2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining

Exploiting and Evaluating MapReduce for Large-Scale Graph Mining

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

Abstract—**Graph mining is a popular technique for discovering the hidden structures or important instances in a graph, but the computational efficiency is usually a cause for concern when dealing with large-scale graphs containing billions of entities. Cloud computing is widely regarded as a feasible solution to the 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 four categories and designing 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 better coverage of different graph mining algorithms. We also validate our framework on billion-scaled networks to demonstrate that it is scalable to the number of machines. Furthermore, we test and compare the feasibility between single machine and the cloud computing technique. The effects of different file input formats for MapReduce are investigated as well. 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

I. INTRODUCTION

There are many popular graph mining tasks [1], such as computing properties of networks (e.g., degree distribution), estimating centralities (e.g., PageRank [2], random walk with restart [3], Betweenness Centrality [4]), counting triangles, and finding connected components [1]. Because online social network services are becoming increasingly popular (e.g., Twitter and Facebook), there is an urgent need for an application that can perform graph mining efficiently on large-scale social networks containing billions of vertices or edges. Cloud computing can be regarded as a plausible solution for large-scale graph mining since 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. Many tools [11,12,13] have been developed under MapReduce to solve graph mining problems in large-scaled graphs. 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, PEGASUS still takes 100 seconds per iteration to run PageRank on a graph with 282 million edges and 59 thousand vertices on a cluster machine with 30 nodes. Besides, PEGASUS 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 Centrality and Closeness Centrality. 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, (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. Besides, we test and compare the feasibility between single machine and MapReduce technique according to the runtime. We also investigate the effects of file input format on throughput and efficiency.

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.

2) We divide the graph mining algorithms into four categories. Besides, we propose some novel and efficient implementations for algorithms in each category. The experimental results show that the runtime is reduced to less than 1/3 that of the state-of-the-art MapReduce package.

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 the experiments conducted to evaluate the proposed framework, and analyze the results. Section 5 contains some concluding remarks.

978-0-7695-4799-2/12 $26.00 © 2012 IEEE
DOI 10.1109/ASONAM.2012.77

II. BackGrounds

In this section, we introduce the fundamentals of MapReduce, Hadoop, and some file input formats for graph mining under cloud computing. We also 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}).

First, in Table 1, we define the notations used in the remainder of the paper.

TABLE I. 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}

A. *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.

B. *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 II. 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,…,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 $\mathbf{M}$ into several small square blocks with a fixed width w_{blk}. Each block is represented in the form of ($\mathrm{row}_{\mathrm{blk}}$,$\mathrm{col}_{\mathrm{blk}}$,$\{row_{elemi}, col_{elemi}\}$) 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.

III. 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 III. MapReduce Graph Mining Framework

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

TABLE IV. 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 $\lvert V\rvert$,$\lvert E\rvert$, 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.

A. 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 ($\{v_j,\{m_1,m_2,...,m_n\}\}$). In *Map*() part, *combine2*() is performed while in *Reduce*() part, *combineAll*() and *assign*() are performed.

TABLE V. ONE-STAGE MVM ALGORITHMS ON MAPREDUCE

Stage	***One Stage MVM***
Map Input	(a) data structure $\{v_j,\{m_1,m_2,...,m_n\}\}$
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) $key = j$, $value = \{v_j,\{m_1,m_2,...,m_n\}\}$
Reduce()	For all (c), apply $v_{new} =$ *combineAll*$_i$(x_1,…,x_n), update structure by {*assign*(v_i,v_{new}), $\{m_1,m_2,...,m_n\}$}, and yield (e)
Reduce Output	(e) $key = j$, $value = \{v_j,\{m_1,m_2,...,m_n\}\}$

TABLE VI. 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 ($i$, $m_{i,j}$) $\in$ $v$ 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 $v \in v[1..m]$ 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 VII. 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.

B. 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 *Trav-*

erse 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 Equation 2. Here, $v_{j,prev}$ denotes the vector value in the previous iteration and v_j denotes the vector value in the current iteration.

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

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,…,x_n): *MIN*(x_1,…,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.

C. 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 VIII. 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* = $\{(j,\{(i,m_{i,j})\})\}$

D. 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).

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

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

(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 Equations 3 and 4 for each vertex in *Map*(). Second, the *combineAll*() function performs Equation 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 $O(|V||E|/P_M + |V| P_M/P_R)$.

IV. Experiments

In this section, we evaluate the efficiency and scalability of our 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 IX. 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 9 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).

A. *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 1. 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. Our experimental results also show that, for networks whose density is less than 5%, *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 2. The results demonstrate that on a cluster machine containing up to eight nodes, our proposed methods are close to linear speedup and with efficiency at least 0.8 times.

B. *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 3, 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%.

C. *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 4. It can be observed that the algorithm is scalable up to 8 computing nodes with efficiency at least 0.6 times.

D. *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 5. 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 comprises 1.3 million vertices and 2.1 million edges. It still takes 6 hours

[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/

to finish the experiment on a cluster system with eight computing nodes.

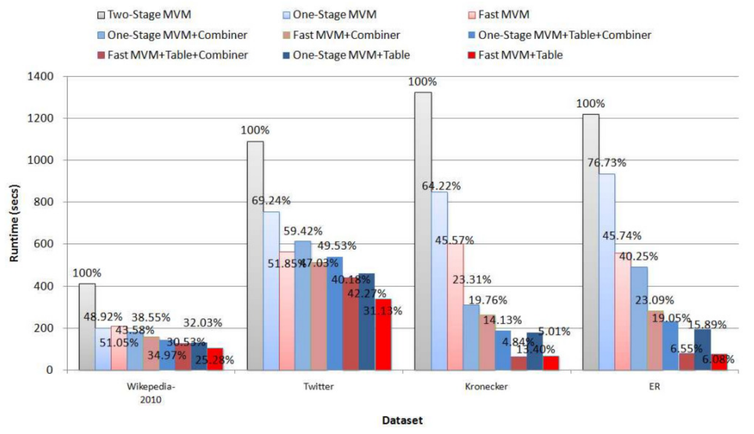
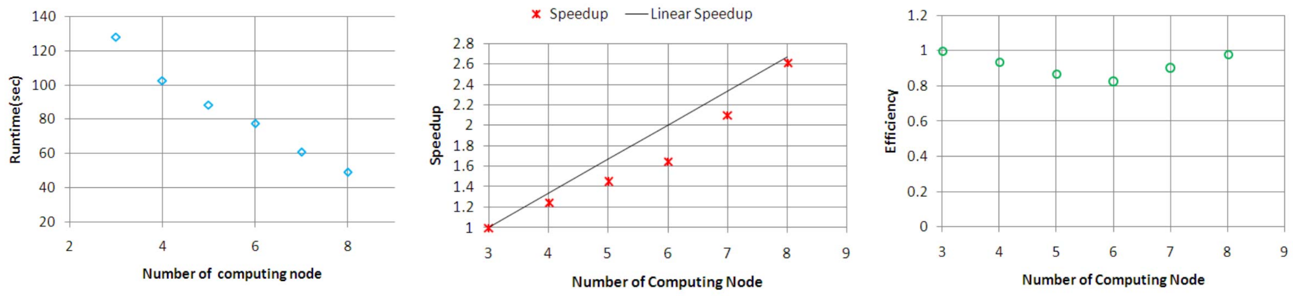


Figure 1. The efficiency of PageRank on *Two-Stage MVM*, *One-Stage MVM* and *Fast MVM* algorithms as well as optimization methods



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

Figure 2. Experiments for *Traverse All*: Implementation of PageRank by exploiting *Fast MVM* and *Table* method on Kronecker Network

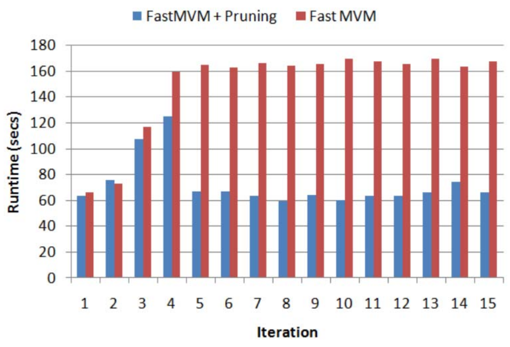
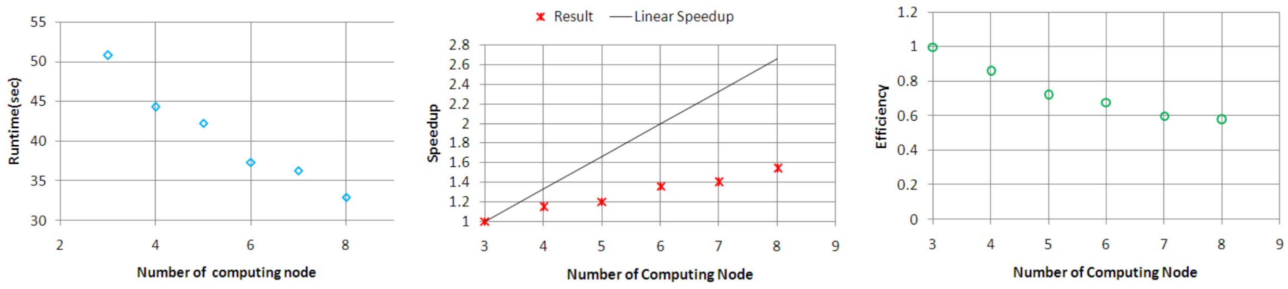
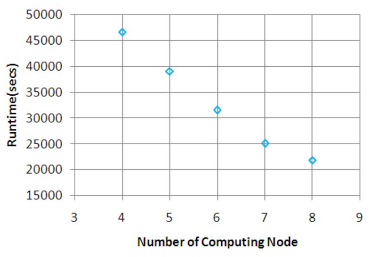


Figure 3. Experimental result for *Traverse Partial*: time efficiency of primitive functions with and without *pruning*

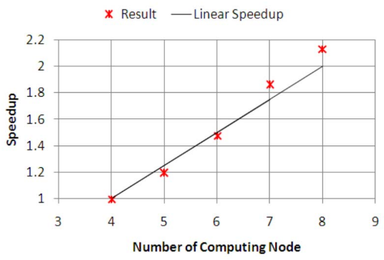


(a) Runtime (b) Speedup compared with using 3 computing nodes (c) Efficiency compared with using 3 computing nodes

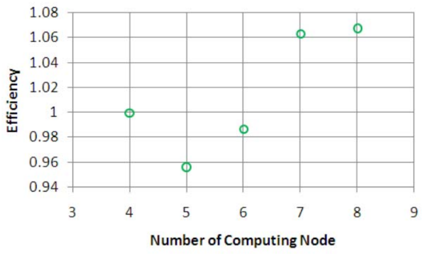
Figure 4. Experiments for *One-Hop*: Implementation of Out Degree Distribution on Kronecker Network



(a) Runtime



(b) Speedup compared with using 4 computing nodes



(c) Efficiency compared with using 4 computing nodes

Figure 5. Experiments for *Multi-Hop*: Calculating Betweenness Centrality by *Vertex-Iterator* on DBLP Network

V. Concluding Remarks

We have proposed the MapReduce Graph Mining Framework (MGMF) for graph mining algorithms 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. Besides, 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. In the end, we evaluate the usefulness and scalability of MGMF in a cluster machine. We also demonstrate the usage of MGMF in diverse scenarios.

For the future direction, we will improve the efficiency of the computation on graph mining by modifying MapReduce. 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.

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.