Boundary Vertices Sensitive Vertex-cut Partitioning Algorithm

Asad Feroz Ali[1], Dr. Syed Saif ur Rahman[2]

[1,2] *Shaheed Zulfikar Ali Bhutto Institute of Science and Technology (SZABIST) Karachi, Pakistan*

[1]`AsadFerozAli@gmail.com`
[2]`saif.rahman@szabist.edu.pk`

Abstract - Graph Partitioning is one of the favorite research topics among researchers since the 70s. It attracts a diverse group of researchers from various fields such as engineering, science and mathematics. In the last decade, the graphs have increased in size to billions of vertices. Despite the fact that storage devices have become cheaper, processing these huge spanning graphs is not possible for a single machine. This call for the need of partitioning the graph so a group of machines can perform various parallel calculations on them which would save time and produce quick results. The research problem is that the ratio of boundary vertices to interior vertices increases with the increase in number of partitions for existing partitioning techniques available. To address this issue, the random edge selection method of Graph Lab algorithm was replaced with four suggested edge sorting techniques. The results were compared with the random edge selection method of Graph Lab using various performance parameters.

Keywords - Graph Partitioning, Vertices, Graph Lab

I. INTRODUCTION

It is becoming progressively important to retrieve information and knowledge from graph datasets [1] as the graph data sets are increasing in extent [2]. The processing of these huge graph datasets in different researches is of great significance and value specifically in the field of computing and biology for example processing graph datasets of social networks [3] and that of protein interaction [1]. The calculations and computations that need to be performed on these data sets can vary from identification of associations in protein in the field of biology to performing PageRank [3] and status updates on webpages and social networks respectively. Performing complex calculations on these huge datasets would not be possible for a single system because of memory and time restrictions [4]. Therefore, these graphs can be partitioned and sent for calculations on to different machines on a network so each of those machines could perform the processes on their portion of the graph in parallel and return the result [5]. In graph theory, Graph partitioning is classified as a NP (Non-deterministic Polynomial-time)-hard problem [1]. The parallel processing of these graphs requires every edge or vertex to be processed in context to their surrounding neighbors [6]. Hence, it is vital to retain the locality of information when the graphs are being partitioned over multiple machines [7]. When deciding over the pros and cons of edge and vertex cut partitioning, it is important to know the real world graphs are usually power-law graphs and edge cut partitioning does not yield good performance in comparison to vertex-cut partitioning which divides the edges of a graph into equal size clusters [3]. The vertices that hold the endpoints of an edge are placed in the same cluster as the edge [7].

The communication among machines in a distributed graph processing environment is dependent on the number of boundary vertices. To minimize the communication, this study was conducted with the goal to minimize the number of vertices at the boundary of the partitions by implementing boundary vertices sensitive vertex-cut partitioning algorithm. This research was an empirical research and the focus was to identify and implement a solution by altering existing balanced p-way vertex-cut and greedy cut partitioning algorithms used by Power graph.

A. Graph Partitioning

A graph partitioning problem is to cut a graph into two or more good pieces. In graph theory, a cut is the partition of the vertices of a graph into two disjoint subsets. Any cut determines a cut-set, the set of edges that have one endpoint in each subset of the partition. These edges are said to cross the cut [1].

B. Edge Cut

An edge cut assigns vertices to the partitions. Edge cut optimization aims to reduce the cross communication. In assigning vertices to partitions, one strives to optimize the assignment such that frequently co-traversed vertices are hosted on the same machine [1]. Assume vertex A is assigned to machine 1 and vertex B is assigned to machine 2. An edge between the vertices is called a cut edge because its end points are hosted on separate machines. Traversing this edge as part of a graph query requires communication between the machines which slows down query processing [3] as shown in figure 1.

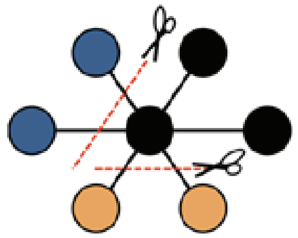

Fig. (1). Edge Cut

C. Vertex Cut

A vertex cut assigns edges to partitions. Cutting a vertex means storing a subset of that vertex's adjacency list on each partition in the graph. Vertex cuts address the hotspot issue caused by vertices with a large number of incident edges [1] as shown in figure 2.

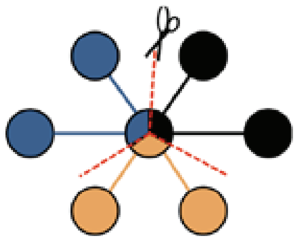

Fig. (2). Vertex Cut

D. Boundary vertices

Boundary vertices are those vertices which are replicated i.e. they are present in both the partitions during a vertex cut. The greater the number of boundary vertices, the more communication will take place across the partitions for the replicated vertices so the key motive in graph partitioning is to reduce the number of boundary vertices [1].

II. RELATED WORK

Graph partitioning is an important pre-processing step for the processing of distributed graphs [8]. Partitioning is used to distribute partitions of a graph across memory for faster processing [9]. Graph partitioning is a NP-Complete problem.

The Graph partitioning techniques attempts to ensure the following characteristics to create good quality partitions:

- Every partition should be allotted similar number of nodes [7]
- Minimizing the number of cross-partition edges in every partition [7]
- Attain spatial locality for processing the graph [7]
- Decrease memory contention when graph processing is carried out on shared memory [7]
- Make the most of parallelism [10]
- Reduce communication/latency [3]

Pregel [6] is a distributed large-scale graph processing system that uses vertex-oriented abstraction. It uses message passing approach for parallel processing and it is a bulk synchronous message passing abstraction. Pregel uses hash-based (random) partitioning. GraphLab [4, 5] abstraction expresses asynchronous, dynamic, graph-parallel computation. Shared-memory approach is used for parallel processing. Hence, it is a sequential shared memory abstraction. Graph Lab allows system to choose when and how to move program state. It also ensures serializability and data consistency using techniques such as pipelined locking and data versioning. It achieves fault-tolerance using Chandy-Lamport snapshot algorithm.

GraphLab uses distributed FIFO (First in, First out) and priority scheduling and it is implemented using C++. PowerGraph [3] clearly factors computation over edges rather than vertices. It follows the GraphLab data-graph and shared-memory view of computation which makes the movement of information hidden to users. PowerGraph follows the commutative, associative gather concept from Pregel. The partitioning approaches it uses are balanced p-way Vertex Cut and Greedy Vertex-cut. GraphLab PowerGraph partitions the graph across machines using a "vertex separator" approach where edges are assigned to machines while vertices can span across multiple machines. GraphLab PowerGraph automatically distributes computation across different nodes and all the computations run in the memory. PowerGraph uses two approaches for vertex-cut namely Random p-way vertex cuts and Greedy Vertex-cuts.

Random p-way vertex cut selects the edges randomly from the dataset and then assigns it to any of the partitions randomly. This can result in any number of edges being assigned to one partition and remaining to other partitions. Greedy vertex-cut uses sequential greedy heuristics edge placement rules for edge. It places the next edge on the machine that minimizes the conditional expected replication factor.

III. PROPOSED METHODS

This section describes the different vertices sorting methods that were used to test on different datasets to check for their performance. The different sorting approaches of the edges used were as follows:

- Low Degree Vertices to High Degree Vertices
- High Degree Vertices to Low Degree Vertices
- Low Degree Vertices to High Degree Vertices Round Robin Approach
- High Degree Vertices to Low Degree Vertices Round Robin Approach

Natural Datasets [2] were used to compare the performance of different sorting methods mentioned above to the random approach of Graph Lab. In each of the above mentioned approaches, the four cases of the Graph Lab algorithm were used to assign the edges to different partitions once the input data was sorted using a particular approach.

The ratio of boundary vertices to total vertices was calculated for each of those methods and was compared with the random method of Graph Lab. This ratio is the major factor in deciding which partitioning method performs well. The lower the ratio of boundary vertices to total vertices the better the method it is because the lower ratio indicates that the replicated nodes are less. In addition to this, the number of edges should also be considered. The partitioning method should allot equal number of edges in the partitions. If the assignment of edges is unequal i.e. if one of the partitions has more edges to process than the other partition than that partition would take more time to process the edges. As a result it would be an uneven distribution of workload.

A. Low Degree Vertices to High Degree Vertices

In this approach, the edges were sorted in a low degree to high degree order. To sort the edges in order of low degree to high degree vertices, all the vertexes of the graph were noted and their degrees were calculated. This data was kept in a table which had a list of vertex and their corresponding degrees. The degree column was sorted in ascending order to get the table in a form where it showed the vertexes in low to high degree order. The next step was to read the vertex from the table one by one and find all edges of the particular vertex from the data and store it in another table "Sorted Edge List" which had the edges sorted in order of increasing degree of the vertex. Once this was done, the nodes were read one by one from the "Sorted Edge List" table and processed by the four conditions of the Graph Lab algorithm and assigned to the partitions i.e. partition 1 or partition 2.

Pseudo Code

A is a data frame containing the edges of the graph. AF_SORTED is the data frame that contains the list of vertex with their degrees sorted in an ascending order. SORTED_EDGE_LIST is a data frame that contains the edge lists from the data frame A sorted in increasing order of degree. NROW is the number of vertexes present in A. Both A and SORTED_EDGE_LIST contain two columns V1 and V2 which identify the 2 vertex that forms an edge. AF_SORTED contains 2 columns VERTEX_NAME and DEGREE. An additional column named USED is added to AF_SORTED. This column is used for the loop to track how many times a vertex has been used. Every time a vertex is used, the USED column is incremented by one. When a vertex's degree is equal to the value in the USED column the loop moves to the next vertex.

```
1. Create a vertex list with their degrees sorted in ascending
order and store it in AF_SORTED. Add a column USED to
AF_SORTED.
2. [Initialize] Set NROW = Number of Vertex in AF_SORTED and set SORTED_EDGE_LIST as an empty data frame.
3. [Initialize Counter] Set I=1
4. Repeat Steps below until I = NROW
    a) FREQ_VAR = Degree of the current vertex in
       AF_SORTED, VAR1_VALUE = Current Vertex in
       the AF_SORTED vertex column.
    b) [Initialize Counter] Set J=1
    c) Repeat Steps below until J = FREQ_VAR
        1. Find the first edge from A for the vertex
           present in VAR1_VALUE
        2. Add that edge to the SORTED_EDGE_LIST
           data frame
        3. Remove the edge from the data frame A
        4. Increase the value of the USED column in
           AF_SORTED by 1
        5. [Increment Counter] Set J = J+1
           [End of Step c inner loop]
       [Increment Counter] Set I = I+1
[End of Step 4 outer loop]
```

B. High Degree Vertices to Low Degree Vertices

In this approach, edges were sorted in a high degree to low degree order using a same approach as described in the previous method.

Pseudo Code

```
1. Create a vertex list with their degrees sorted in
descending order and store it in AF_SORTED. Add a
column USED to AF_SORTED.
2. [Initialize] Set NROW = Number of Vertex in
AF_SORTED and set SORTED_EDGE_LIST as an
empty data frame.
3. [Initialize Counter] Set I=1
4. Repeat Steps below until I = NROW
        a) FREQ_VAR = Degree of the current vertex in
           AF_SORTED, VAR1_VALUE = Current Vertex
           in the AF_SORTED vertex column.
        b) [Initialize Counter] Set J=1
        c) Repeat Steps below until J = FREQ_VAR
            1. Find the first edge from A for the vertex
               present in VAR1_VALUE
            2. Add that edge to the
               SORTED_EDGE_LIST data frame
            3. Remove the edge from the data frame A
            4. Increase the value of the USED column in
               AF_SORTED by 1
            5. [Increment Counter] Set J = J+1
            [End of Step c inner loop]
        [Increment Counter] Set I = I+1
[End of Step 4 outer loop]
```

C. Low Degree Vertices to High Degree Vertices – Round Robin Approach

Low degree to high degree approach was explained earlier but in round robin approach, every other vertex was selected by increasing order of degree. Therefore, an edge was selected from a low degree vertex then rather than selecting other edges for the same vertex like it was done earlier, the next edge was selected from the next vertex which had a higher degree than the previous vertex. This was done until reached to the bottom of the table which has the highest degree vertex and last one is select from that. Then this whole process was repeated until all the edges had been selected. In this approach, the edges of the graph were first read by the program and then only relevant columns of the dataset were kept such as edges lists and remaining data was removed. As a result, the dataset with edges of the graph was acquired. To sort the edges; first of all, the vertices of the graph were noted and their degrees were calculated. This data was kept in a table which had a list of vertex and their corresponding degrees. In this way the degree column was sorted in increasing order to get the table in a form where it showed the vertexes in low to high degree. The next step was to read the vertex from the table one by one in increasing order of degree and to find the first edge of the particular vertex from the data and store it in the table "Sorted Edge List" which had the edges sorted in order of increasing degree with round robin approach. Then the consecutive rounds of the same approach were continued until all the vertices had been covered according to their degrees. Once this was done, the nodes were read one by one from the "Sorted Edge List" table and processed by the four conditions of the Graph Lab algorithm and assigned to the partitions i.e. partition 1 or partition 2

Pseudo Code

MAX_FREQ is the value of the maximum degree that a vertex has in the table AF_SORTED.

```
1. Create a vertex list with their degrees sorted in ascending order and store it in AF_SORTED. Add a column USED to AF_SORTED.
2. [Initialize] Set NROW = Number of Vertex in AF_SORTED, set SORTED_EDGE_LIST as an empty data frame and MAX_FREQ as the maximum value of degree in the AF_SORTED table
3. [Initialize Counter] Set I=1
4. Repeat Steps below until I = MAX_FREQ
    a) [Initialize Counter] Set J=1
    b) FREQ_VAR = Degree of the current vertex in AF_SORTED, VAR1_VALUE = Current Vertex in the AF_SORTED vertex column.
    c) Repeat Steps below until J = NROW
            1. If the value of USED =DEGREE for current vertex, then
            Skip the remaining commands and move to step 6
            [End of If Structure]
            2. Find the first edge from A for the vertex present in VAR1_VALUE
            3. Add that edge to the SORTED_EDGE_LIST data frame
            4. Remove the edge from the data frame A
            5. Increase the value of the USED column in AF_SORTED by 1
            6. [Increment Counter] Set J = J+1
            [End of Step c inner loop]
        [Increment Counter] Set I = I+1
[End of Step 4 outer loop]
```

D. High Degree Vertices to Low Degree Vertices – Round Robin Approach

In this approach, all the steps were same as in the previous round robin approach but the degree table was sorted in high to low degree instead of low to high degree as in previous method. Then the round robin approach was carried out as explained in the previous section.

Pseudo Code

```
1. Create a vertex list with their degrees sorted in descending order and store it in AF_SORTED. Add a column USED to AF_SORTED.
2. [Initialize] Set NROW = Number of Vertex in AF_SORTED, set SORTED_EDGE_LIST as an empty data frame and MAX_FREQ as the maximum value of degree in the AF_SORTED table
3. [Initialize Counter] Set I=1
4. Repeat Steps below until I = MAX_FREQ
    a) [Initialize Counter] Set J=1
    b) FREQ_VAR = Degree of the current vertex in AF_SORTED, VAR1_VALUE = Current Vertex in the AF_SORTED vertex column.
    c) Repeat Steps below until J = NROW
        1. If the value of USED =DEGREE for current vertex, then
        Skip the remaining commands and move to step 6
        [End of If Structure]
        2. Find the first edge from A for the vertex present in VAR1_VALUE
        3. Add that edge to the SORTED_EDGE_LIST
        data frame
        4. Remove the edge from the data frame A
        5. Increase the value of the USED column in
        AF_SORTED by 1
        6. [Increment Counter] Set J = J+1
        [End of Step c inner loop]
    [Increment Counter] Set I = I+1
[End of Step 4 outer loop]
```

IV. PERFORMANCE PARAMETERS

The different parameters used to calculate the performance of the sorting methods were:

- Boundary Vertices to Total Vertices Ratio
- Ratio of edges in each partitions
- Ratio of nodes in each partition

A. Boundary Vertices to Total Vertices Ratio

The boundary vertices were calculated by finding the same vertices in the two partitions. The ratio is then calculated by dividing boundary vertices by the total vertices present in the dataset. The method which results in the lowest ratio would be considered better than other methods.

B. Ratio of edges in each partitions

This ratio was calculated by dividing the number of edges contained in one partition to the number of edges present in the other partition. A ratio closer to 1 would indicate that both partition contained equal number of edges i.e. edge distribution was even. The method which results in a ratio closer to 1 would be considered better than the other methods.

C. Ratio of vertices in each partition

To calculate this parameter, the individual vertices present in each partition were counted and divided by the number of vertices present in the other partition. A ratio of 1 would indicate that vertices were evenly distributed in the two partitions. The method which resulted in a ratio closer to 1 would be considered better than the other methods.

V. EXPERIMENTAL EVALUATION

To calculate the performance of the methods, two Natural datasets were used from Large Natural Dataset Collection by Stanford University. They were snapshots of the Gnutella peer-to-peer file sharing network taken on two different days of August 2002 i.e. 5th and 6th. The numbers of vertices or nodes contained were 8,846 and 8,717 respectively whereas the edges they contained were 31,839 and 31,525 respectively. The datasets were imported in a program created in R language and all the computation and sorting methods were tested using that program. The program was used to calculate performance parameter of the Graph Lab's random method of edge selection to the four proposed methods for each of the two datasets. The pseudo code for that program has been described in detail in the earlier section.

VI. RESULTS

After obtaining the list of edges assigned to different partitions by the program i.e. both random methods from Graph Lab and proposed methods, the performance parameter values for both the data sets were evaluated and plotted individually for each parameter for both the datasets for all the five methods.

A.Boundary Vertices to Total Vertices Ratio

The boundary vertices to total vertices ratio was calculated by finding the number of edges that were duplicated in both the partitions and then dividing it by total number of vertices. The purpose of calculating this parameter was to identify which method resulted in the least ratio.

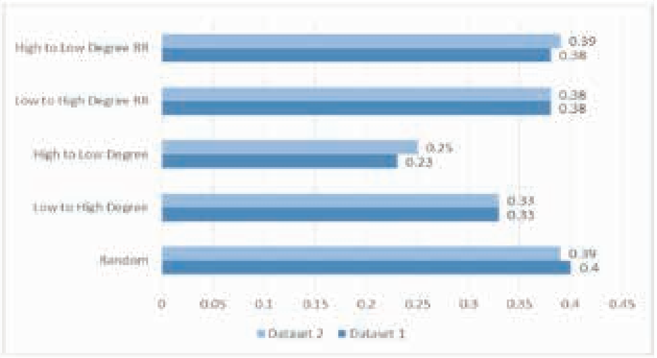

Fig. (3). Bar plot for boundary vertices to total vertices ratio

The figure 3 above shows that the ratios achieved by random method were the highest which is 0.39 and 0.4 whereas the ratios achieved by high to low degree method were the lowest which is 0.23 and 0.25 followed by Low to High degree method. The results of round robin approach of both the low to high degree method and high to low degree method were close to each other with values of 0.38 and 0.39 and both of them varied slightly. Apart from that the difference in values achieved for both the datasets is marginal.

B. Ratio of Edges in Each Partition

Once the partitioning methods divided the edges in the two partitions, the numbers of rows presented in each partition were divided to find this required ratio. This ratio is important because it shows the equality of partition i.e. how many edges each of the partitions received for processing. The results from the bar plot shows that none of the methods resulted in a ratio of 1. Random Method gave the ratio close to 1 but even then the ratio is 2.3 which mean that one of the partitions received 2.3 times more edges to process than the other partition. Among the proposed methods, High to Low Degree Round Robin method gave a result closer to the random method which is 2.4 followed by Low to High Degree Round Robin method which gave a ratio of 2.6. The High to Low Degree method approaches gave the highest ratio which is why the boundary vertices for this method were the least as highlighted in the previous graph.

The results from the bar plot in figure 4 shows that none of the methods reached a ratio of 1. Random method gave the ratio close to 1 but even then the ratio is 2.3 which mean that one of the partitions received 2.3 times more edges to process than the other partition. Among the proposed methods, High to Low Degree Round Robin method gave a result closer to the random method which is 2.4 followed by Low to High Degree Round Robin method which gave a ratio of 2.6.

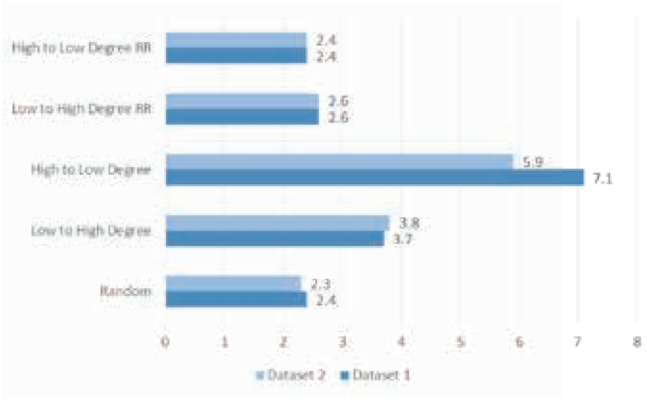

Fig. (4). Bar plot for ratio of Edges distribution

The High to Low Degree method gave the highest ratio which is why the boundary vertices for this method were the least. Since the edge distribution by this method is almost five to seven times higher among the two partitions, the partition with the lower number of edges resulted in less replicated nodes hence less boundary vertices.

C. Ratio of Vertices in Each Partition

When the partitioning method divided the edges into two partitions, unique vertices present among all the edges in each partition were divided with each other to find the distribution of vertices among the partitions. This is also an import performance parameter because the closer the ratio is to 1 the better the distribution of vertices.

The results from the bar plot in figure 5 shows that none of the method resulted in a ratio of 1. The closest any method reached to 1 was random partitioning method which resulted in a ratio of 1.4 for both the datasets. All the four proposed

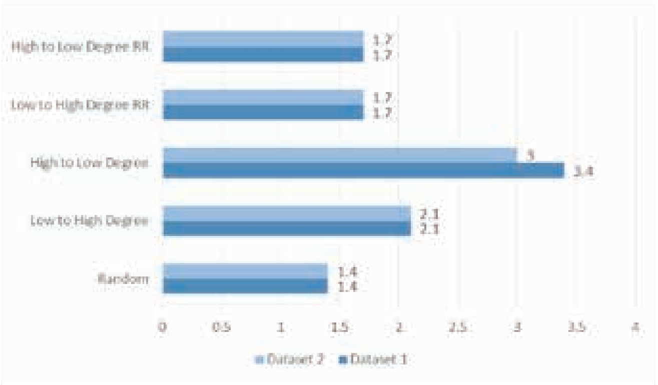

Fig. (5). Bar plot for ratio of Vertices Distribution

methods remained above that ratio with the round robin approach of high to low degree and low to high degree gave similar results for both the datasets i.e. 1.7.

The ratios achieved by low to high degree method for both the datasets were 2.1 and the ratios achieved by high to low degree method were 3 and 3.4. Since the high to low degree method resulted in a 5:1 and 7:1 ratio for distribution of edges, this is the cause for the low vertices ratio. One of the partitions has 5 to 7 times higher edges than the other partition; hence, it would have higher vertices than the other partition resulting in a low ratio of 0.3.

VII. CONCLUSION AND FUTURE WORK

Among all the four proposed method, high to low degree approach resulted in the lowest value for boundary vertices to total vertices ratio which was the objective of this study but the distribution of edges and vertices for this approach were uneven i.e. both the edge and vertices distribution ratios were higher according to the results which showed that it resulted in an uneven distribution of edges and vertices across the two partitions. The edges and vertices distribution for round robin approach of high to low degree method was same in case of edges distribution and higher by a ratio of 0.3 in vertices distribution but it had a high boundary vertices to total vertices ratios of 0.38 and 0.39 close to the random method which had ratios of 0.39 and 0.40 for the two data sets.

For future work, the high to low degree and the round robin approach for high to low degree can be further modified so it can give better results for edge and vertices distribution as well as better results for boundary vertices as well. The major reason of one of the partitions having a larger number of vertices was the condition in the greedy heuristics of Graph Lab algorithm which stated that if one of the vertices of an edge currently in consideration is already present in one of the partitions then that edge should be assigned to that particular partition which caused this uneven partition distribution and hence high vertices and edges distribution ratios.

ACKNOWLEDGEMENT

First of all, I would like to thank GOD, my family (because of whom I have been here) and I would like to take this opportunity to express my profound gratitude and deep regard to my supervisor Dr. Syed Saif Ur Rahman for stimulating suggestions and constant encouragement throughout the duration of this Study.

REFERENCES

[1] C-E. Bichot and P. Siarry, *Graph Partitioning*. New Jersey: Wiley, 2011.

[2] *Stanford Large Network Dataset Collection*. [Online]. Available from: https://snap.stanford.edu/data/.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “PowerGraph: Distributed Graph-parallel Computation on Natural Graphs,” In *Proceedings of 10th USENIX symposium on Operating Systems Designs and Implementation (OSDI’12)*, 2012, pp: 17-30.

[4] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud,” In *Proceedings of VLDB Endowment.,* 2012, vol. 5, no. 8, pp: 716-727.

[5] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, “GraphLab: A New Parallel Framework for Machine Learning,” In *Proceedings of Conference on Uncertainty in Artificial Intelligence (UAI),* Catalina Island: CA, 2010.

[6] G. Czajkowski, “Pregel: A System for Large-scale Graph Processing,” In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD 2010),* 2010, pp: 135-146.

[7] X. Sui, D. Nguyen, M. Burtscher, and K. Pingali, “Parallel Graph Partitioning on Multicore Architectures,” In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing (LCPC 2010),* 2010, pp: 246-260.

[8] D. Crankshaw, A. Dave, R. S. Xin, J. E. Gonzalez, M. J. Franklin, and Ion Stoica. "*The GraphX Graph Processing System,*" Res. Rep., AMP lab, Univ. of California: Berkley, 2013.

[9] F. Bourse, M. Lelarge, and M. Vojnovic. "Balanced graph edge partition," In *Proceedings of the 20th ACM SIGKDD International Conference On Knowledge Discovery And Data Mining (KDD 2014),* 2014, pp: 1456-1465.

[10] L. Golab, M. Hadjieleftheriou, H. Karloff and B. Saha, “Distributed Data Placement via Graph Partitioning,” *CoRR*, vol. abs/1312.0285, 2013.