Chen R, Shi JX, Chen HB *et al.* Bipartite-oriented distributed graph partitioning for big learning. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 30(1): 20–29 Jan. 2015. DOI 10.1007/s11390-015-1501-x

Bipartite-Oriented Distributed Graph Partitioning for Big Learning

Rong Chen (陈 榕), *Member, CCF, ACM, IEEE*, Jia-Xin Shi (施佳鑫)
Hai-Bo Chen* (陈海波), *Senior Member, CCF, Member, ACM, IEEE*, and
Bin-Yu Zang (臧斌宇), *Senior Member, CCF, Member, ACM, IEEE*

Shanghai Key Laboratory of Scalable Computing and Systems, Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University, Shanghai 200240, China

E-mail: {rongchen, jiaxinshi, haibochen, byzang}@sjtu.edu.cn

Received July 15, 2014; revised October 14, 2014.

Abstract Many machine learning and data mining (MLDM) problems like recommendation, topic modeling, and medical diagnosis can be modeled as computing on bipartite graphs. However, most distributed graph-parallel systems are oblivious to the unique characteristics in such graphs and existing online graph partitioning algorithms usually cause excessive replication of vertices as well as significant pressure on network communication. This article identifies the challenges and opportunities of partitioning bipartite graphs for distributed MLDM processing and proposes BiGraph, a set of bipartite-oriented graph partitioning algorithms. BiGraph leverages observations such as the skewed distribution of vertices, discriminated computation load and imbalanced data sizes between the two subsets of vertices to derive a set of optimal graph partitioning algorithms that result in minimal vertex replication and network communication. BiGraph has been implemented on PowerGraph and is shown to have a performance boost up to 17.75X (from 1.16X) for four typical MLDM algorithms, due to reducing up to 80% vertex replication, and up to 96% network traffic.

Keywords bipartite graph, graph partitioning, graph-parallel system

1 Introduction

As the concept of "Big Data" gains more and more momentum, running many MLDM problems in a cluster of machines has become a norm. This also stimulates a new research area called big learning, which leverages a set of networked machines for parallel and distributed processing of more complex algorithms and larger problem sizes. This, however, creates new challenges to efficiently partition a set of input data across multiple machines to balance load and reduce network traffic.

Currently, many MLDM problems concern large graphs such as social and web graphs. These problems are usually coded as vertex-centric programs by following the "think as a vertex" philosophy[1], where vertices are processed in parallel and communicate with their neighbors through edges. For the distributed processing of such graph-structured programs, graph partitioning plays a central role to distribute vertices and their edges across multiple machines, as well as to create replicated vertices and/or edges to form a locally-consistent sub-graph states in each machine.

Though graphs can be arbitrarily formed, real world graphs usually have some specific properties to reflect their application domains. Among them, many MLDM algorithms model their input graphs as bipar-

Regular Paper
Special Section on Computer Architecture and Systems for Big Data
This work was supported in part by the Doctoral Fund of Ministry of Education of China under Grant No. 20130073120040, the Program for New Century Excellent Talents in University of Ministry of Education of China, the Shanghai Science and Technology Development Funds under Grant No. 12QA1401700, a foundation for the Author of National Excellent Doctoral Dissertation of China, the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing under Grant No. 2014A05, the National Natural Science Foundation of China under Grant Nos. 61003002, 61402284, the Shanghai Science and Technology Development Fund for High-Tech Achievement Translation under Grant No. 14511100902, and the Singapore National Research Foundation under Grant No. CREATE E2S2.
A preliminary version of the paper was published in the Proceedings of APSys 2014.
* Corresponding Author

tite graphs, whose vertices can be separated into two disjoint sets U and V and every edge connects a vertex in U and V. Example bipartite graphs include the customers/goods graph in recommendation systems and the topics/documents graph in topic modeling algorithms. Due to the wide existence and importance of bipartite graphs, there have been a number of popular MLDM algorithms that operate on such graphs, including Singular Value Decomposition (SVD), Alternating Least Squares (ALS), Stochastic Gradient Descent (SGD), Belief Propagation (BP), and Latent Dirichlet Allocation (LDA), with the application domains ranging from recommendation systems to medical diagnosis and text analysis.

Despite the popularity of bipartite graph, there is little study on how to efficiently partition bipartite graphs in large-scale graph computation systems. Most existing systems simply apply general graph partitioning algorithms that are oblivious to the unique features of bipartite graphs. This results in suboptimal graph partitioning with significant replicas of vertices and/or edges, leading to not only redundant computation, but also excessive network communication to synchronize graph states. Though there are some graph partitioning algorithms for bipartite graphs, none of them satisfies the requirement of large-scale graph computation. Many of them[2-3] are offline partitioning algorithms that require full information of graphs, and thus are very time-consuming and not scalable to large graph dataset. Some others[4-5] only work on a special type of graphs or algorithms, making them hard to be generalized to a wide range of MLDM problems.

In this paper, we make a systematic study on the characteristics of real world bipartite graphs and the related MLDM algorithms, and describe why existing online distributed graph partitioning algorithms fail to produce an optimal graph partition for bipartite graphs. Based on the study, we argue that the unique properties of bipartite graphs and the special requirement of bipartite algorithms demand *differentiated* partitioning[6] of the two disjoint sets of vertices. Based on our analysis, we introduce BiGraph, a set of distributed graph partitioning algorithms designed for bipartite applications. The key of BiGraph is to partition graphs in a differentiated way and load data according to the data affinity.

We have implemented BiGraph① as separate graph partitioning modules of GraphLab[7-8], a state-of-the-art graph-parallel framework. Our experiment using three typical MLDM algorithms on an in-house 6-machine multicore cluster with 144 CPU cores and an EC2-like 48-machine multicore cluster with 192 CPU cores shows that BiGraph reduces up to 81% vertex replication and saves up to 96% network traffic. This transforms to a speedup up to 17.75X (from 1.16X) compared with the state-of-the-art Grid[9] partitioning algorithm (the default partitioning algorithm in GraphLab).

In the next section (Section 2), we introduce graph-parallel systems and existing graph partitioning algorithms. The observation to motivate bipartite-oriented graph partitioning is then discussed in Section 3. Next, the detailed design and implementations are described in Section 4. Then we provide our experimental setting, workload characteristics, and results analysis in Section 5. Finally, we conclude this paper in Section 6.

It is worth noting that a preliminary version of this paper was published in the Proceedings of the 5th Asia-Pacific Workshop on Systems[10]. This version includes a detailed analysis and comparison on the greedy heuristic vertex-cut, evaluates the performance on a new EC2-like 48-machine cluster, and provides experiments on greedy heuristic vertex-cut.

2 Background and Related Work

This section provides the background information and related work on distributed graph-parallel systems and graph partitioning algorithms.

2.1 Graph-Parallel Systems

Many distributed graph-parallel systems, including Pregel[1], Giraph②, Cyclops[11] and GraphLab[7-8], follow the "think as a vertex" philosophy and abstract a graph algorithm as a vertex-centric program P. The program is executed in parallel on each vertex $v \in V$ in a sparse graph $G = \{V, E, D\}$. The scope of computation and communication in each $P(v)$ is restricted to neighboring vertices through edges where $(s, t) \in E$. Programmers can also associate arbitrary data D_v and $D_{(s,t)}$ with vertex $v \in V$ and edge $(s, t) \in E$ respectively.

Fig.1(b) illustrates the overall execution flow for the sample graph on PowerGraph[8], the latest version of

① The source code and a brief instruction of how to use BiGraph are at http://ipads.se.sjtu.edu.cn/projects/powerlyra.html, Nov. 2014.

② http://giraph.apache.org/, Nov. 2014.

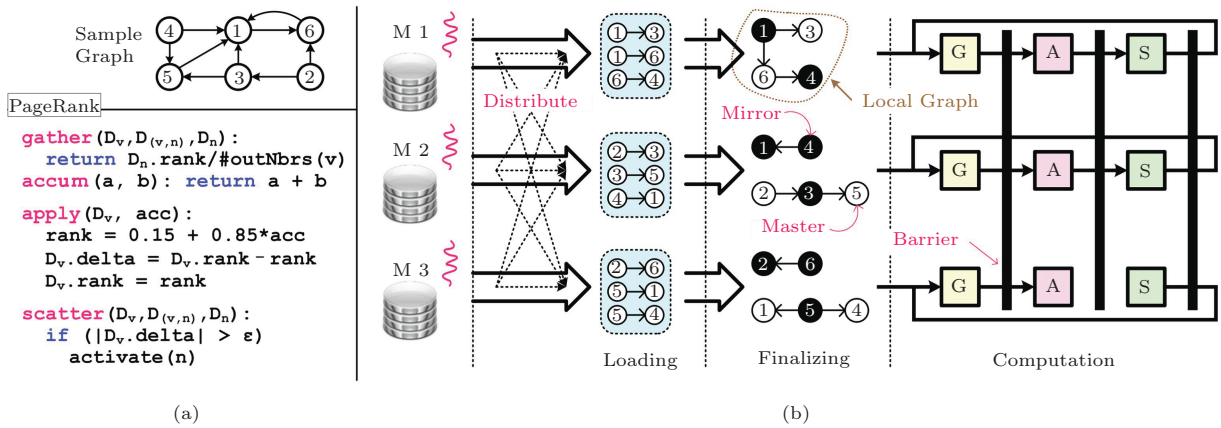


Fig.1. (a) Pseudo-code of PageRank algorithm using GAS model. (b) Execution flow of graph-parallel system for a sample graph.

the GraphLab framework. To exploit distributed computing resources of a cluster, the runtime splits an input graph into multiple partitions and iteratively executes user-defined programs on each vertex in parallel.

First, the runtime *loads* graph topology and data files from a secondary storage (e.g., Hadoop distributed file system (HDFS) or network file system (NFS)), and *distributes* vertices and edges to target machines according to a graph partitioning algorithm. Then, the replicas for vertices are created for each edge spanning machines to *finalize* in-memory local graphs such that each machine has a locally-consistent sub-graph. PowerGraph adopts the GAS (Gather, Apply, Scatter) model to abstract graph algorithms and employs a loop to express iterative computation in many MLDM algorithms. The pseudo-code in Fig.1(a) illustrates an example implementation of the PageRank[12] algorithm implemented by the GAS model. In the Gather phase, the `gather` and the `accum` functions accumulate the rank of neighboring vertices through in-edges; and then the `apply` function calculates and updates a new rank to vertex using accumulated values in the Apply phase; finally the `scatter` function sends messages and activates neighboring vertices through out-edges in the scatter phase.

2.2 Distributed Graph Partitioning

A key to efficient big learning on graph-parallel systems is optimally placing graph-structured data including vertices and edges across multiple machines. As graph computation highly relies on the distributed graph structures to store graph computation states and encode interactions among vertices, an optimal graph partitioning algorithm can minimize communication cost and ensure the load balance of vertex computation.

There are two main types of approaches: offline and online (streaming) graph partitioning. Offline graph partitioning algorithms (e.g., Metis[13], spectral clustering[14] and k-partitioning[15]) require that full graph information has been known by all workers (e.g., machines), which requires frequent coordination among workers in a distributed environment. Though it may produce a distributed graph with optimal graph placement, it causes not only significant resources consumption, but also lengthy execution time even for a small-scale graph. Consequently, offline partitioning algorithms are rarely adopted by large-scale graph-parallel systems for big learning. In contrast, online graph partitioning algorithms[8-9,16-17] aim to find a near-optimal graph placement by distributing vertices and edges with only limited graphs information. Due to the significant less partitioning time yet still-good graph placement, they have been widely adopted by almost all large-scale graph-parallel systems.

There are usually two mechanisms in online graph partitioning: edge-cut[16-17], which divides a graph by cutting cross-partition edges among sub-graphs; and vertex-cut[6,8-9], which partitions cross-partition vertices among sub-graphs. Generally speaking, edge-cut can evenly distribute vertices among multiple partitions, but may result in imbalanced computation and communication as well as high replication factor for skewed graphs[18-19]. In contrast, vertex-cut can evenly distribute edges, but may incur high communication cost among partitioned vertices.

PowerGraph employs several vertex-cut algorithms[8-9] to provide *edge* balanced partitions, because the workload of graph algorithms mainly depends on

the number of edges. Fig.2 illustrates the hash-based (random) vertex-cut to evenly assign edges to machines, which has a high replication factor (i.e., $\lambda = \#replicas/\#vertices$) but very simple implementation; to reduce replication factor, the greedy heuristic[8] is used to accumulate edges with a shared endpoint vertex on the same machine③; and currently the default graph partitioning algorithm in PowerGraph, Grid[9] vertex-cut, uses a 2-dimensional (2D) grid-based heuristic to reduce replication factor by constraining the location of edges. It should be noted that all heuristic vertex-cut algorithms must also maintain the load balance of partitions during assignment. In this case, random partitioning simply assigns edges by hashing the sum of source and target vertex-IDs, while Grid vertex-cut further specifies the location to only an intersection of shards of the source and target vertices. Further, the Oblivious greedy vertex-cut prioritizes the machine holding the endpoint vertices for placing edges.

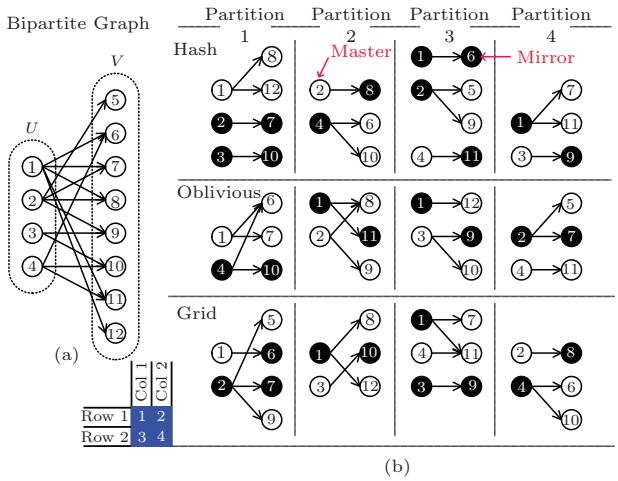


Fig.2. Comparison of various vertex-cut algorithms on a sample bipartite graph.

For example, in random vertex-cut (i.e., hash), the edge $(1, 8)$ is assigned to partition 1 as the sum of 1 and 8 divided by 4 (the total partition number) is 1. In Oblivious greedy vertex-cut, the edge $(1, 7)$ is assigned to partition 1 as the prior edge $(1, 6)$ has been assigned to partition 1. In Grid vertex-cut, the edge $(1, 8)$ is randomly assigned by Grid to one of the intersected partitions (2 and 3) according to the partitioning grid (Fig.2(b)). This is because vertex 1 is hashed to partition 1, which constrains the shards of vertex 1 to row 1 and column 1, while vertex 8 is hashed to partition 4, which constrains the shards of vertex 8 to row 2 and column 2. Thus, the resulting shards of the intersection are 2 and 3.

Unfortunately, all of partitioning algorithms result in suboptimal graph placement and the replication factor is high (2.00, 1.67 and 1.83 accordingly), due to the lack of awareness of the unique features in bipartite graphs. Our prior work, PowerLyra[6], also uses differentiated graph partitioning for skewed power-law graphs. However, it does not consider the special properties of bipartite graphs as well as data affinity during partitioning.

3 Challenges and Opportunities

All vertices in a bipartite graph can be partitioned into two disjoint subsets U and V, and each edge connects a vertex from U to one from V, as shown in Fig.2(a). Such special properties of bipartite graphs and the special requirement of MLDM algorithms impede existing graph partitioning algorithms to obtain a proper graph cut and performance. Here, we describe several observations from real world bipartite graphs and the characteristics of MLDM algorithms.

First, real world bipartite graphs for MLDM are usually imbalanced. This means that the size of two subsets in a bipartite graph is significantly skewed, even in the scale of several orders of magnitude. For example, there are only ten thousands of terms in Wikipedia, while the number of articles has exceeded four millions. The number of grades from students may be dozen times of the number of courses. As a concrete example, the SLS dataset, 10 years of grade points at a large state university, has 62 729 objects (e.g., students, instructors, and departments) and 1 748 122 scores (ratio: 27.87). This implies that a graph partitioning algorithm needs to employ differentiated mechanisms on vertices from different subsets.

Second, the computation load of many MLDM algorithms for bipartite graphs may also be skewed among vertices from the two subsets. For example, Stochastic Gradient Descent (SGD)[20], a collaborative filtering algorithm for recommendation systems, only calculates new cumulative sums of gradient updates for *user* vertices in each iteration, but none for *item* vertices. Therefore, an ideal graph partitioning algorithm should be able to discriminate the computation to one set of vertices and exploit the locality of computation

③ Currently, PowerGraph only retains Oblivious greedy vertex-cut, and coordinated greedy vertex-cut has been deprecated due to its excessive graph ingress time and buggy.

by avoiding an excessive replication of vertices.

Finally, the size of data associated with vertices from the two subsets can be significantly skewed. For example, to use the probabilistic inference on large astronomical images, the data of an *observation* vertex can reach several terabytes, while the latent *stellar* vertex has only very little data. If a graph partitioning algorithm distributes these vertices to random machines without awareness of data location, it may lead to excessive network traffic and significant delay in graph partitioning time. Further, the replication of these vertices and the synchronization among them may also cause significant memory and network pressure during computation. Consequently, it is critical for a graph partitioning algorithm to be built with data affinity support.

4 Bipartite-Oriented Graph Partitioning

The unique features of real world bipartite graphs and the associated MLDM algorithms demand a bipartite-aware online graph partitioning algorithm. BiCut is a new heuristic algorithm to partition bipartite graphs, by leveraging our observations. Based on the algorithm, we describe a refinement to further reduce replications and improve load balance, and show how BiCut supports data affinity for bipartite graphs with skewed data distribution to reduce network traffic.

4.1 Randomized Bipartite-Cut

Existing distributed graph partitioning algorithms use the same strategy to distribute all vertices of a bipartite graph, which cannot fully take advantage of the graph structures in bipartite graphs, especially for skewed graphs.

In a bipartite graph, two vertices connected by an edge must be from different disjointed subsets. This implies that *arbitrarily partitioning vertices belonging to the same subset may not introduce any replicas of vertices*, as there is no edge connecting them. Based on above observation, the new *bipartite-cut* algorithm applies a differentiated partitioning strategy to avoid the replication for vertices in one favorite subset and provide fully local access to adjacent edges. First, the vertices in this set are first-class citizens during partitioning, and are evenly assigned to machines with all adjacent edges at first. Such vertices contain no replicas. Then, the replicas of vertices in the other subset are created to construct a local graph on each machine. One replica of the vertex is randomly nominated as the master vertex, which coordinates the execution of all remaining replicas.

As shown in Fig.3(a), since the favorite subset is V, its vertices (from 5 to 12) with all edges are evenly assigned to four partitions without replication. The vertices in subset U (from 1 to 4) are replicated on demand in each partition. All edges of vertices in the favorite set can be accessed locally. By contrast, the vertex in the subset U has to rely on its mirrors for accessing its edges (e.g., vertex 1). Hence, BiCut only results in a replication factor of 1.5.

By default, BiCut will use the subset with more vertices as the favorite subset to reduce the replication factor. However, if the computation on one subset is extremely sensitive to locality, this subset can also be designated as the favorite subset to avoid network traffic in computation, since the edges are always from one subset to the other. As there is no extra step with high cost, the performance of BiCut should be comparable to random vertex-cut.

4.2 Greedy Bipartite-Cut

Our experiences show that the workload of graph-parallel system highly depends on the balance of edges. Unfortunately, randomized bipartite-cut only naturally provides the balance of favorite vertices. To remedy this issue, we propose a new greedy heuristic algorithm, namely Aweto, inspired by Ginger[6], a greedy heuristic hybrid-cut for natural graphs.

Aweto uses an additional round of edge exchange to exploit the similarity of neighbors between vertices in the favorite subset and the balance of edges in partitions. After the random assignment of the first round, each worker greedily re-assigns a favorite vertex v with all edges to partition i such that $\delta g(v, S_i) \geqslant \delta g(v, S_j)$, for all $j \in \{1, 2, \ldots, p\}$, where S_i is the current vertex set on partition i ($P = (S_1, S_2, \ldots, S_p)$). Here, $\delta g(v, S_i) = |N(v) \cap S_i| - B(|S_i|)$, where $N(v)$ denotes the set of neighbors of vertex v, $|S_i|$ denotes the current number of edges on partition i and $B(x) = |S_i|^{1/2}$. $B(x)$ is used to balance the edges re-assigned from current partition.

Because each partition maintains its own current vertex set S_i, the greedy-heuristic algorithm can independently execute on all machines without any communication. Further, the balance of edges re-assigned by each partition implies a global edge balance of partitions.

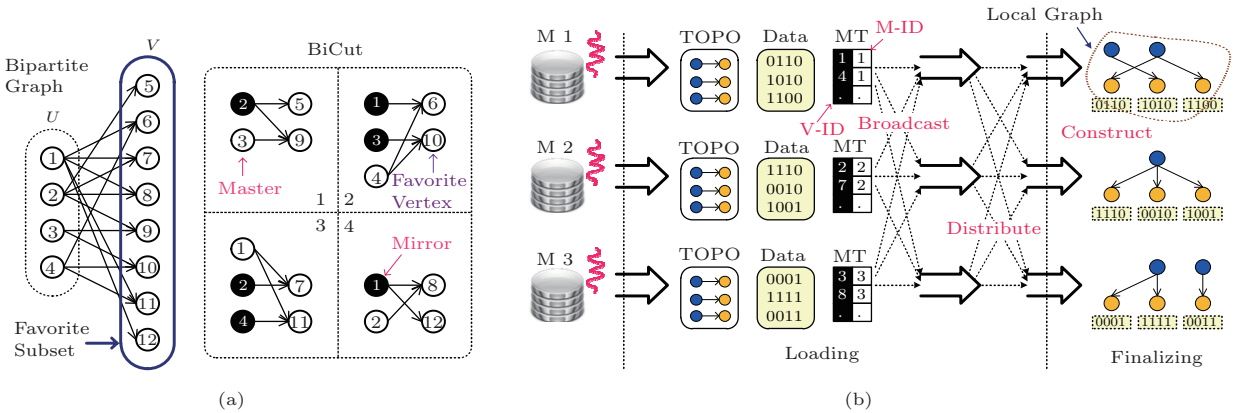


Fig.3. (a) Example of BiCut on a sample bipartite graph. (b) Execution flow of bipartite-cut with data affinity. TOPO: topology, MT: mapping table.

4.3 Data Affinity

A large amount of graph data may be stored on each machine of a cluster. For example, the output files of a MapReduce job will not be merged and stored to the local secondary storage. The modern distributed file system (e.g., HDFS) also splits files into multiple blocks and stores on multiple machines to improve availability and fault tolerance. Without data affinity, existing data partitioning only considers the affinity within the graph to reduce replication, which results in a large amount of data movement from the secondary storage of one machine to the main memory of other machines.

To reduce the movement of a huge amount of data and avoid the replication of favorite vertices, bipartite-cut is further extended to support data affinity. The favorite vertex with a huge amount of data is fixedly placed on machines holding their vertex data, and its edges are also re-assigned to those machines. In this way, the computation on favorite vertex is restricted to local machine without network traffic.

Fig.3(b) illustrates the execution flow of BiCut with data affinity. First, all topology information and the data of graph are *loaded* from local secondary storage to memory, and a local mapping table (MT) from favorite vertices to the current machine is generated on each machine. Then the local mapping table on each machine is *broadcast* to other machines to create a global mapping table on each machine. The edge *distribution* originally in graph loading is delayed to the end of exchanging mapping tables. Finally, the local graph is *constructed* as before by replicating vertices.

5 Performance Benefit of BiGraph

We have implemented BiGraph based on the latest GraphLab 2.2 (released in March 2014), which runs the PowerGraph engine④. BiCut and Aweto are implemented as separate graph-cut modules for GraphLab and thus can transparently run all existing toolkits in GraphLab.

We evaluate BiGraph against the default graph partitioning algorithms, Grid[9], and Oblivious greedy[8], on GraphLab framework using three typical bipartite graph algorithms: Singular Value Decomposition (SVD), Alternating Least Squares (ALS), and Stochastic Gradient Descent (SGD).

5.1 Experimental Setup

We use two clusters with different configurations in our experiments. An in-house 6-machine multicore cluster has a total of 144 CPU cores, in which each machine has a 24-core AMD processor, 64GB RAM and 1 GigE network port. An EC2-like 48-machine multicore cluster has a total of 192 CPU cores, in which each machine has a 4-core AMD processor, 10GB RAM and 1 GigE network ports. We run NFS on the same cluster as the underlying storage layer.

Table 1 lists a collection of bipartite graphs used in our experiments. They are from Stanford Large Network Dataset Collection⑤ and The University of Florida Sparse Matrix Collection⑥. The former three bipartite graphs are balanced, and the ratios of $|U|/|V|$ are from 1.03 to 1.75. These can be regarded as the

④ GraphLab prior to 2.1 runs the distributed GraphLab engine.

⑤ http://snap.stanford.edu/data/, Nov. 2014.

⑥ http://www.cise.ufl.edu/research/sparse/matrices/index.html, Nov. 2014.

worst case for bipartite-cut due to their balanced distribution. On the contrary, the latter three bipartite graphs are relative skewed, and the ratios of $|U|/|V|$ are from 8.65 to 27.87. These are more suitable for bipartite-cut. The last Netflix prize dataset⑦ is used as a building block to simulate large datasets with various sizes for the weak scalability experiment (in Subsection 5.4). All vertices in the subset U and edges are duplicated to scale the dataset[21].

Table 1. Collection of Real World Bipartite Graphs

Graphs	$\|U\|$(K)	$\|V\|$(K)	$\|E\|$(M)	$\|U\|/\|V\|$
LJournal (LJ)	4 489	4 308	69.0	1.04
AS-skitter (AS)	1 696	967	11.1	1.75
GWeb (GW)	739	715	5.1	1.03
RUCCI (RUC)	1 978	110	7.8	18.00
SLS	1 748	63	6.8	27.87
ESOC (ESO)	327	38	6.0	8.65
Netflix	480	17	100.0	27.02

In Fig.4, we first compare the replication factor using various vertex-cut algorithms for real world bipartite graphs. Grid and Oblivious present close replication factor in such cases. For skewed bipartite graphs, BiCut and Aweto can significantly reduce up to 63% (from 59%) and 67% (from 59%) vertex replication on 6 partitions respectively, and up to 73% (from 40%) and 80% (from 54%) vertex replication on 48 partitions respectively. For balanced bipartite graphs, BiCut and Aweto still show notable improvement against Grid and Oblivious, especially for 48 partitions reducing up to 56% vertex replication.

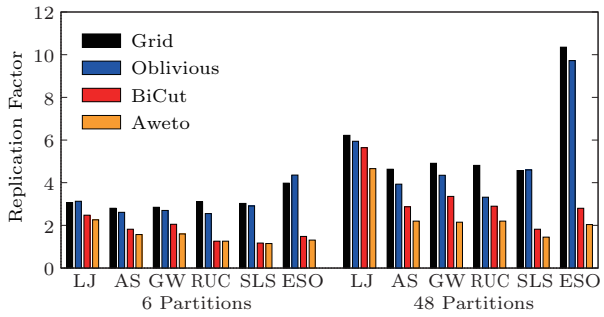


Fig.4. Replication factor for real world bipartite graphs using different partitioning algorithms on 6 and 48 partitions.

5.2 Overall Performance

Fig.5 summarizes the normalized speedup of BiCut and Aweto compared to Grid and Oblivious graph partitioning algorithms on the computation phase with different algorithms and graphs on two clusters. Since all partitioning algorithms use the same execution engine in the computation phase, the speedup highly depends on the reduction of replication factors, which dominates the communication cost. On the in-house 6-machine cluster, BiCut significantly outperforms Grid partitioning by up to 15.65X (from 5.28X) and 3.73X (from 2.33X) for ALS and SGD accordingly for skewed bipartite graphs. For balanced bipartite graphs, BiCut still provides a notable speedup by 1.24X, 1.39X and 1.17X for LJ, AS and GW graphs on SVD respectively. Aweto can further reduce replication factor and provide up to 38% additional improvement. On the EC2-like 48-machine cluster, though the improvement is weaken due to sequential operations in MLDM algorithms and message batching, BiCut and Aweto still provide moderate speedup by up to 3.67X and 5.43X accordingly.

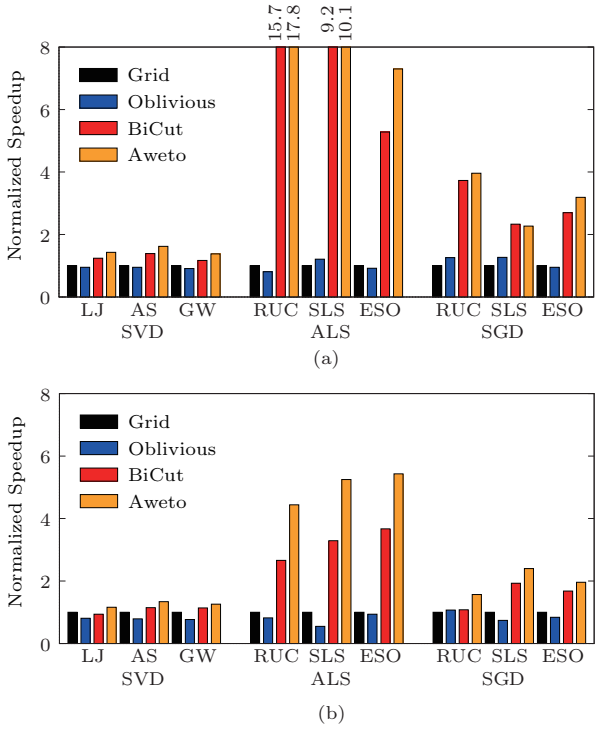


Fig.5. Comparison of overall graph computation performance using various partitioning algorithms for SVD, ALS and SGD with real world graphs on the (a) 6-machine cluster and (b) 48-machine cluster.

Fig.6 illustrates the graph partitioning performance of BiCut and Aweto against Grid and Oblivious, including loading and finalizing time. BiCut outperforms Grid by up to 2.63X and 2.47X for two clusters respectively due to lower replication factor, which reduces the

⑦ http://www.netflixprize.com/, Nov. 2014.

cost for data movement and replication construction. In the worst cases (i.e., for balanced bipartite graphs), Aweto is lightly slower than Grid because of additional edge exchange. However, the increase of ingress time is trivial compared to the improvement of computation time, just ranging from 1.8% to 10.8% and from 3.8% to 5.1% for 6 and 48 machines respectively.

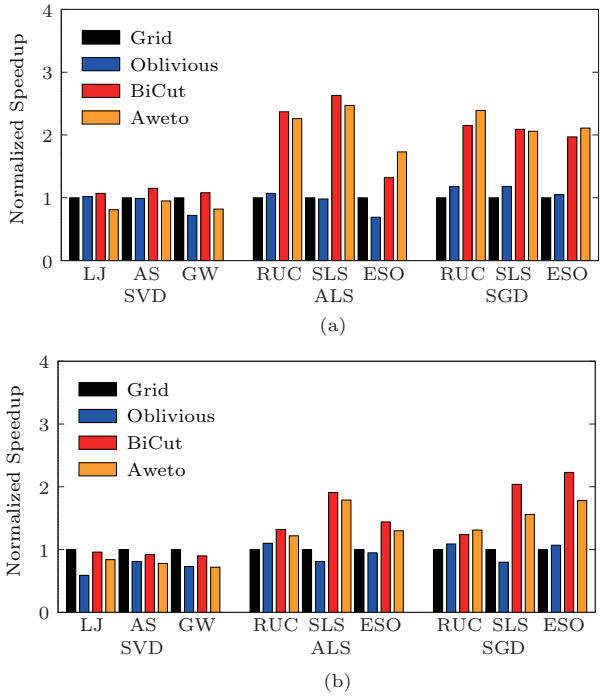


Fig.6. Comparison of overall graph partitioning performance using various partitioning algorithms for SVD, ALS and SGD with real world graphs on the (a) 6-machine cluster and (b) 48-machine cluster.

5.3 Network Traffic Reduction

Since the major source of speedup is from reducing network traffic in the partitioning and computation phases, we compare the total network traffic of BiCut and Aweto against Grid. As shown in Fig.7, the percent of network traffic reduction can perfectly match the performance improvement. On the in-house 6-machine cluster, BiCut and Aweto can reduce up to 96% (from 78%) and 45% (from 22%) network traffic against Grid for skewed and balanced bipartite graphs accordingly. On the EC2-like 48-machine cluster, BiCut and Aweto still can reduce up to 90% (from 33%) and 43% (from 11%) network traffic in such cases.

5.4 Scalability

Fig.8 shows that BiCut has better weak scalability than Grid and Oblivious on our in-house 6-machine cluster, and keeps the improvement with increasing graph size. For the increase of graph size from 100 to 400 million edges, BiCut and Aweto outperform Grid partitioning by up to 2.27X (from 1.89X). Note that using Grid partitioning even cannot scale past 400 million edges on a 6-machine cluster with 144 CPU cores and 384GB memory due to exhausting memory. Oblivious can run on 800 million edges due to relative better replication factor, but just provides a close performance to Grid and also fails in larger input. While using BiCut and Aweto partitioning can scale well with even more than 1.6 billion edges.

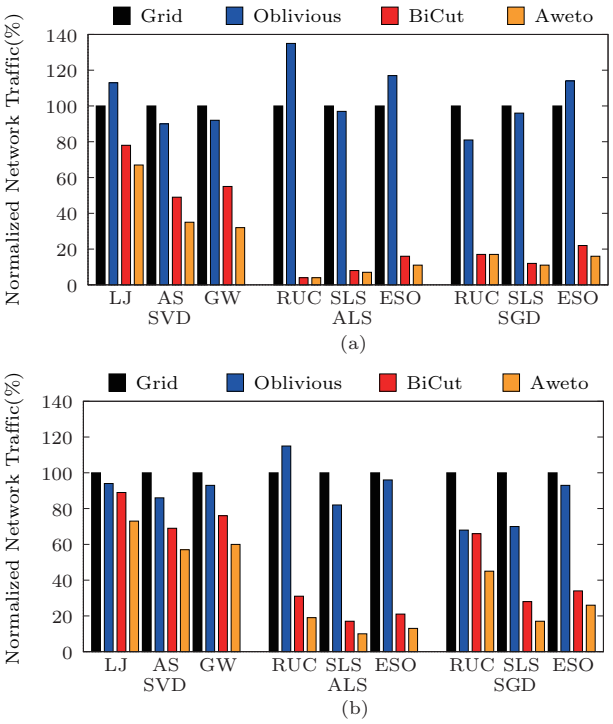


Fig.7. Percent of network traffic reduction over Grid on the (a) 6-machine cluster and (b) 48-machine cluster.

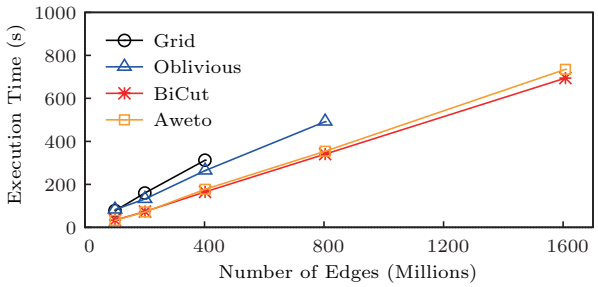


Fig.8. Comparison of the performance on various partitioning algorithms with increasing graph size.

5.5 Benefit of Data Affinity Support

To demonstrate the effectiveness of data affinity extension, we use an algorithm to calculate the occur-

rences of a user-defined keyword touched by users on a collection of web pages at fixed intervals. The application models users and web pages as two subsets of vertices, and the access operations as the edges from users to web pages. In our experiment, the input graph has 4 000 users and 84 000 web pages, and the vertex data of the users and web pages are the occurrences of the keywords (4-byte integer) and the content of a page (dozens to several hundreds of kilobytes) respectively. All web pages are from Wikipedia (about 4.82GB) and separately stored on the local disk of each machine of cluster.

For this graph, Grid and Oblivious result in a replication factor of 3.55 and 3.06, and cause about 4.23GB and 3.97GB network traffic respectively, due to a large amount of data movement for web page vertices. However, BiCut has only a replication factor of 1.23 and causes unbelievable 1.43MB network traffic only from exchanging mapping table and dispatching of user vertices. This transforms to a performance speedup of 8.35X and 6.51X (6.7s vs 55.7s and 43.4s respectively) over Grid and Oblivious partitioning algorithms respectively. It should be noted that, without data affinity support, the graph computation phase may also result in a large amount of data movement if the vertex data is modified.

6 Conclusions

In this paper, we identified the main issues with existing graph partitioning algorithms in large-scale graph analytics framework for bipartite graphs and the related MLDM algorithms. A new set of graph partitioning algorithms, called BiGraph, leveraged three key observations from bipartite graphs. BiCut employs a differentiated partitioning strategy to minimize the replication of vertices, and also exploits the locality for all vertices from the favorite subset of a bipartite graphs. Based on BiCut, a new greedy heuristic algorithm, called Aweto, was provided to optimize partition by exploiting the similarity of neighbors and load balance. In addition, based on the observation of skewed distribution of data size between two subsets, BiGraph was further refined with the support of data affinity to minimize network traffic. Our evaluation showed that BiGraph is effective in not only significantly reducing network traffic, but also resulting in a notable performance boost of graph processing.

References

[1] Malewicz G, Austern M H, Bik A J, Dehnert J C, Horn I, Leiser N, Czajkowski G. Pregel: A system for large-scale graph processing. In *Proc. the 2010 ACM SIGMOD International Conference on Management of Data*, June 2010, pp.135–146.

[2] Dhillon I S. Co-clustering documents and words using bipartite spectral graph partitioning. In *Proc. the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug. 2001, pp.269–274.

[3] Zha H, He X, Ding C, Simon H, Gu M. Bipartite graph partitioning and data clustering. In *Proc. the 10th International Conference on Information and Knowledge Management*, August 2001, pp.25–32.

[4] Gao B, Liu T Y, Zheng X, Cheng Q S, Ma W Y. Consistent bipartite graph co-partitioning for star-structured high-order heterogeneous data co-clustering. In *Proc. the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, August 2005, pp.41–50.

[5] Gao B, Liu T Y, Feng G, Qin T, Cheng Q S, Ma W Y. Hierarchical taxonomy preparation for text categorization using consistent bipartite spectral graph copartitioning. *IEEE Transactions on Knowledge and Data Engineering*, 2005, 17(9): 1263–1273.

[6] Chen R, Shi J, Chen Y, Guan H, Zang B, Chen H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. Technical Report, IPADSTR-2013-001, Shanghai Jiao Tong University, 2013.

[7] Low Y, Bickson D, Gonzalez J, Guestrin C, Kyrola A, Hellerstein J M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 2012, 5(8): 716–727.

[8] Gonzalez J E, Low Y, Gu H, Bickson D, Guestrin C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. the 10th USENIX Symp. Operating Systems Design and Implementation*, October 2012, pp.17–30.

[9] Jain N, Liao G, Willke T L. Graphbuilder: Scalable graph ETL framework. In *Proc. the 1st International Workshop on Graph Data Management Experiences and Systems*, June 2013, Article No.4.

[10] Chen R, Shi J, Zang B, Guan H. Bipartite-oriented distributed graph partitioning for big learning. In *Proc. the 5th Asia-Pacific Workshop on Systems*, June 2014, pp.14:1–14:7.

[11] Chen R, Ding X, Wang P, Chen H, Zang B, Guan H. Computation and communication efficient graph processing with distributed immutable view. In *Proc. the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, June 2014, pp.215–226.

[12] Brin S, Page L. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 1998, 30(1): 107–117.

[13] Schloegel K, Karypis G, Kumar V. Parallel multilevel algorithms for multi-constraint graph partitioning. In *Proc. the 6th Int. Euro-Par Conf. Parallel Processing*, August 2000, pp.296–310.

[14] Ng A Y, Jordan M I, Weiss Y. On spectral clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems*, Dietterich T G, Becker S, Ghahramani Z (eds), MIT Press, 2002, pp.849–856.

[15] LÜcking T, Monien B, Elsässer R. New spectral bounds on k-partitioning of graphs. In *Proc. the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, July 2001, pp.255–262.

[16] Stanton I, Kliot G. Streaming graph partitioning for large distributed graphs. In *Proc. the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, August 2012, pp.1222–1230.

[17] Tsourakakis C, Gkantsidis C, Radunovic B, Vojnovic M. FENNEL: Streaming graph partitioning for massive scale graphs. In *Proc. the 7th ACM International Conference on Web Search and Data Mining*, February 2014, pp.333–342.

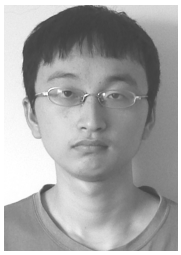
[18] Abou-Rjeili A, Karypis G. Multilevel algorithms for partitioning power-law graphs. In *Proc. the 20th International Parallel and Distributed Processing Symposium*, April 2006, p.124.

[19] Leskovec J, Lang K J, Dasgupta A, Mahoney M W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 2009, 6(1): 29–123.

[20] Koren Y, Bell R, Volinsky C. Matrix factorization techniques for recommender systems. *Computer*, 2009, 42(8): 30–37.

[21] Kumar A, Beutel A, Ho Q, Xing E P. Fugue: Slow-worker-agnostic distributed learning for big models on big data. In *Proc. the 17th International Conference on Artificial Intelligence and Statistics*, April 2014, pp.531–539.

Rong Chen received his B.S., M.S., and Ph.D. degrees in computer science from Fudan University, Shanghai, in 2004, 2007, and 2011, respectively. He is currently an assistant professor of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China. He is a member of CCF, ACM, and IEEE. His current research interests include, but are not limited to, distributed systems, operating systems and virtualization.



Jia-Xin Shi received his B.S. degree in computer science from Shanghai Jiao Tong University, China, in 2014. He is currently a graduate student of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University. His current research interests include large-scale graph-parallel processing, parallel and distributed processing.

Hai-Bo Chen received his B.S. and Ph.D. degrees in computer science from Fudan University, Shanghai, in 2004 and 2009, respectively. He is currently a professor of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China. He is a senior member of CCF, and a member of ACM and IEEE. His research interests include software evolution, system software, and computer architecture.

Bin-Yu Zang received his Ph.D. degree in computer science from Fudan University, Shanghai, in 1999. He is currently a professor and the director of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China. He is a senior member of CCF, and a member of ACM and IEEE. His research interests include compilers, computer architecture, and systems software.