}{2t_{end}}}$$

$$=\sqrt{1/\prod_{k=1}^{t_{end}}\left(\frac{2k}{2k-1}\cdot\frac{2k}{2k+1}\right)}\sqrt{(2t_{end}+1)}$$

$$\approx\sqrt{2/\pi}\sqrt{2t_{end}} \quad (7)$$

Similarly,

$$\frac{3}{2}\frac{5}{4}\frac{7}{6}*\ldots*\frac{2t_{start}}{(2t_{start}-1)}\approx\sqrt{2/\pi}\sqrt{2t_{start}} \quad (8)$$

Finally we obtained the approximation.

$$\prod_{i=j}^{j+d}\left(1+m/total\left(t_i\right)\right)\approx\sqrt{1+d/(t_i+C_2^{m0}/m)}$$

To further discuss the error from this approximation, by *Stirling's approximation*, result of (7) becomes

$$\sqrt{(2/\pi)(1/1+O(1/(j+d)))}\sqrt{2t_{end}}$$

and (8) becomes

$$\sqrt{(2/\pi)(1/1+O(1/j))}\sqrt{2t_{start}}$$

So we consider the error in (6)

$$\frac{\sqrt{(2/\pi)(1+O(1/j))}\sqrt{2t_{end}}}{\sqrt{(2/\pi)(1+O(1/(j+d)))}\sqrt{2t_{start}}}$$

$$=\sqrt{\frac{t_{end}}{t_{start}}}\sqrt{O\left(\frac{1+1/j}{1+1/(j+d)}\right)}$$

$$=\sqrt{\frac{t_{end}}{t_{start}}}\sqrt{O\left(\frac{j+d+(j+d)/j}{j+d+1}\right)}$$

$$=\sqrt{t_{end}/t_{start}}\sqrt{O\left(1+(d/j)/(j+d+1)\right)}$$

$$=\sqrt{t_{end}/t_{start}}\left(1+O\left(1+d/(j^2+jd)\right)\right)^{1/2}$$

In this approximation we obtained an error term of $(1+O(d/(j^2+jd)))^{1/2}$. After the second partition, $d \le j$ sustains since maximal value of d is N/P, the size of one partition and from the second partition, j is larger than the size of one parti-

tion. So the error term can be bounded as $(1+O(1/j))^{1/2}$ or $(1+O(P/N))^{1/2}$ for partitions after the second partition.

However, this approximation does not work well in the beginning partitions. So we calculate and cache the exact *initial product* value for beginning partitions. We apply approximation only to the later partitions. With this approximation, partitions that we have to cache the *initial product* value reduce from all partitions to only the beginning partitions. We leave the number of partitions to cache as a parameter in the system. As a fact, in our experiment the approximation is almost precise enough to cache only the first partition so that the power-law degree distribution is reserved.

Since we can calculate accumulated degree by composing *initial product*, *internal product*, and *external product*, performing PA for each new coming vertex can be parallelized. We implement our algorithm in MapReduce.

E. MapReduce Algorithms

Table 1 shows the symbols and functions we use in our algorithms. Table 2 shows the pseudo code of our algorithm and table 3 presents the complexity analysis. We design the algorithm into one MapReduce job. In both of *Map* and *Reduce* phase, we use a single input number as the process id. Note that in our example in Figure 2, we didn't describe the details to handle initial vertices in the BA model for simplicity, here we will include them.

In *Map* phase, each processor calculates the *external product* belonging to its partition and send the result as a *key-value* pair to each process. The *key* is target process id, and the *value* contains source process id and the partial *external product*.

In *Reduce* phase, each processor generates vertices with id it is responsible for, performing PA by calculating expected accumulated degree to build edges. Since in the first partition (*pid=0*) there are m_0 initial vertices, we handle it as special cases in line 4, 13, 15-18. At the beginning of *Reduce* phase, in line 1,2 it receives partial *external products* from all other processors and calculate *external products* in line 5-8. Then in line 9-13 *initial product* is cached according to *C*, the parameter of number of partitions set to cache. In line 19-28, for each iteration *t,* one new vertex is joined and attaches edges with *m* existing vertices. Note that we use the iteration number to assign id to a vertex because only one vertex is added in every iteration. In line 20 the *internal product* is accumulated with the progress of iteration. In line 21-25, for each edges to be added, a random number *R* is generated to perform PA. In line 23-25, binary search is conducted to find which vertex in the known partition should be attached. In line 26 we handle the case of choosing initial vertices with equal probability.

In the *binarySearch* function, *from* and *to* are the candidate vertices with minimal and maximal vertex id. Line 1 shows the condition of continuing searching when there are at least two candidates. Line 5-7 handles the case when *internal product* overlaps with *initial product.* We set *initial product* to 1 and remove not used part in the latest *internal product*. In line 8-9, the queried vertex is in a partition whose *initial products* are cached. In line 10-13 it handles the normal case that we assign latest *internal product* and approximate the *initial products* with the formula in Section 3.D. In line 14 we calculate *ECumDeg.* In line 15-18 we compare *ECumDeg* with the target accumulated degree and decide the interval to search for next loop. Finally we take *from* as the return value which presents the vertex id chosen to be attached with the new generated id.

F. Complexity Analysis

In Table 3 the complexity is presented in the aspect of one processor (a *mapper* or a *reducer*). In *Map phase,* the input size is $O(1)$ since for a *mapper* it reads in a single number as the process id. The time and space complexity in *Map* is the number of vertices in a partition, $O(N/P)$. The *Map* output size and *Reduce* input size is $O(P)$ *partial external product* to send to or to receive from every other partition.

In *Reduce* phase, in line 5-8, it spends $O(P^2)$ time to calculate *external products* and needs $O(P)$ space to cache them. In line 9-13 it spends $O(C*(N/P))$ time and space to calculate and cache the *initial products* for first *C* partitions.

Note that in our analysis in Section 3.D, the error can be bounded $(1+O(P/N))^{1/2}$ if we set C=1. In line 19-28, it computes $O(m*N/P)$ edges, and thus using space of $O(m*N/P)$ to store the output edges. For each edge, it spends time of $O(P)$ to determine which partition the target vertex falls in and spend time of $O(\log(N/P))$ to perform binary search in one interval partition. As to the *Reduce* output size, each process output *m* edges for each of the $O(N/P)$ vertices.

TABLE I. SYMBOL AND FUNCTION

Symbol	Definition
m	the number of edges to be added for each iteration
m_0	the number of vertices in initial state
C	the number of partitions to cache exact *initial product*
N	the number of vertices to be generated
P	the number of processes (the number of graph partitions)
partSize	*N*/*P,* the number of vertices in one graph partition
pid	the partition id
partialExt	the *partial external product* from single partition to compute *external product*
partialExt_vec	vector of *partialExt*
internal_vec	vector of *internal product*
external_vec	vector of *external product*
initCache	vector of exact *initial product* cache
external	*external product*
internal	*internal product*
initial	*initial product*
R	*random*() * *total*(*t*-1), accumulated degree of the vertex to be linked with the new vertex
inPart	the partition where the queried vertex locates in
from	the lower bound of search interval
to	the upper bound of search interval
t_end	the last iteration in the queried partition
random()	random generate a number within [0, 1]
sqrt(*t*)	square root of value *t*
total(*t*)	$2mt+2C(m_0,2)$, total degree of the graph in time *t*
local(*t*)	(*t*-1) mod *partSize* + 1, the serial number of vertex *t* in the partition. $1 \le local(t) \le partSize$
real(*t*)	$t + m_0 - 1$, the real vertex id of vertex *t* after take initial vertices into consider

TABLE II. THE PARALLEL BA ALGORITHM

The Parallel BA Algorithm

```
Map      Input: single number represents pid
         Output: (key, value)=(p, [q, partialExt])
Reduce   Input: (key, value)=(p, [q, partialExt])
         Output: output graph in edges

Map
1    pid ← input value
2    partialExt ← 1.0
3    for t = pid*partSize+1 to (pid+1)*partSize
4       partialExt ← partialExt * ( 1+ m/total(t-1) )
5    for p = 0 to P-1
6       output(p, [pid, partialExt]);

Reduce
1    For each values pair [q, partialExt] gathered by key p
2       partialExt_vec[q] ←partialExt
3       pid ← p
4    if( pid = 0) output initial edges            // initial vertices
5    for i = pid to 0                    // calculate external product
6      external[i] ← 1.0
7      for j = pid to i+1
8        external_vec[i] ← external_vec[i] * partialExt_vec[j] ;
9    for p = 0 to C                          // caching initial
     product
10      initCache[(p+1)*partSize] = 1.0
11      for t = (p+1)*partSize -1 to p*partSize +1
12         initCache[t] = initCache[t+1] *( 1+ m/total(t) )
13      if(p = 0) initCache[0] ← initCache[1] * (1 + m / total(0))
14
15   if(pid = 0)
16      internal_vec[local(1)] ← 1 + m / total(0)
17   else
18      internal_vec[local(1)] ← 1.0
19   for t = pid*partSize+1 to (pid+1)*partSize // generate vertices
20      internal_vec[local(t)] ← internal_vec[local(t-1)] * (1 +
     m/total(t-1))
     //update internal product
21         for i = 0 to m-1
22            R = random() * total(t-1)   //Get random R accumulat-
     ed degree
23            from ← 1
24            to ← t-1
25            v ← binarySearch(R, from, to, external_vec, inter-
     nal_vec, initialCache, t)
26         if(v=0) v ← randomly assign a value from 0 to m-1
27         else v ← real(v)
28         output([real(t), v] );

binarySearch(R, from, to, external_vec[P], internal_vec[partSize],
initialCache[partSize*C], t )
1    while(to – from ≥ 1)
2       mid ← (from+to)/2
3       inPart ← mid mod partSize
4       external ←external_vec[inPart]
5       if(inPart = pid)
6          internal ← inter-
     nal_vec[local(t)]/internal_vec[local(mid)];
7          initial ← 1.0;
8       else if(inPart < C)
9          initial ← initialCache[mid]      // assign cached initial
     product
10      else
11         internal ← internal_vec[local(t)]
12         t_end ← (inPart+1) * partSize
13         initial ← sqrt( 1+ (t_end – mid)/(t+C(m0, 2)/m) )
14      ECumDeg ← total(mid)*initial*external*internal
15      if(ECumDeg ≥ R )
16         from ← mid
17      else
18         to ← mid - 1
19   end while
20   return from
```

TABLE III. TIME AND SPACE COMPLEXITY

Stage	Time Complexity	Space complexity
Map Input	$O(1)$	$O(1)$
Map()	$O(N/P)$	$O(N/P)$
Map Output/ Reduce Input	$O(P)$	$O(P)$
Reduce()	$O((N/P)*(m*log(N)+C) + P^2)$	$O((N/P)*(C+m) + P)$
Reduce Output	$O(m*N/P)$	$O(m*N/P)$

IV. EXPERIMENTS

The goal of this experiment is to examine whether our model satisfies the power-law distribution. We evaluate the scalability and efficiency on our model. Our experiment runs on a cluster composed of 16 machines of total storage 4.7 TB and each machine has 8 processors of Intel(R) Xeon(R) CPU X5550 @ 2.67GHzs and 32 GB of memory. Each computing vertex executes 8 map/reduce tasks for 8 partitions and each task using 2GB of memory. In the following experiments, we set the number of initial vertices, $\boldsymbol{m_0} = 3$, the number of edges to add for each iteration, $\boldsymbol{m} = 5$, and the number of partition to cache for *initial product* as 1.

A. PROPERTY OF POWER-LAW DEGREE DISTRIBUTION

We check the power law property of the generated graph with 32 million vertices which is separated into 128 partitions.

We use the maximum likelihood estimation to fit the curve of degree distribution of the generated graph, where degree distribution is expected to correspond the form of $p(x) = Cx^{\alpha}$ with α being close to 3. As to test the closeness of fit, we choose symmetric KL-divergence, which indicates better fit with lower value. In Figure 6, the curve fit is observed corresponding to the power-law distribution form of $p(x) = Cx^{\alpha}$ with α value -3.072330 which is close to -3 with symmetric KL-divergence 0.096542. We generated 10 graphs by our algorithm and the mean and variance of α are -3.072382 and $2.5867*10^{-7}$ and the mean and variance of symmetric KL-divergence are 0.095940 and $1.2997*10^{-8}$. This results shows that graph generated by our BA model algorithm consistently reserves the property of power-law degree distribution in the BA model.

We also evaluate the power-law degree distribution property of graphs generated with Yoo's parallel BA algorithm [9] using the same graph size and number of partitions. Our experiment confirms the conjecture mentioned in Section 2 that

due to large set of parameters in their model, the power-law property can be very sensitive to those parameters and not very stable. Unfortunately the paper did not mention an ideal setup for the parameters, so we have tried several.

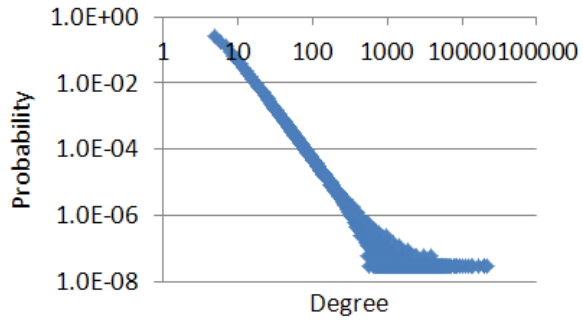

Figure 5. The degree distribution of the graph generated by our algorithm

We tried combinations of three kinds of parameters:

1) *Faction distribution:* the frequency distribution of processes in factions. We tried linear distribution, normal distribution, and exponential distribution.

2) *Number of Factions*

3) *Inter-faction probability*: the probability of establishing an inter-faction edge.

The parameter sets we tried along with the result α value and symmetric KL-divergence are listed in Table 4. The best fit configuration we tried is to set the number of factions as 32, probability of inter-faction edges as 0.2, and the frequency distribution of processes in factions as exponential distribution. The degree distribution of the generated graph is shown in Figure 6. It has α value -3.169821 and symmetric KL-divergence 0.162165. In comparison, our algorithm still fits better. Furthermore, in their framework the higher-frequency process would receive more messages which cause more memory space usage and imbalanced computation load.

TABLE IV. THE PARAMETERS WE TRIED FOR YOO'S ALGORITHM AND EVALUATION

Faction distribution	Number of Factions	*Inter-faction probability*	α value	Symmetric KL Divergence
Linear	12	0.2	2.595655	0.862763
Linear	12	0.4	2.597332	0.851255
Linear	32	0.2	2.640092	0.543105
Linear	32	0.4	2.658891	0.509376
Normal	12	0.2	2.904816	0.744332
Normal	12	0.4	2.858247	0.740452
Normal	32	0.2	3.695472	0.329612
Normal	32	0.4	5.011678	0.426804
Exponential	12	0.2	3.012600	0.401900
Exponential	12	0.4	2.993955	0.385739
Exponential	32	0.2	3.169821	0.162166
Exponential	32	0.4	3.456464	0.183178

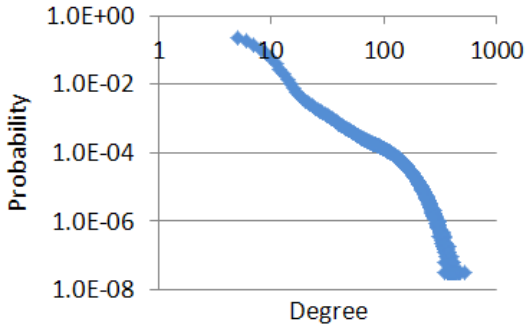

Figure 6. The degree distribution of the graph generated by Yoo's algorithm

The parameter setup in our algorithm is very straightforward. Besides the original parameters of the BA model, we only have to determine the number of partitions to cache the exact value of *initial product.* Moreover, our algorithm has better load balance because every partition generates the same number of vertices and edges.

B. TESTING SCALABILITY OF THE PARALLEL BA MODEL

To test the scalability and efficiency of our parallel algorithm for the BA model, we experiment on how it scales up with the graph size and number of computing nodes.

In the experiment of scaling up graph size, we fix the number of computing nodes at 16 and scale up the size of the graph to be generated to observe variation of execution time. As shown in Figure 7, the execution time has linear relationship with the graph size. The maximum graph was generated in only 1149 seconds has 1,792,000,000 vertices and 8,960,000,000 edges with data size 177GB.

In the experiment of scaling up computing nodes, we fix the graph size to generate at 128 million vertices and increase computing nodes to test speedup and efficiency of our algorithm. Speedup using P processes is $S_P=T_1/T_P$, where T_1 is runtime using single process and T_P is runtime using P processes. Efficiency using P processes is defined as $E_P= S_P/P$, where E_P =1 is the ideal efficiency that there is no overhead during parallelization. Figure 8, 9, 10 shows the experiment result of runtime, speedup, and efficiency when we using 16 computing nodes, the speedup is about 9.6 and efficiency is about 0.6. The results show that our algorithm has good scalability and efficiency.

There is no big difference between runtime of our algorithm and Yoo's algorithm since in both algorithms writing output data into the distributed file system dominates the runtime.

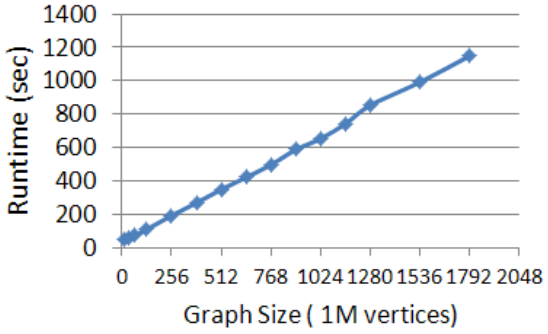

Figure 7. Experiment for runtime with 16 computing nodes when increasing graph size

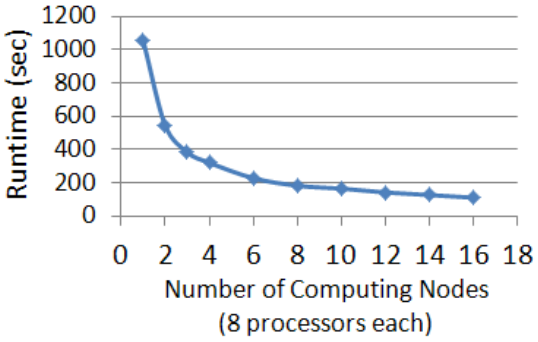

Figure 8. Experiments for runtime generating 128 million vertices to test scalability

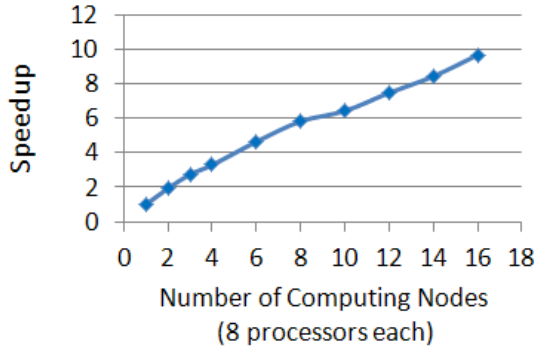

Figure 9. Speedup of generating 128 million vertices

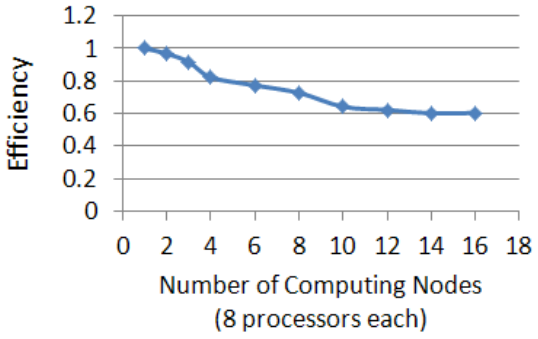

Figure 10. Efficiency of generating 128 million vertices

V. Concluding Remarks

To parallelize the BA model, we proposed an idea of 'expected accumulated degree' for preferential attachment. We further exploit several novel ideas to further reduce the complexity. We can then parallelize the process in MapReduce and create billion-size networks that satisfy power-law distribution in minutes.

We believe some of our proposed ideas such as *approximating the expected degree value*s can be applied to other parallelism tasks in social network, which becomes our future work.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *OSDI*, 2004.

[2] "Hadoop information," http://hadoop.apache.org/

[3] P. Erdo˝s and A. Re´nyi, Bull. Inst. Int. Stat. **38**, 343 (1961).

[4] D. Watts and S. Strogatz (1998): Collective dynamics of small-world networks. *Nature, 363*:202–204.

[5] R. Albert; A.-L. Barabási (2002). "Statistical mechanics of complex networks". *Reviews of Modern Physics* **74**: 47–97.

[6] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. Data Mining, IEEE International Conference on, 0:229–238, 2009.

[7] Hung-Che Lai, Exploiting and Evaluating MapReduce for Large-scale Graph Mining

[8] Douglas Gregor, Andrew Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In Parallel Object-Oriented Scientific Computing (POOSC), July 2005

[9] Andy Yoo, Keith W. Henderson. Parallel Generation of Massive Scale-Free Graphs. CoRR, 2010.

[10] Cloud9: A MapReduce Library for Hadoop, http://www.umiacs.umd.edu/~jimmylin/cloud9/docs/content/patterns.html

[11] Jimmy Lin and Chris Dyer, "Data-Intensive Text Processing with MapReduce"

[12] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, Steven Huss-Lederman (1995) *MPI: The Complete Reference*

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: Arecursive model for graph mining. In *SDM*, 2004.

[14] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Mathematically tractablegraph generation and evolution, using kronecker multiplication. In Proceedings of European Conference on Principles and Practice of Knowledge Discovery in Databases, 2005

[15] NVIDIA. 2007. CUDA Technology, http://www.nvidia.com/CUDA.

[16] Sondow, Jonathan and Weisstein, Eric W. "Wallis Formula." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/WallisFormula.html

[17] Weisstein, Eric W. "Stirling's Approximation." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/StirlingsApproximation.html