Mining and Generating Large-Scaled Social Networks via MapReduce

Yi-Chen Lo*[1], Hung-Che Lai*[2], Cheng-Te Li[#3], Shou-De Lin*[#4]

* Department of Computer Science and Information Engineering,
[#] Graduate Institute of Networking and Multimedia,
National Taiwan University, Taipei 106, Taiwan
{[1]d00922006, [3]d98944005, [4]sdlin}@csie.ntu.edu.tw, [2]jimmy.lai@oi-sys.com

Abstract

The computational efficiency is usually a concern when dealing with large-scale social network mining tasks containing billions of entities. Cloud computing is widely regarded as a feasible solution to this problem. In this work, we present an open source graph mining library called the MapReduce Graph Mining Framework (MGMF) to be a robust and efficient MapReduce-based graph mining tool. We start from dividing graph mining algorithms into several categories and design a MapReduce framework for algorithms in each category. The experimental results show that MGMF is 3 to 20 times more efficient than PEGASUS, a state-of-the-art library for graph mining on MapReduce. Moreover, it provides broader coverage of a variety of graph mining algorithms. Furthermore, we designed a model to generate large-scale social networks capturing the power-law degree distribution property by parallelizing the mechanism of preferential attachment so that it is possible to produce billion-sized scale-free network in minutes. Our implemented open-source library can be downloaded from http://mslab.csie.ntu.edu.tw/~noahsark/MGMF/

Keywords: *Graph mining, large-scaled social networks, MapReduce.*

1. INTRODUCTION

Researchers have spent decades tackling a variety of social network mining tasks [1], such as generating graphs that satisfy real properties of social networks (e.g., power law distribution [27]), estimating centralities (e.g., PageRank [2], random walk with restart [3], Betweenness Centrality [4]), counting triangles, and finding connected components [1]. In recent years, online social network services such as Facebook and Twitter are becoming increasingly popular and have created huge amount of social network data, and therefore creates urgent need for upgrading the existing mining models to handle networks containing billions of vertices or edges. Cloud computing have been regarded as a plausible solution for large-scale graph mining for its scalability and efficiency. At present, there are three major cloud computing frameworks for graph mining: MPI-based (e.g., Parallel BGL [5]), BSP-based (e.g., Pregel [6] and Surfer [7]) and MapReduce-based (e.g., PEGASUS [8]) frameworks. MapReduce [9, 10] is a popular cloud computing framework that provides high throughput, satisfiable scalability and fault tolerance. Some tools [11,12,13] have been developed under MapReduce to solve graph mining problems in large-scaled graphs. Among them PEGASUS [8] is a state-of-the-art open source MapReduce-based library that provides several implementations of graph mining algorithms. Although PEGASUS has proposed several efficient implementations for cloud computing, we found that there is still room for improvements in terms of time and space complexity. For example, in our experiment we found that PEGASUS still takes more than 1200 seconds per iteration to run PageRank on a graph with 2 billion edges and 177 thousand vertices on a cluster with 8 nodes while given the same conditions our framework finishes it in only one minute. Besides, it only provides solutions for matrix-vector-multiplication (MVM) based algorithms; hence, it cannot be used with other important graph mining algorithms, such as the Betweenness/closeness Centrality and social network generation, which has been included in our design.

In this work, we present an open source graph mining library called MapReduce Graph Mining Framework (MGMF), which provides scalable and efficient solutions for various graph mining algorithms. MGMF comprises primitive functions, distributed algorithms and optimization methods for diverse kinds of algorithms. We can divide the graph mining algorithms into the following four

categories based on how information is collected in a graph: (1) Traverse all, (2) Traverse Partial, (3) One-Hop, and (4) Multi-Hop. We discuss them in Section 3. We conduct experiments on real-world networks and synthetic networks at the billion scale of vertices/edges to compare the performance of the proposed framework with that of PEGASUS. The results show that using the PageRank algorithm, our framework can run up to 20 times faster than PEGASUS on the Kronecker synthetic dataset, and 3 times faster on the real-world Twitter dataset. We also investigate the effects of file input format on throughput and efficiency.

Furthermore, we propose an efficient parallel social network generation algorithm in this framework. The goal is to generate billion-sized large-scale synthetic graphs that fit real-world social network properties such as short average path length and power-law degree distribution. Among the social network generation algorithms, the Erdős-Rényi model [22], the Watts–Strogatz model [26], and the Barabási–Albert model [27] (respectively abbreviated as ER, WS, BA model) are the most popular models. All of them can fit the property of short average path length. The WS model further provides high clustering coefficient while the BA model generates graph with power-law degree distribution. To parallelize the generation algorithms for efficiency and overcome the limit of memory on single machine, we divide the generation task of the whole graph into generating subgraphs using multiple processors and machines. The ER and WS model can be easily implemented in parallel because the generation tasks of vertices are independent of each other. Unfortunately for the case of BA model, the preferential attachment (abbreviated as PA) mechanism, which generates the edges based on the degree distribution of all vertices, introduces strong dependency between the generation processes of vertices. In other words, generating each vertex requires the degree information from all other vertices spread in all machines, which dramatically increase the transmission load between processors. 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 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.

The major contributions of this paper can be summarized as follows:

1) We propose an efficient framework, MGMF, which comprises primitive functions, distributed algorithms, and optimization methods for various graph mining tasks based on MapReduce. The framework is scalable to the number of machines. The experimental results show that the runtime is reduced to less than 1/3 that of the state-of-the-art MapReduce package.

2) We propose a social network generation model to use *expected cumulative degree* of nodes to facilitate the parallelism of PA models when the target size of networks is too large to fit into a single processor's memory. Such idea, to our knowledge, has not yet been mentioned in the area of social network generation. We further proposed methods to calculate the 'expected cumulative degree' in efficient and accurate approximation.

3) We implemented a high performance open source library for large-scaled graph mining tasks. Our framework is available by this URL: http://mslab.csie.ntu.edu.tw/~noahsark/MGMF/ .

The remainder of the paper is organized as follows. Section 2 provides some essential background information. In Section 3, we introduce the proposed MapReduce Graph Mining Framework. In Section 4, we describe our social network generation model to parallelize preferential attachment. In Section 5, we describe the experiments conducted to evaluate the proposed graph mining framework and social network generation model, and analyze the results.

2. BACKGROUNDS

In this section, we introduce the fundamentals of MapReduce, Hadoop, and some file input formats for graph mining under cloud computing. We review the state-of-the-art graph mining solutions provided by PEGASUS, which comprises primitive functions, MapReduce-based algorithms, and block input formats (M_{block}). We also introduce the background of BA model and existing parallel algorithms that generating scale-free network. We list the notations used in the remainder of the paper, as shown in Table 1.

TABLE 1. SYMBOLS

Symbol	Definition
$G = (V,E)$	a graph consists of vertices V and edges E
d_{avg}	average degree of all vertex in G
d_{outi}	output degree of vertex i
V	vertex vector consists of value (e.g. PageRank) for each vertex in G
$v_i \in V$	element of V, which is vector value (e.g. PageRank), identified by unique vertex id i
$\|V\|$	number of vertices in G
E	edge set consists of all edges in G
$e_{i,j} \in E$	edge, represented by two vertex id (v_i,v_j) denote the edge direction from i to j
$\|E\|$	number of edges in G
$\mathbf{M}$	adjacency matrix, consists of all $e \in G$
$m_{i,j}$	element of $\mathbf{M}$, represents edge $e_{i,j}$ and the weight
M_{edge}	adjacency matrix represented in edge format
M_{adj}	adjacency matrix represented in adjacency list format
M_{block}	adjacency matrix represented in block format
Mapper	process run *Map*(*key*, *value*) on each input record
Reducer	process run *Reduce*(*key*, *values*) on each *key* output by *Map()*
P_M	number of maximal concurrent *Mapper* processes (i.e. the Mapper Capacity of Hadoop cluster)
P_R	number of maximal concurrent *Reducer* processes (i.e. the Reducer Capacity of Hadoop cluster)
w_{blk}	block width in format M_{block}
(row_{blk}, col_{blk})	row and column identifier of a block under the format of M_{block}
(row_{elemi}, col_{elemi})	row and column identifier of element i inside the block under the format of M_{block}

2.1 MapReduce, Hadoop, and File Input Format

MapReduce [9, 10] is a distributed computing framework developed by Google. It provides two basic user definable functions – *Map* (*key*, *value*) and *Reduce* (*key*, *values*) - for *key-value* based processing. Each MapReduce *stage* involves the following *steps*: (1) Input, (2) *Map()*, (3) Shuffle & Sort, (4) *Reduce()*, and (5) Output. The framework provides several benefits, including fault tolerance, load balance, and simple APIs; and it has been shown that the throughput is scalable to the number of machines. Hadoop [14] is a mature implementation of MapReduce that allows programmers to define the following utilities: *Map()*, *Reduce()*, *Combiner()*, *getPartition()*, input/output format, and counter. For more details, readers may refer to [9, 10, 15, 16]. To handle graph data using MapReduce, we have to save the graph in a file. The efficiency of I/O can vary significantly with different file input formats. There are two basic file input formats for graphs [16]: the edge input format (M_{edge}) and the adjacent list input format (M_{adj}). The edge input format (M_{edge}) represents an edge m_{ij} in the form (i, j, m_{ij}) per line; and the unit record is an edge. The adjacency list input format (M_{adj}) represents M in the form of $\{v_j,\{m_1, m_2,\ldots, m_n\}\}$ where m_n is the nth adjacent edge of v_j; and the unit record is a vertex.

2.2 PEGASUS: Matrix-Vector Multiplication on MapReduce

The Matrix-Vector Multiplication (MVM) operation is a useful and general mathematical representation for graph mining. In MVM, each vertex can pass its local information to neighbor vertices (i.e., vertices that are one hop away) by multiplying the adjacency matrix by a vertex vector. As shown in the PEGASUS paper [8], many graph mining algorithms can be represented in the form of MVM, including PageRank and breadth first search.

PEGASUS [8] is an open source library based on Hadoop for large-scale graph mining problems. It provides three primitive functions for MVM:

$$V' = M \times V = assign(v_i, combineAll_i(\{x_j | j=1..n, \text{ and } x_j = combine2(m_{i,j}, v_j)\})). \quad (1)$$

1) *combine2*($m_{i,j}$,v_j): multiplying $m_{i,j}$ and v_j
2) *combineAll*$_i$(x_1,…,x_n): summing n multiplication results for vertex i.
3) *assign*(v_i,v_{new})*:* updating the previous value of v_i with new result v_{new}.

TABLE 2. MAPREDUCE-BASED ALGORITHM FOR MVM: TWO-STAGE MVM

Stage	Two Stage MVM

	1st stage	2nd stage
Map Input	(a)$m_{i,j} \in M$ (b)$v_i \in V$	(a) *key* = *j*, *value* = $m_{i,j} \times v_j$ (b) *key* = *i*, *value* = v_i
Map()	If input(a) then yield (c). If input(b) then yield(d).	identity mapper
Map Output/ Reduce Input	(c) *key* = *j*, *value* = $m_{i,j}$ (d) *key* = *i*, *value* = v_i	(c) *key* = *j*, *value* = $m_{i,j} \times v_j$ (d) *key* = *i*, *value* = v_i
Reduce()	If input(d), then output (f) for all (c),(d), apply *combine2*($m_{i,j}$,v_j), and yield(e)	For all (c), apply v_{new} = $combineAll_i(x_1,\ldots,x_n)$, and then apply *assign*(v_i,v_{new})
Reduce Output	(e) *key* = *j*, *value* = $m_{i,j} \times v_j$ (f) *key* = *i*, *value* = v_i	(e) *key* = *j*, *value* = v_{new}

PEGASUS proposes *GIM-V BASE* algorithm as a MapReduce-based implementation for MVM. The steps of the algorithm are summarized in Table 2. We call the algorithm *Two-Stage MVM* because it must perform MapRedcue twice. In the 1st MapReduce stage, *Map()* receives input from *M* and *V*, and *Reduce()* performs *combine2*(m_i,*j*,v_j) for the same vertex *j*. The output of *Reduce*() is the result of intermediate multiplication. In the 2nd MapReduce stage, *Map*() is used as an identity map (the mapper simply yields its input as output). In addition, *Reduce*() performs *combineAll*() and *assign*() independently to update the new vector value. All MVM-based algorithms (e.g., PageRank and random walk with restart) can be cast in the form of these three primitive functions. Then, PEGASUS applies the *Two-Stage MVM* algorithm to perform the MVM operation. PEGASUS also transforms algorithms of PageRank, random walk with restart, diameter estimation and connected component search into the form of the three primitive functions, and further investigate the scalability of *Two-Stage MVM* based on the implementation of PageRank. Their source codes of PageRank, diameter estimation and connected component search have been released by PEGASUS.

To improve the performance, PEGASUS uses the block file input format (we call it M_{block}), which divides the matrix **M** into several small square blocks with a fixed width w_{blk}. Each block is represented in the form of (row_{blk},col_{blk},{row_{elemi},col_{elemi}}) as a line in the input file. Exploiting M_{block} can reduce the number of input lines (the default unit record in Hadoop), shrink the input file (with a properly chosen w_{blk}), and, reduce the number of records shuffled in MapReduce as a consequence. The results of experiments show that M_{block} is 2 to 5 times faster than M_{edge} on PageRank.

2.3 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 [26], the WS model [27], and the BA model [28] 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.

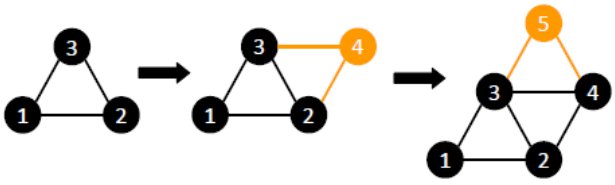


Figure 1. A example of the BA model generation

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 $total(t) = 2mt + 2C_2^{m0}$ 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.

2.4 Existing Parallel Algorithms to Generate Scale-free Network

The parallel version of R-MAT graph algorithm [28] in Parallel BGL [5] and the parallel BA algorithm by Yoo et al. [29] 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 graph generators are implemented including ER model, WS model, and R-MAT graph in MPI. 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. [29] designed two parallel graph generation algorithms to generate scale-free networks, the parallel BA model and parallel Kronecker model [23]. 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

[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.

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.

3. MGMF: MAPREDUCE GRAPH MINING FRAMEWORK

To provide an efficient cloud computing solution for graph mining on MapReduce, we consider the essential components of the successful cloud computing solutions. Generally, the framework of a cloud computing solution is comprised of three types of components:

1) Primitive Function: basic functions, such as APIs for programmers to implement different algorithms.

2) Distributed Algorithm: algorithms with a distributed mechanism to execute the primitive functions.

3) Optimization Method: some methods used to speed up the system.

TABLE 3. MAPREDUCE GRAPH MINING FRAMEWORK

Type of Algorithm	***Primitive function***	***MapReduce-based algorithm***	***Optimization method***
Traverse All	*combine2*()	*One-Stage MVM* *Fast MVM*	*Table*
Traverse partial	*combineAll*() *assign*()		
	combine2() with *prunning*		
One-Hop	None	1 or 2 MapReduce job	*Table*
Multi-Hop	*compute*() *combine*()	*Vertex-Iterator*	*Table*

TABLE 4. CATEGORIZATION OF GRAPH MINING ALGORITHMS

Type of algorithm	**Example Algorithms**
Traverse All	Pagerank [2], Random Walk with Restart [3]
Traverse partial	Diameter Estimation [17], Find Connect Component [1], Eigen Vectors, Belief Propagation [18], Bellman-Ford (Single Source Shortest Sath) [19], Minimum Spanning Tree [19]
One-Hop	In/Out Degree Distribution [20], Input format translation: $M_{edge,}$ $M_{adj,}$ $M_{block,}$ Counting $\|V\|$,$\|E\|$, and graph density [20]
Multi-Hop	Counting triangle [1], Computing Clustering Coefficient [21], Closeness Centrality [20], Betweenness centrality [4]

For example, the PEGASUS framework uses *combine2*(), *combineAll*(), and *assign*() as the primitive functions, *Two-Stage MVM* as the distributed algorithm, and M_{adj} as the optimization method. Several other systems adopt this framework, e.g. MapReduce [9], Pregel[6], and Surfer[7].

The proposed framework, MGMF, first divides graph mining algorithms into four categories. Then, by using the three functions defined above, we propose solutions for algorithms in each category. The proposed MGMF framework is shown in Table 3 and Table 4 lists the graph mining algorithms for each category. We describe these four categories in the following.

1) Traverse All: This category covers the algorithms that must traverse all edges in each iteration. Algorithms in this category can be handled by iterative matrix-vector multiplication. The computation

is repeated until some convergence criterion is met. The time complexity of each algorithm is $O(n(|V|+|E|))$, where n is the number of iterations. Note that the number of iterations is usually much smaller than $|V|$ or $|E|$.

2) Traverse Partial: Only a subset of edges in the graph need to be traversed in each iteration. The algorithms in this category can also be solved by iterative matrix-vector multiplication. However, as only some edges need to be traversed, a *pruning* method can be applied to simplify the computation. The computation cost is less than that of the Traverse All category. The design of the *pruning* algorithm will determine the computation cost.

3) One-Hop: Algorithms in this category only require information from nodes that one hop away from each vertex; and one or two iterations are sufficient for computation (e.g. computing degree distribution). The time complexity is $O(|V|+|E|)$.

4) Multi-Hop: Algorithms of this category require information from nodes that are several hops away for each vertex. The time complexity is much larger than the order of polynomial to $|V|$ or $|E|$. For example, the complexity of Betweenness Centrality is equal to $O(|V||E|)$

Most Traverse All and Traverse Partial algorithms can be implemented by multiplying the vertex vector using the adjacency matrix (MVM); thus, they can be handled by PEGASUS. To improve the efficiency of PEGASUS's MVM, we propose three modifications: 1) using *combine2*() with the *pruning* technique to improve the primitive functions; 2) employing the *Fast MVM* framework to improve the MapReduce-based algorithm; and 3) exploiting the *table* structure proposed by Chen et al. [7] to speed up the optimization process. Normally, One-Hop and Multi-Hop algorithms are not implemented through MVM. Even the modified MVM framework yields little improvement on them. We propose a simple and straightforward implementation on MapReduce for One-Hop tasks. For Multi-Hop tasks, we propose using *compute*() and *combine*() as primitive functions and *Vertex-Iterator* as the MapReduce-based algorithm. We also use the *table* optimization technique to improve the overall performance. In the following subsections, we describe our modifications of MVM, our framework for Traverse All and Traverse Partial, the framework for One-Hop, and the framework for Multi-Hop.

3.1 MapReduce-based algorithms for MVM: One-Stage MVM and Fast MVM

As mentioned earlier, PEGASUS [8] proposes the *Two-Stage MVM* algorithm, which need to perform MapReduce twice; *combine2*(), *combineAll*(), and *assign*() are the primitive functions. Lin et al. [16] describes an implementation of BFS and PageRank on MapReduce. Inspiring from their idea, we propose a generalized MapReduce-based MVM algorithm called *One-Stage MVM*, as shown in Table 5. As its name indicates, our *One-Stage MVM* contains only one stage within MapReduce to perform an MVM operation, but it could incur some overheads. The input format of *One-Stage MVM* is limited to a customized data structure that combines the adjacent list with the vector value of each vertex ($\{v_j,\{m_1,m_2,\ldots,m_n\}\}$). In *Map*() part, *combine2*() is performed while in *Reduce*() part, *combineAll*() and *assign*() are performed.

TABLE 5. ONE-STAGE MVM ALGORITHMS ON MAPREDUCE

Stage	***One Stage MVM***
Map Input	(a) data structure $\{v_j,\{m_1,m_2,\ldots,m_n\}\}$
Map()	for each v_j, $m_{i,j}$ pair, apply *combine2*($m_{i,j}$,v_j) and then yield (c), for(a), yield(d).
Map Output/ Reduce Input	(c) *key* = *j*, *value* = $m_{i,j} \times v_j$ (d) *key* = *j*, *value* = $\{v_j,\{m_1,m_2,\ldots,m_n\}\}$
Reduce()	For all (c), apply $v_{new} = combineAll_i(x_1,\ldots,x_n)$, update structure by {*assign*(v_i,v_{new}), $\{m_1,m_2,\ldots,m_n\}$}, and yield (e)
Reduce Output	(e) *key* = *j*, *value* = $\{v_j,\{m_1,m_2,\ldots,m_n\}\}$

TABLE 6. FAST MVM ALGORITHMS ON MAPREDUCE

Fast MVM

Input: Matrix M = adj list$\{(j, \{i, m_{i,j}\})\}$, Vector $V = \{(i, v_i)\}$
Output: Result Vector $V = \{i, v_{new}\}$

```
Map(Key k, Value v);
  if (k, v) is of type V then
      Output (k, ("self", v))        //(k: i, v: v_i)
  else if (k, v) is of type M then
    v_j = ReadValue(k)               //(k: j)
    for each (i, m_{i,j}) ∈ v do     //(v:{(i, m_{i,j})})
      Output(k, ("others", combine2(m_{i,j},v_j)));
Reduce(Key , Value);
  others_v ← [];                     //initialize
  self_v ← [];                       //initialize
  for each v ∈ v[1..m] do
    (tag, v´) ← v;
    if tag == "self" then
      self_v ← v´
    else if tag == "others" then
      Add v to others_v;
  Output(k, assign(self_v, combineAll_k(others_v)));
```

TABLE 7. TIME AND SPACE COMPLEXITY OF TWO-STAGE MVM, ONE-STAGE MVM, FAST MVM ALGORITHM

Stage	**Time Complexity**		
	Two-Stage MVM	***One-Stage MVM***	***Fast MVM***
Map Input	$2(\|E\|+\|V\|)/P_M$	$(\|E\|+\|V\|)/P_M$	$(\|E\|+\|V\|)/P_M$
Map()	$2(\|E\|+\|V\|)/P_M$	$(\|E\|+\|V\|)/P_M$	$(\|E\|+\|V\|)/P_M$ + O(**ReadValue**) $(\|V\|/P_M)$
Map Output/ Reduce Input	$2(\|E\|+\|V\|)/P_R$	$(\|E\|+2\|V\|)/P_R$	$(\|E\|+\|V\|)/P_R$
Reduce()	$2d_{avg}(\|V\|/P_R)$	$2d_{avg}(\|V\|/P_R)$	$d_{avg}(\|V\|/P_R)$
Reduce Output	$(\|E\|+2\|V\|)/P_R$	$(\|E\|+\|V\|)/P_R$	$\|V\|/P_R$
Stage	**Space Complexity**		
	Two-Stage MVM	***One-Stage MVM***	***Fast MVM***
Map Input	$2(\|E\|+\|V\|)/P_M$	$(\|E\|+\|V\|)/P_M$	$(\|E\|+\|V\|)/P_M$
Map()	O(1)	d_{avg}	d_{avg}
Map Output/ Reduce Input	$2(\|E\|+\|V\|)/P_R$	$(\|E\|+2\|V\|)/P_R$	$(\|E\|+\|V\|)/P_R$
Reduce()	d_{avg}	d_{avg}	d_{avg}
Reduce Output	$(\|E\|+2\|V\|)/P_R$	$(\|E\|+\|V\|)/P_R$	$\|V\|/P_R$

The time complexity and space complexity of *Two-Stage MVM* and *One-Stage MVM* are shown in Table 7. A MapReduce stage involves 5 steps: *Input*, *Map*(), *Shuffle & Sort*, *Reduce*(), and *Output*. Here, we consider the data size and the number of operations performed in each step. We also compare the complexity of different MapReduce-based algorithms. The Steps involved in *Input*, *Shuffle & Sort*, and *Output* perform I/O via the memory/disk/network, so the computation cost is proportional to the size of the required storage. As the steps of *Map*() and *Reduce*() execute user defined algorithms, the input records are processed one by one. That means the minimal space requirement is the largest recorded input. We denote it as the space complexity of *Map*() and *Reduce*(). The time complexity is estimated by multiplying the time complexity of an operation per record by the total number of records

processed. The number of records can be distributed in parallel to different mappers P_M or reducers P_R, so the complexity can be reduced accordingly.

The complexity analysis shows that, in each stage, the computation cost of *Two-Stage MVM* is either greater than or equal to that of *One-Stage MVM*. For *Input* and *Shuffle & Sort*, the computational cost of *Two-Stage MVM* in terms of time complexity is about twice than that of *One-Stage MVM*. Thus, we expect the overall performance of *One-Stage MVM* is at most twice as good as that of the *Two-Stage MVM*. The disadvantage of *One-Stage MVM* is that the output to yields an adjacent list and a vector value in each iteration, which could put some overheads for I/O.

We propose an efficient *Fast MVM* algorithm based on *One-Stage MVM*. Specifically, we use a function *ReadValue*(*k*) to read the vector value of vertex *k* from external source when needed. This approach eliminates the need to combine the adjacency list and vector value as an input record, at the cost of some extra effort in executing the *ReadValue*() function. The pseudo code of *Fast MVM* and its complexity are shown in Table 6. We implement *ReadValue*(*k*) by reading the file of vectors on HDFS (Hadoop Distributed File System [14]) via a file stream. We also modify the partition function *getPartition*(*key, value, numOfReducer*) in Hadoop to ensure that the order of vectors is consistent for each output of MapReduce. The partition function is called *bin partitioner*. It divides all vertex ids into several bins of equal size (*size*=$|V|/P_R$), after which each bin is processed by a specific *Reducer*.

The results in Table 7 show that Fast MVM outperforms or is equivalent to One-Stage MVM in different stage; therefore, we expect Fast MVM to yield a better performance than One-Stage MVM. The experiments described later will follow the analysis in Table 7.

3.2 MVM algorithms: Traverse All and Traverse Partial

We divide the MVM algorithms into *Traverse All* and *Traverse Partial* categories according to whether or all edges need to be traversed during multiplication. For algorithms of *Traverse All* category, to achieve the best performance, we apply the *Fast MVM* algorithm instead of Two-Stage algorithm for implementation. The primitive function of PageRank implementation is the same as PEGASUS, and only the MapReduce-based algorithm is refined in our case. For *Traverse Partial*, we propose to modify *combine2*() with *pruning* technique. This is possible since only updated vector values from the previous MVM iteration are required for multiplication. The *combine2*() function can then be simplified to *combine2*() with *pruning*" as shown in (2). Here, $v_{j,prev}$ denotes the vector value in the previous iteration and v_j denotes the vector value in the current iteration.

$$Combine2(m_{i,j}, v_j) = m_{i,j} \times v_j \text{ if } v_{j,prev} \neq v_j \qquad (2)$$

Diameter Estimation, Identification of Connected Components and Breadth First Search (BFS) belong to this Traverse Partial category. We take the Breadth First Search (BFS) algorithm as an example, which can be applied to solve the single source shortest problem. Initially, the distance value of each vertex is set as infinity, except the source vertex (set as zero). For each MVM iteration, every vertex calculates its distance value by increasing one unit from its neighbors. The three primitive functions are defined in the following:

1) $combine2(m_{i,j},v_j)$: $m_{i,j} \times (v_j+1)$ if $v_{j,prev} \neq v_j$
2) $combineAll_i(x_1,\ldots,x_n)$: $MIN(x_1,\ldots,x_n)$
3) $assign(v_i,v_{new})$: $MIN(v_i,v_{new})$

The *pruning* method is applied to improve the efficiency. The MVM operation is repeated until all vector values converge.

3.3 One-Hop Algorithms

The One-Hop algorithms in Table 5 need only local information (i.e., one-hop) for each vertex. Since there is no need to perform iterative computation for One-Hop algorithms, they can be executed in one or two stages of MapReduce. We show the implementation of Out Degree Distribution and Input Format Transformation from M_{edge} to M_{adj} in Table 8.

TABLE 8. EXAMPLES OF ONE-HOP

Stage	Out Degree Distribution	Format Transformation from M_{edge} to M_{adj}
Map Input	(a){(i,$m_{i,j}$)} $\in M_{adj}$	(a) $m_{i,j} \in M_{edge}$
Map()	yield (b).	yield (b).
Map Output/ Reduce Input	(b) *key* = d_{outi}, *value* = 1	(b) *key* = *j*, *value* = $m_{i,j}$
Reduce()	For all (b), calculate $d = \sum_k 1$, and yield(c)	For all (b), concatenate all $m_{i,j}$, and yield(c)
Reduce Output	(c) *key* = *i*, (out degree) *value* = *d* (frequency)	(c) *key* = *j*, *value* = {(*j*,{(*i*,$m_{i,j}$)})}

3.4 Multi-Hop Algorithms

Graph mining algorithms that require global information during computation (e.g. computing Betweenness Centrality) are belonged to this Multi-Hop category. Since for each iteration, a large amount of computation is needed to go through all vertices in the graph, the time complexity of these algorithms is usually very high. We will show it is inefficient to exploit MVM for Multi-Hop algorithms in next paragraph. We propose to use the *Vertex-Iterator* Algorithm for Multi-Hop algorithms. In *Map*() part, *LoadGraph*() loads the whole graph into the memory and performs *Compute*(*G*, *i*, v_i) on each vertex *i*. In *Reduce*() part, the partial values are gathered by *combine*().

We take and implement Betweenness Centrality (BC) as an example. Brandes [4] proposed an efficient algorithm for calculating BC, as shown in equations (3), (4), (5). The operation involves performing a bread-first-search on each vertex (source) and calculating the partial centrality scores for other vertices (targets).

$$\sigma_{sv} = \sum_{u \in P_s(v)} \sigma_{su} \quad (3)$$

$$\delta_{s\bullet}(v) = \sum_{w:v \in P_s(w)} (\sigma_{sv} / \sigma_{sw}) \cdot (1 + \delta_{s\bullet}(w)) \quad (4)$$

$$C_B(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v) \quad (5)$$

Where σ_{sv} is the number of the shortest path from vertex s to vertex *v*; $P_s(v)$ is the set of predecessor vertexes of *v* in the BFS tree starting from vertex *s*; $\delta_{s\bullet}(v)$ is the partial BC score contributed from source vertex *s* to vertex *v*; $C_B(v)$ is the BC degree of vertex *v*. The time complexity for this algorithm is $O(|V||E|)$. For a large-scale graph, $|V|$ and $|E|$ could be in the order of billions. Thus, under MVM, the volume of intermediate information shuffled between machines would be $O(|V||E|)$, which could be intractable.

To apply the *Vertex-Iterator* algorithm to generate BC, we implement equations (3), (4) and (5) using *Compute*() and *combineAll*() in MapReduce. First, the *Compute*() function executes (3) and (4) for each vertex in *Map*(). Second, the *combineAll*() function performs (5) in *Reduce*(). Through this approach, the volume of intermediate information shuffled between machines would be $|E|P_M + |V|P_M = (|E|+|V|)\ P_M$, which is more tractable than $O(|V||E|)$. Finally, the time complexity is also reduced from $O(|V||E|)$ to $O(|V||E|/P_M + |V|\ P_M/P_R)$.

4. PARALLEL PREFERENTIAL ATTACHMENT MODEL

In this section we introduce our method to parallelize the BA model. We use the same notations as in Section 2.3 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 $total(t_i) = 2mi + 2C_2^{m0}$

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 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.

4.1 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

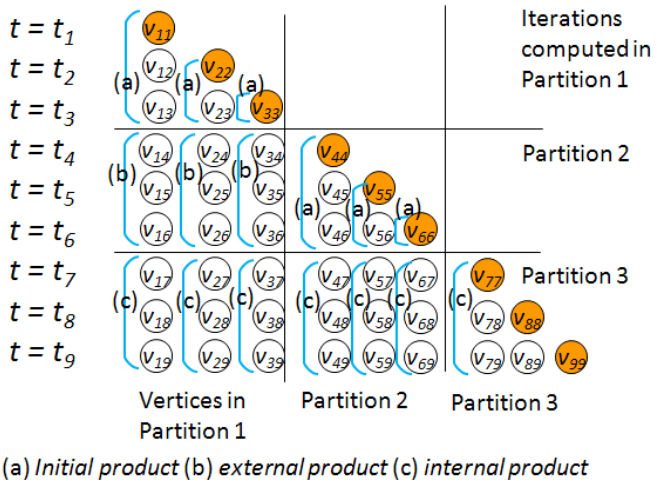


Figure 2. Generating a graph of 9 vertices divided into 3 partitions in the BA model, the colored vertices are new vertices

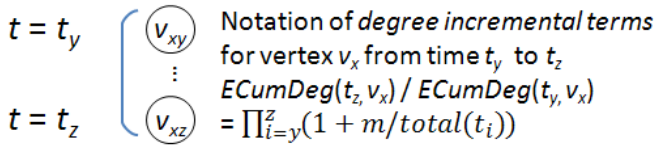


Figure 3. Explanation of graphic notation in Figure 2.

As mentioned in Section 2.3, PA depends on knowing 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.

Assuming 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 4.2 we will prove that

$$EcumDeg(t_j, v_k) = total(t_j) * \prod_{i=k}^{j} (1+m/total(t_i)) \quad (6)$$

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 (6) 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 pre-processed 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.

4.2 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.

$$ExpDeg(t_j,v_k) = ExpDeg(t_k,v_k) \prod_{i=k}^{j-1} (1+m/total(t_i)) \quad (7)$$

(7) 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 $total(t_j)= 2mj+2C_2^{m0}$, since the total degree is $2C_2^{m0}$ at the initial state ($\boldsymbol{t} = 0$) 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

$$ECumDeg(t_j, v_k) = \sum_{i=1}^{k} ExpDeg(t_j, v_k) \qquad (8)$$

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 (7), (8),

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

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 $O(\log N)$.

4.3 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, $ECumDeg(t_9, v_1)/ECumDeg(t_6, v_1) = ECumDeg(t_9, v_2) / ECumDeg(t_6, v_2) = \ldots = ECumDeg(t_9, v_7) / ECumDeg(t_6, v_7) =$, 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 $O(1)$.

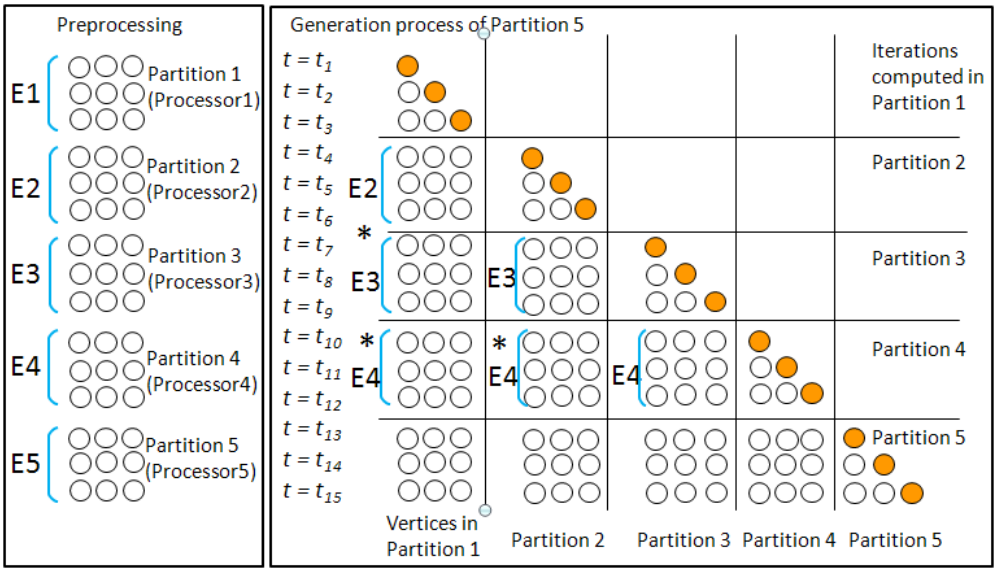


Figure 4. Computing *external product* in process of *partition 5*

4.4 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*.

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

The following is the process we obtain such approximation:

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

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

$$=\frac{2t_{start}+1}{2t_{start}}*\ldots*\frac{2t_{end}+1}{2t_{end}} \qquad (9)$$

$$=\frac{3/2*5/4*7/6*\ldots*(2t_{end}+1)/2t_{end}}{3/2*5/4*7/6*\ldots*2t_{start}/(2t_{start}-1)} \qquad (10)$$

$$\approx\frac{\sqrt{2/\pi}\sqrt{2t_{end}}}{\sqrt{2/\pi}\sqrt{2t_{start}}} \qquad (11)$$

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

In (9) 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 (10) we multiply the term below which is equal to 1

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

Then in (11) we apply the following equation of *Wallis product* [30] for π.
This equation is true if j is infinite, or the error is known to be $O(1/j)$ from *Stirling's approximation.* [31]

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

The following are detailed induction for *Wallis product* substitution in (11).

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

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

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

$$\approx \sqrt{2/\pi}\sqrt{2t_{end}} \qquad (12)$$

Similarly,

$$\frac{3}{2}\frac{5}{4}\frac{7}{6}*...*\frac{2t_{start}}{(2t_{start}-1)} \approx \sqrt{2/\pi}\sqrt{2t_{start}} \qquad (13)$$

Finally we obtained the approximation.

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

To further discuss the error from this approximation, by *Stirling's approximation*, result of (12) becomes

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

and (13) becomes

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

So we consider the error in (11)

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

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

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

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

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

In this approximation we obtained an error term of $(1+O(d/(j^2+jd)))^{1/2}$. After the second partition, $d \leq j$ 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 partition. 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.

4.5 MapReduce Algorithm

Table 9 shows the symbols and functions we use in our algorithm. Table 10 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 edge 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.

4.6 Complexity Analysis

In Table 11 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 9. 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 10. 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 accumulated degree
23              from ← 1
24              to ← t-1
25              v ← binarySearch(R, from, to, external_vec, internal_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 ← internal_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 11. TIME AND SPACE COMPLEXITY

Stage	Time Complexity	Space complexity
Map Input	$O(1)$	$O(1)$
Map()	$O(N/P)$	$O(N/P)$

Stage	Time Complexity	Space complexity
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	$O(m*N/P)$	$O(m*N/P)$

5. EXPERIMENTS

In this section, we present our experiments in two parts. In Section 5.1, we evaluate the efficiency and scalability of our MapReduce Graph Mining Framework on the four algorithm categories. In Section 5.2, we evaluate our parallelized preferential attachment model on the property of power-law degree distribution and also the efficiency and scalability.

5.1 Evaluating MapReduce Graph Mining Framework

We conduct a series of experiments to answer the following questions:

1) Traverse All. Which of the following algorithms can achieve the best performance given the same conditions: *Two-Stage MVM*, *One-Stage MVM* or *Fast MVM*? Which one would yield the best performance if we further exploit the proposed optimization method (i.e., *table* or *combiner*)

2) Traverse Partial. How much improvement can the pruning method achieve?

3) One-Hop. Are One-Hop algorithms scalable to the number of available machines?

4) Multi-Hop. How much improvement does the *Vertex-Iterator* algorithm yield in terms of efficiency?

We implement one algorithm for each question. Besides, we use *speedup* and time *efficiency* criteria to evaluate the scalability of the algorithms. Both criteria are widely used to measure the performance of parallel systems. Speedup S_p is defined as $S_p = T_1/T_P$, where T_1 is the running time using only one process and T_P is the runtime using P processes. The efficiency E_P is defined as $E_P = S_P/P$.

TABLE 12. ORDER AND SIZE OF NETWORKS

Dataset	Nodes	Edges	Description
Twitter[1]	75,792,429	1,468,365,182	person-person in 2009
Erdős-Rényi[2]	177,147	1,977,149,596	synthetic
Kronecker[3]	177,147	1,977,149,596	synthetic
Wikipedia[4]	68,869,555	406,914,840	doc-doc in 2010/01
DBLP[5]	1,380,123	2,194,624	person-doc in 2010/03

Table 12 lists the statistics of the datasets used in our experiments, including two real-world and two synthetic networks. For the synthetic networks, we generate graphs using two random graph models: Erdős-Rényi [22] and Kronecker [23]. The Erdős-Rényi model is a very simple graph generation process producing random graphs. With carefully-designed of the parameter (i.e., the link probability p), we generate graphs following a certain real-world property (i.e., the existence of a giant component). On the other hand, the Kronecker graph is a sophisticated one. It has been proven that the graph obeys many real-world network properties, such as a power-law distribution and the small-world property. For real-world networks, we use Twitter [24] and Wikipedia, two of the most popular web and social network services. Each network, except Wikipedia, contains over 1 billion edges. We evaluate the performance on a cluster system with eight computing nodes, each node is equipped with two Intel E5550 2.67GHz CPUs, 32G DDR3-1333 RAM, 1Gb Ethernet network and 1TB hard disk. In addition, we use Hadoop 0.20.2 with Mapper capacity = 8 and Reducer capacity = 3 per node. All runtimes are averaged over 5 runs. The input format of M_{adj} with Gzip binary compression is exploited for all of our algorithms and the compared approaches. It is because the experiment results indicate that M_{adj} with Gzip outperforms than other setups, which are combination of M_{edge} and M_{block} (discussed in the succeeding section).

5.1.1 Evaluating Traverse All Algorithms: Exemplifying by PageRank

We implement PageRank on *Two-Stage MVM*, *One-Stage MVM* and *Fast MVM*. All algorithms are executed on the four datasets. The average runtime per iteration of PageRank is shown in Figure 5. We can observe that: (1) Both *One-Stage MVM* and *Fast MVM* significantly outperform *Two-Stage MVM* (they are roughly twice as fast). Compared to *Two-Stage MVM*, the *Fast MVM* reduces the runtime to 51% on real-world networks and 40-45% on synthetic networks. *One-Stage MVM* reduces the runtime only by 48-69% on real-world networks and by 64-76% on synthetic networks. Their variation is quite large. (2) Generally, for low density dataset (i.e., the average degree is low), such as Wikipedia, *One-Stage MVM* performs slightly better than *Fast MVM*. For high-density graphs, *Fast MVM* yields a better performance. The reason is that *Fast MVM* has to do external access (HDFS, in our implementation) to fetch the vector values, but *One-Stage MVM* performs internal access (from HDFS, refer to [9] for the details), which has less overheads. Thus, the overhead is more for *Fast MVM* in sparse graphs since the ratio of access and processing (which is equal to the inverse of the graph's density) is larger. Figure 5 also shows that, for networks whose density is less than 5% such as Wikipedia, *One-Stage MVM* is a more suitable choice. (3) By integrating the proposed *Fast MVM* with the *Table* method (a buffer in the mapper to cache the intermediate data) proposed by Chen et al. [7], the performance becomes even better. Compared to *Two-Stage MVM*, the runtime is reduced to about 1/4 on real-world networks and 1/20 on synthetic networks. Similarly, the performance improves on dense networks. (4) The *Table* method outperforms *Combiner* in terms of runtime on both *One-Stage MVM* and *Fast MVM*. We believe this is because the *Table* method simplifies the output before the sorting step, and the *Combiner* simplifies the output after the sorting step. Note that sorting tasks in MapReduce procedures spend undeniable amount of time. Interestingly, the experiment results show that using both *Combiner* and *Table* do not achieve the best performance. We believe this is because *Table* is a cache mechanism inside the Mapper with fewer costs, whereas *Combiner* is a process outside the *Mapper*.

We examine the scalability of the proposed framework by exploiting *Fast MVM* and the *Table* method on PageRank. The runtime, speedup, and efficiency on the Kronecker network are shown in Figure 6. The results demonstrate that on a cluster machine containing up to eight nodes. When the computation of a job is simple enough, the runtime would be dominated by overheads of MapReduce. That is the reason why the performance for more than 5 nodes does not improve significantly in Figure 6 and Figure 8. Nevertheless, our implementation is able to finish the computation of PageRank of the Kronecker network with 2 billion edges in 45 seconds.

5.1.2 Evaluating Traverse Partial Algorithms: Exemplifying by Breadth First Search

We implement Breadth First Search described in Section 3.B with *Fast MVM* and run on it on the Twitter dataset. In addition, we implement the primitive function *combine2*() with and without *pruning*. From the results in Figure 7, we can observe that the BFS takes 15 iterations to converge. However, by applying *pruning*, the run time can be further reduced to roughly 40%.

5.1.3 Evaluating One-Hop Algorithms: Exemplifying by Calculating the Out-degree Distribution

We implement the Out-degree Distribution described in in Table 8 on the Kronecker dataset. The results using different numbers of computing nodes are shown in Figure 8. The reason for low efficiency when using more computing nodes is similar to the case we discussed in Section 5.1.1.

5.1.4 Evaluating Multi-Hop Algorithms: Exemplifying by Calculating the Betweenness Centrality

We realize Brandes' Betweenness Centrality (BC) algorithm as the example of the Multi-Hop category in the proposed framework. We implement the *Mapper* program in C++ and use Hadoop Streaming to run MapReduce. The input file of each *Mapper* is copied to each machine. The scalability is shown in Figure 9. Our BC implementation algorithm has good linear scalability. Since the complexity of the BC algorithm is $O(|V||E|)$, it is computationally intractable to run it on a network containing billions of entities. Therefore, we perform the experiment on the DBLP dataset, which

[1]Twitter: available at http://an.kaist.ac.kr/traces/WWW2010.html
[2]Erdős-Rényi: available at http://www.cs.cmu.edu/~ukang/HADI/
[3]Kronecker: available at http://www.cs.cmu.edu/~ukang/HADI/
[4]Wikipedia: available at http://mslab.csie.ntu.edu.tw/~noahsark/
[5]DBLP: available at http://mslab.csie.ntu.edu.tw/~noahsark/

comprises 1.3 million vertices and 2.1 million edges. It still takes 6 hours to finish the experiment on a cluster system with eight computing nodes.

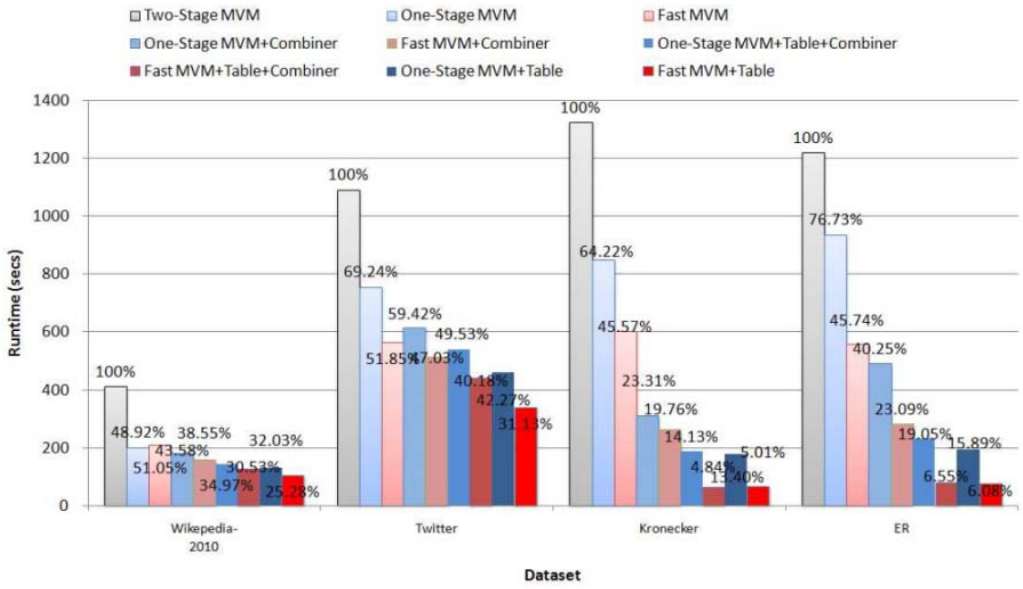
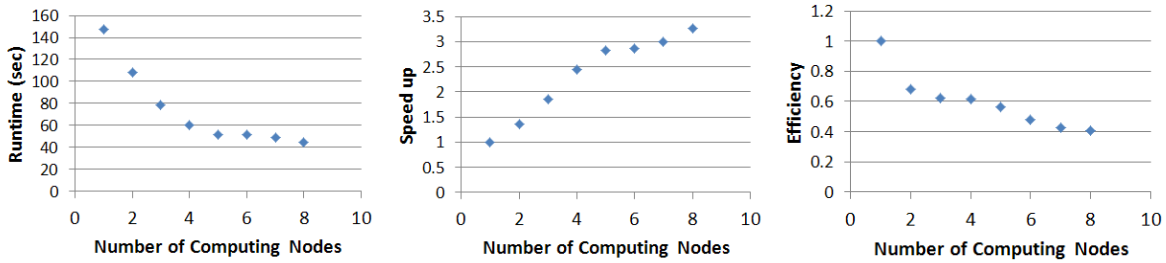


Figure 5. The efficiency of PageRank on *Two-Stage MVM*, *One-Stage MVM* and *Fast MVM* algorithms as well as optimization methods



(a) Runtime (b) Speedup (c) Efficiency

Figure 6. Experiments for *Traverse All*: Implementation of PageRank by exploiting *Fast MVM* and *Table* method on Kronecker Network

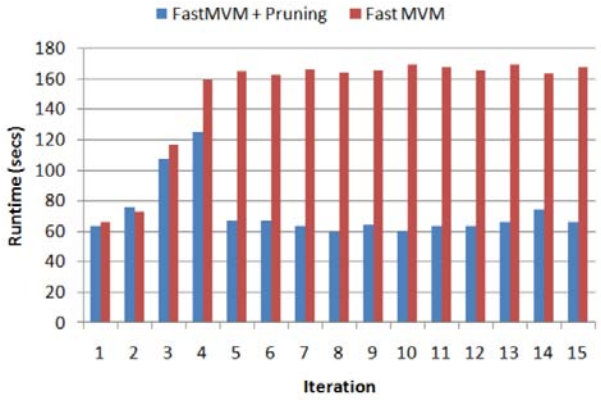
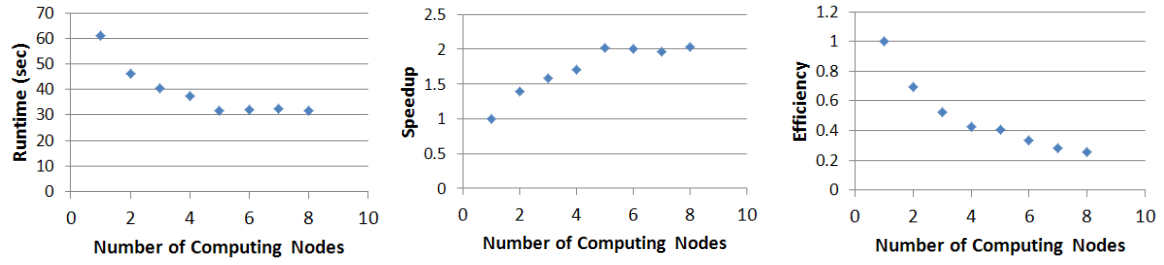
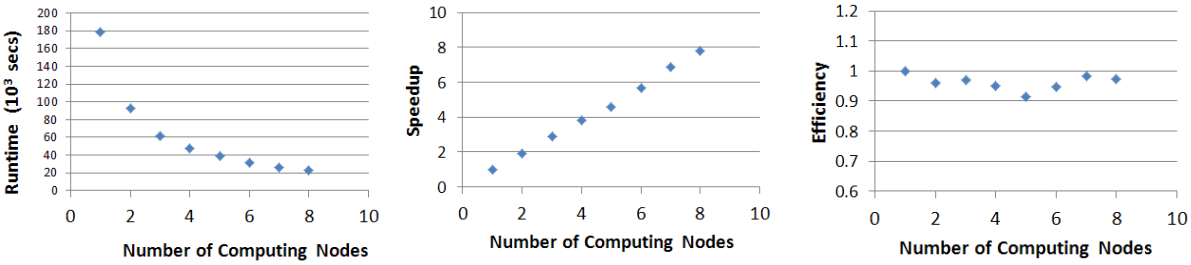


Figure 7. Experimental result for *Traverse Partial*: time efficiency of primitive functions with and without *pruning*



(a) Runtime (b) Speedup (c) Efficiency

Figure 8. Experiments for *One-Hop*: Implementation of Out Degree Distribution on Kronecker Network



(a) Runtime (b) Speedup (c) Efficiency

Figure 9. Experiments for *Multi-Hop*: Calculating Betweenness Centrality by *Vertex-Iterator* on DBLP Network

5.2 Evaluating Parallelized Preferentail Attachment Model

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.

5.2.1 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.

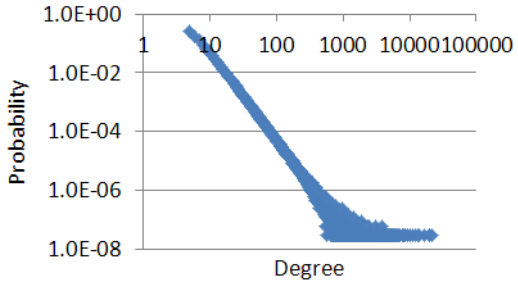


Figure 10. The degree distribution of the graph generated by our algorithm

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 13. 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 11. 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 13. 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

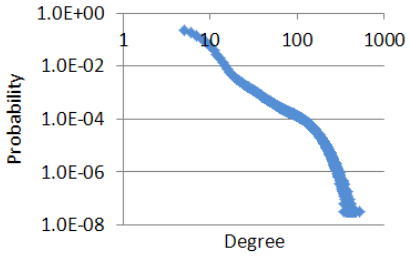


Figure 11. The degree distribution of the graph generated by Yoo's algorithm

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

5.2.2 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 12, 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 13 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.

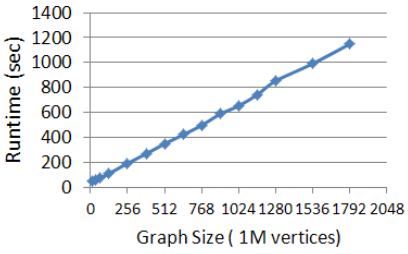
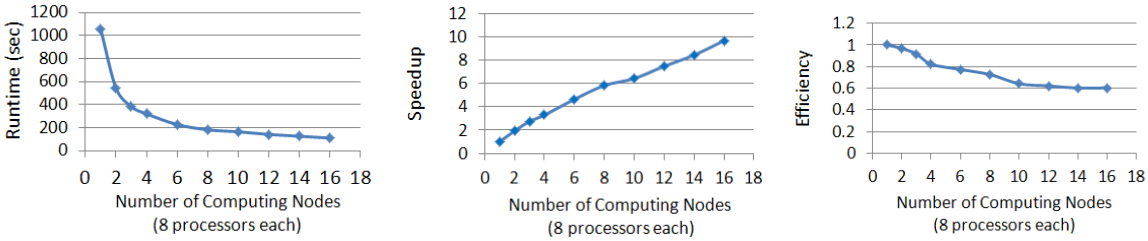


Figure 12. Experiment for runtime with 16 computing nodes when increasing graph size



(a) Runtime (b) Speedup (c) Efficiency

Figure 13. Experiment on generating 128 million vertices

6. CONCLUSION

We have proposed the MapReduce Graph Mining Framework (MGMF) for graph mining algorithms and a parallel social network generation algorithm based on MapReduce.

MGMF divides the graph mining algorithms into four categories based on what kind of graph information is needed when performing algorithms. Our designed framework consists of three components, including primitive functions, a MapReduce-based algorithm, and an optimization method. We demonstrate that the proposed algorithms are more scalable and efficient than the state-of-the-art MapReduce-based graph mining package (i.e., PEGASUS). Specifically, we propose a comprehensive, end-to-end system design for graph mining in a cloud environment. Our framework contains a File Input Format, Primitive Functions, MapReduce-based Algorithm, and Cache with Compression methods for optimization. We compile our MGMF as an open source, ready-for-use library available at http://mslab.csie.ntu.edu.tw/~noahsark/MGMF/. The data sets used in the experiments are also available online. We also demonstrate the usage of MGMF in diverse scenarios. Cohen [25] noted that map-reduce-reduce is a common operation for graph mining on MapReduce. That says carefully design of chained MapReduce operation could further reduce the overhead between iterations, which becomes one of our future work.

Besides graph mining algorithms, we also proposed an idea of 'expected accumulated degree' to parallelize the preferential attachment mechanism in BA model. We 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 valu*es can be applied to other parallelism tasks in social network.

REFERENCES

[1] M. E. J. Newman, The structure and function of complex networks, SIAM Review 45 (2) (2003) 167-256.

[2] L. Page, S. Brin, R. Motwani, T. Winograd, The pagerank citation ranking: Bringing order to the web., Technical Report 1999-66, Stanford InfoLab (November 1999).

[3] H. Tong, C. Faloutsos, J.-Y. Pan, Fast random walk with restart and its applications, in: ICDM '06: Proceedings of the Sixth International Conference on Data Mining, IEEE Computer Society, Washington, DC, USA, 2006, pp. 613-622.

[4] U. Brandes, A faster algorithm for betweenness centrality, Journal ofMathematical Sociology 25 (2) (2001) 163-177.

[5] D. Gregor, A. Lumsdaine, The Parallel BGL: A generic library for distributedgraph computations, Parallel Object-Oriented Scientific Computing (POOSC).

[6] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing, in: SIGMOD'10: Proceedings of the 36th SIGMOD international conference on Management of data, ACM, New York, NY, USA, 2010.

[7] R. Chen, X. Weng, B. He, M. Yang, C. B., L. X., On the Efficiency and Programmability of Large Graph Processing in the Cloud, Microsoft Research TechReport.

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

[9] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: OSDI, 2004, pp. 137-150.

[10] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107-113.

[11] J. Hebert, Hadoop examples - pagerank, available at http://code.google.com/p/canopy-clustering/ (2007).

[12] B. Cai, W. Xue, X-rime: Hadoop based large scale social network analysis, available at http://xrime.sourceforge.net/ (2009).

[13] L. Akoglu, P. Chau, C. Faloutsos, U. Kang, K. Maruhashi, M. McGlohon, P. Stancioli, C. E. Tsourakakis, Available at http://www.cs.cmu.edu/~pegasus/

[14] D. Cutting, Apache hadoop, available at http://hadoop.apache.org/ (2007).

[15] S. Ghemawat, H. Gobioff, S.-T. Leung, The google file system, in: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM, New York, NY, USA, 2003, pp. 29-43.

[16] J. Lin, C. Dyer, Data-intensive text processing with MapReduce, in: Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts, Association for Computational Linguistics, 2009, pp. 1-2.

[17] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, J. Leskovec., Radius plots for mining tera-byte scale graphs: Algorithms, patterns, and observations, SIAM International Conference on Data Mining.

[18] J. Pearl, Probabilistic reasoning in intelligent systems: networks of plausible inference, Morgan Kaufmann, 1988.

[19] T. Cormen, Introduction to algorithms, The MIT press, 2001.

[20] J. Scott, Social Network Analysis: A Handbook, second. Edition, Sage Publications, 2000.

[21] T. Schank, Algorithmic aspects of triangle-based network analysis., Ph. d. thesis, University Karlsruhe (February 2007).

[22] P.Erdős, A. Rényi, On the evolution of random graphs, Publication of the Mathematical Institute of the Hungarian Academy of Science (1960) 17-61.

[23] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication, Knowledge Discovery in Databases: PKDD 2005 (2005) 133-145.
[24] H. Kwak, C. Lee, H. Park, S. Moon, What is Twitter, a social network or a news media?, in: Proceedings of the 19th international conference on World wide web, ACM, 2010, pp. 591-600.
[25] J. Cohen, Graph twiddling in a mapreduce world., Computing in Science and Engineering 11 (4) (2009) 29-41.
[26] D. Watts and S. Strogatz (1998): Collective dynamics of small-world networks. *Nature, 363*:202–204.
[27] R. Albert; A.-L. Barabási (2002). "Statistical mechanics of complex networks". *Reviews of Modern Physics* **74**: 47–97.
[28] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: Arecursive model for graph mining. In *SDM*, 2004.
[29] Andy Yoo, Keith W. Henderson. Parallel Generation of Massive Scale-Free Graphs. CoRR, 2010.
[30] Sondow, Jonathan and Weisstein, Eric W. "Wallis Formula." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/WallisFormula.html
[31] Weisstein, Eric W. "Stirling's Approximation." From MathWorld--A Wolfram Web Resource. http://mathwor-ld.wolfram.com/StirlingsApproximation.html