2017 IEEE 33rd International Conference on Data Engineering

Skew-Resistant Graph Partitioning

Angen Zheng
University of Pittsburgh
anz28@cs.pitt.edu

Alexandros Labrinidis
University of Pittsburgh
labrinid@cs.pitt.edu

Christos Faloutsos
Carnegie Mellon University
christos@cs.cmu.edu

***Abstract*—Large graph datasets have caused renewed interest for *graph partitioning*. However, existing well-studied graph partitioners often assume that vertices of the graph are always active during the computation, which may lead to *time-varying skewness* for traversal-style graph workloads, like Breadth First Search, since they only explore part of the graph in each superstep. Additionally, existing solutions do not consider what vertices each partition will have; as a result, high-degree vertices may be concentrated into a few partitions, causing imbalance. Towards this, we introduce the idea of *skew-resistant graph partitioning*, where the objective is to create an initial partitioning that will "hold well" over time without suffering from skewness. Skew-resistant graph partitioning tries to mitigate skewness by taking the characteristics of both the target workload and the graph structure into consideration.**

I. Introduction

The increasing popularity of large graphs with millions and billions of nodes and edges, such as the World Wide Web, Biological Networks, and Social Networks, has led to the development of many distributed graph computing frameworks (e.g., Pregel [8], GraphLab [7] and PowerGraph [3]). These frameworks often require a partitioning of the graph across a set of machines for parallel computation.

Pregel, one of the most representative graph computing frameworks, adopts the *BSP (Bulk Synchronous Parallel)* model. In such a model, computations are performed in a sequence of *supersteps* separated by a global synchronization barrier. Each superstep *thinks like a vertex*, in which a user-defined function is computed against each vertex. The function can change the vertex state and the state of its outgoing edges based on the messages it received, send messages to its neighbors, or even modify the structure of the graph. Vertices can vote to halt during the computation and be reactivated again by messages from their neighbors. The program terminates when all the vertices become inactive.

Clearly, a good partitioning of the graph can allow for high degrees of parallelism and can greatly reduce both the network communication and the overall runtime. To this end, there are dozens of graph partitioners, from the "classic" ones (e.g., [4], [14], [9], [15]) to new, *(re)streaming* graph partitioners (e.g., [17], [18], [20], [11]), which address the scalability challenge. However, despite the large amount of work so far (including our own prior work, [21], [24], [22], [23]), *largely overlooked are the effects of different types of skewness* on the performance of the graph partitioning.

Algorithmic Skewness Current graph partitioners all assume that a balanced partitioning of the graph is equivalent to an even load distribution. Put simply, they all assume that vertices of the graph are always active during the computation. This is true for *always-active-style* graph algorithms, like PageRank. However, for *traversal-style* graph algorithms, like breadth first search (BFS) and single-source shortest path (SSSP), only a subset of vertices are explored in each superstep. As a result, vertices active in the same superstep may be concentrated into a few partitions by existing graph partitioners, leading to load imbalance, resource underutilization, and contention on the network interface. One way to avoid this *algorithmic skewness* is to migrate vertices dynamically based on some system metrics [16], [6], [19]. However, this is too late and the migration is not cost-free. Migrating a vertex to a new partition requires migrating both its edge list and its associated application data plus an update of the vertex location.

Structural Skewness Existing graph partitioners often do not care about what vertices each partition will have. As a result, high-degree vertices may be concentrated into a few partitions, causing a new type of imbalance, *structural skewness*. This is because high-degree vertices are often the computation and communication hotspots given their large neighborhood. Unfortunately, graphs from various important domains are scale-free, where the vertex degree-distribution asymptotically follows a power law distribution [2], [13].

Side-Effect of Algorithmic and Structural Skewness Another side effect of the skewness on modern multicore machines is that it may lead to contention for the shared resources in the memory subsystems, especially when the partitions that contain most of the active vertices are assigned to the cores of the same machine for parallel processing. This is because intra-node data communication (the communication among cores of the same machine) is often implemented via shared memory [5], [1], requiring additional data copies. Thus, having too much data communication among partitions that are residing on the same machine may lead to serious cache pollution and therefore contention for the shared last level cache, front side bus, and memory controller. Several recent works [24], [22], [23] have shown that contention on the memory subsystems may have a great performance impact on distributed graph computation.

Contributions To address the needs of efficient distributed graph computation, we make the following contributions:

1. To better understand the skewness issue, we experimentally demonstrate the runtime characteristics of two classic traversal-style-graph workloads (Section II-A) and their predictability using real-world graphs (Section II-B).
2. We introduce the idea of multi-label graph partitioning (MLGP) (Section III-A) and an application of MLGP to do skew-resistant graph partitioning (Section III-B).

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.62

II. Traversal-Style Graph Workload Characterization

A. Active Vertex Distribution Across Supersteps (Table I)

Configuration In this Section, we examined the runtime characteristics of BFS and SSSP on the Orkut dataset. Orkut is a social network run by Google [12] for people across the world to discuss their common interests. The dataset used is a subset of Orkut's user population (around 11.3% at the time crawled by A. Mislove et. al. [10]). The degree distribution of the dataset follows a power-law distribution with average and maximal vertex degree equal 76.281 and 33,313, respectively. The maximal diameter of dataset is 10 with the effective diameter of 5.4489.

In the experiment, the graph was partitioned across six 20-core machines using three different techniques with one partition per core. The techniques examined included: (a) *METIS*, a well-known multilevel graph partitioner [9]; (b) *LDG*, a state-of-the-art streaming graph partitioner [17]; and (c) *reLDG*, a state-of-the-art restreaming graph partitioner [11].

Results Table I presents the number of vertices that are active in each superstep for the execution of BFS/SSSP with one randomly selected source vertex. As shown, only a subset of the vertices were active in each superstep, and the execution exhibited highly skewed active vertex distribution across supersteps. The top-3 supersteps with largest fraction of active vertices covered around 96% of vertices of the graph. This was expected for *small-world* and *scale-free* graphs. Small-world graphs are known to have low diameter. Consequently, the execution of BFS/SSSP on such graphs usually ends in a few supersteps, causing a large number of vertices to be visited per superstep. On the other hand, the scale-free property allows the number of vertices active in each superstep to be expended and shrink exponentially. As a result, a majority of vertices were visited in very few supersteps. These supersteps were also the top-3 most time-consuming supersteps.

We observed similar results for the execution of BFS/SSSP on the partitionings computed by *METIS*, *LDG*, and *reLDG*. This was because (1) the execution of BFS/SSSP on the partitionings all started from the same randomly selected source vertex; and (2) the way the graph was distributed across partitions only affected the amount of data communication performed by BFS/SSSP (but not the algorithm characteristics).

Take-away *To achieve superior performance, we should offer differentiated partitioning for vertices that are active in the peak supersteps. That is, we should focus more on reducing the edge-cut of vertices that are active in the peak supersteps and balancing the load of the peak supersteps.*

B. Active Vertex Distribution Across Partitions (Fig. 1 & 2)

Configuration This experiment examined the corresponding *active vertex* and *active high-degree* vertex distribution across partitions for the execution of BFS/SSSP on the partitionings. We treated the top 1% vertices as the high-degree ones. For brevity, we only showed the results of BFS in Figures 1 and 2 for the most time-consuming superstep (Step 4 of Table I).

Results As can be seen, the execution of BFS on the partitionings computed by *METIS*, *LDG*, and *reLDG* all exhibited highly skewed active vertex and active high-degree vertex distribution across partitions, especially the distribution of high degree vertices (around half of the partitions have nearly zero active high-degree vertices). This may lead to potential significant load imbalance and thus resource underutilization as well as contention on both the network interface and memory subsystems. Another interesting result was that the decomposition computed by *METIS* had the largest skewness followed *reLDG* next to it. This was somehow expected considering the fact that *METIS* tends to produce partitionings of the highest quality, while *LDG* performed the worst among the three. Put simply, *METIS* and *reLDG* were better than *LDG* in grouping tightly connected vertices together, leading to higher chance of load imbalance. This also explains the reason why simple partitioning techniques (e.g., hashing partitioning) may sometimes perform better than those well-studied ones.

TABLE I: Active vertex distribution across supersteps of BFS & SSSP execution with one randomly selected source vertex

com-orkut	# of Active Vertices	
Supersteps	BFS	SSSP
0	1	1
1	72	45
2	5,871	4,663
3	215,425	297,943
4	1,753,891	1,421,993
5	1,088,870	1,229,917
6	8,242	117,496
7	69	383
8	0	0

Take-away *We should consider the characteristics of both the target workload and the graph structure while partitioning.*

C. Workload Predictability (Table II)

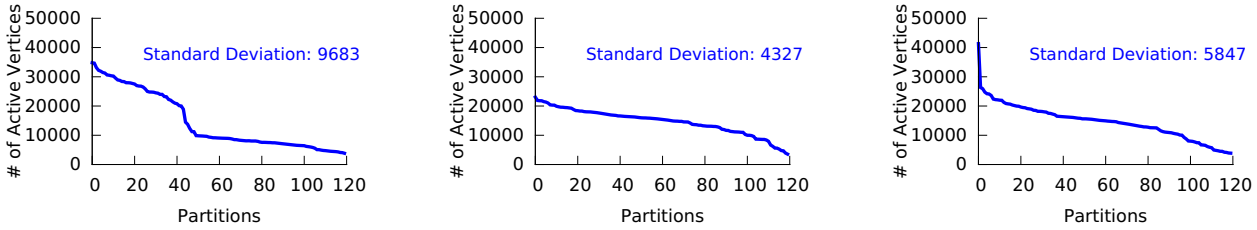
Given the above observations, one may wonder if we could incorporate such characteristics into the partitioning process, such that both the algorithmic and structural skewness are minimized. Towards this, we kept track of vertices that were active in each superstep for five distinct executions of BFS/SSSP on the Orkut dataset. Each such execution was performed with one randomly selected source vertex. Then, we examined the repeatability of the execution traces. Considering the highly skewed active vertex distribution across supersteps, we only considered the top-3 most time-consuming supersteps for repeatability computation. We defined the repeatability of execution trace tr_1 with respect to tr_2 as:

$$repeat(tr_1, tr_2) = \frac{\sum_{i=1}^{3} \max_{j=1,2,3} |s_{tr_1(i)} \cap s_{tr_2(j)}|}{\sum_{i=1}^{3} |s_{tr_1(i)}|} \quad (1)$$

where $s_{tr_1(i)}$ denotes the set of vertices that are active in the ith most time-consuming superstep of trace tr_1. The execution trace repeatability indicates the degree of overlap among traces. It should be noted that this was a conservative estimation, because $s_{tr_1(i)}$ may overlap with multiple supersteps of execution trace tr_2. Yet, we only considered the superstep that overlaps $s_{tr_1(i)}$ the most. We then defined the predictability of a workload W on a specific graph G as:

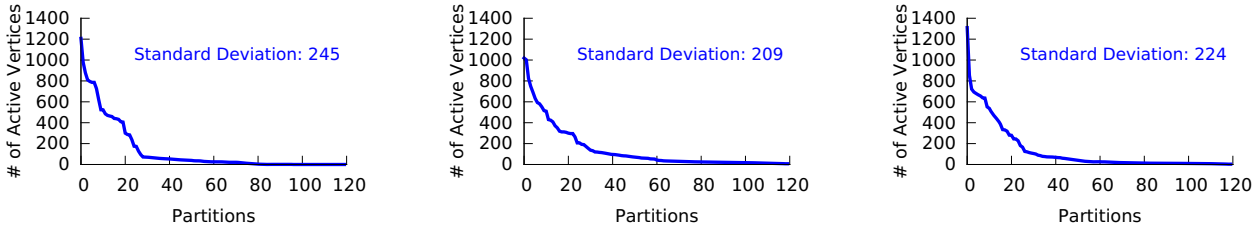
$$predict(W, G, Tr) = \frac{\sum_{i=1}^{|Tr|} \sum_{j=1 \text{ and } i \neq j}^{|Tr|} repeat(tr_i, tr_j)}{|Tr|(|Tr|-1)} \quad (2)$$

where Tr is a given set of execution traces of W on G.



(a) *METIS* (b) *LDG* (c) *reLDG*

Fig. 1: BFS *active vertex* distribution across partitions for the most time-consuming superstep (Step 4 of Table I) on com-orkut dataset with one randomly selected source vertex. The distribution was measured, when the graph was partitioned across six 20-core machines with one partition per core.



(a) *METIS* (b) *LDG* (c) *reLDG*

Fig. 2: BFS *active high-degree vertex* distribution across partitions for the most time-consuming superstep (Step 4 of Table I) on com-orkut dataset with one randomly selected source vertex. The distribution was measured, when the graph was partitioned across six 20-core machines with one partition per core.

Table II presents the predictability of BFS and SSSP as well as the standard deviation of the repeatability. As shown, the runtime characteristics of both BFS and SSSP on orkut were actually quite predictable. On average, around 60% of vertices are always active in the same supersteps for two distinct executions of BFS/SSSP with one randomly selected source vertex. The relatively high repeatability can be explained by the *wave* access pattern of the traversal-style graph workloads. That is, once the superstep active vertex set of distinct executions of the workload intersects at certain superstep, the common active vertex set will be expanded larger and larger (up to a certain point), especially if we hit a high-degree vertex. This is because all the neighbours of the vertices in the current common active vertex set will become active in the next superstep. Note we observed similar results for the execution of BFS/SSSP on partitionings computed by *METIS*, *LDG*, and *reLDG*.

Take-away *The execution trace of the traversal-style graph workloads on many small-world and scale-free graphs can be used as a representative of the runtime characteristics of the target workloads. This provides us an opportunity to leverage the runtime characteristics of the target workload into the partitioning process (using the execution trace).*

TABLE II: Workload Predictability

Algorithm	Predictability	Standard Deviation
BFS	0.6080	0.0843
SSSP	0.6473	0.1063

III. Skew-Resistant Graph Partitioning

In this section, we first introduce the *Multi-Label Graph Partitioning (MLGP)* problem (Section III-A) and then present an application of such idea to do *Skew-Resistant Graph Partitioning* (Section III-B)

A. MLGP: Multi-Label Graph Partitioning

Let $G = (V, E, L)$ be a graph with labels on vertices, where V is the set of vertices, E is the set of edges, and $L = \{L_1, L_2, ...L_m\}$ is the set of labels associated with vertices in V. Each vertex is associated with a binary label vector, indicating if the corresponding label exists on the vertex. MLGP aims to minimize the communication cost among partitions under the constraint (1) that each partition is balanced; (2) and that vertices of each partition follow a user-defined distribution in terms of their labels. The quality

of the partitioning (communication cost) is defined as:

$$comm(G, MLGP) = \sum_{\substack{e=(u,v)\in E \text{ and} \\ u\in P_i \text{ and } v\in P_j \text{ and } i\neq j}} w(e) \quad (3)$$

where $w(e)$ is the edge weight, indicating the amount of data communication between the vertex pair.

Constraint 1 can be formally defined as:

$$\sum_{v\in P_i} w(v) \leq C(P_i) \text{ for i} \in [1, k] \quad (4)$$

where k corresponds to the number of partitions we want, $w(v)$ is the vertex weight (indicating the computational requirements of the vertex), and $C(P_i)$ denotes the partition capacity. As for *Constraint 2* (the vertex distribution of each partition), we are particularly interested in distributing vertices of the same labels evenly across partitions, which can be formulated as:

$$\sum_{v^l\in P_i} w(v^l) \leq C^l(P_i) \text{ for i} \in [1, k] \quad (5)$$

where v^l denotes vertices that have label L_l, whereas $w(v^l)$ and $C^l(P_i)$, respectively, corresponds to the vertex weight and the partition capacity of P_i for l-labelled vertices. In other words, we want each partition to eventually have a similar vertex distribution to the original graph in terms of their labels. In case of vertices of the graph do not have any label, Constraint 1 is self-included in Eq. 5. Sometimes, we may only want to apply Constraint 2 to a subset of $|V|$ while guaranteeing the rest of vertices do not violate Constraint 1.

B. MLGP: Skew-Resistant Graph Partitioning

1) Avoiding Algorithmic Skewness: To guarantee that the load of the traversal-style graph workloads is evenly distributed in every superstep, we model it as a MLGP problem, in which we only need to divide the entire execution time into finite time periods, and associate each vertex with a label vector. The label vector indicates the time periods in which the vertex is active. Given the relatively high predictability of the runtime characteristics of BFS and SSSP on the datasets of interest (Section II), we use the supersteps as the natural time periods and obtain the label vector from the execution trace. With the augmented label information, MLGP will automatically split vertices active in the same superstep evenly across partitions while keeping the communication among partitions as small as possible, thus eliminating algorithmic skewness.

2) Avoiding Structural Skewness: Considering the relatively small number of high-degree vertices and vast disparity in the vertex weights the graph may have, we avoid structural skewness by simply assuming that all high-degree vertices are active in a single additional superstep. By doing this, MLGP will attempt to distribute high-degree vertices evenly across partitions. At the same time, the labels that high-degree vertices originally have can serve as a way to penalize partitions that have a large number of vertices that are active at the same time with the high-degree ones. This also means that high-degree vertices originally active in the same superstep will have a smaller chance to be put together.

IV. Conclusion

In this paper, we introduce the multi-label graph partitioning problem and an application of such idea to avoid the skewness of traversal-style graph workloads by being aware of the characteristics of both the workload and the graph structure. We also show that the execution traces of many traversal-style graph workloads on small-world and scale-free graphs are actually quite representative of their runtime characteristics.

Acknowledgement

We would like to thank Panos K. Chrysanthis, Cory Thoma, and Kenrick Fernandes for their help on the paper. This work was funded in part by NSF awards CBET-1250171 and CBET-1609120.

References

[1] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud. Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis. In *ICPP*, 2009.
[2] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM CCR*, 1999.
[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Power-Graph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.
[4] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 2000.
[5] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *ICPP*, 2005.
[6] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, 2013.
[7] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv:1408.2041*, 2014.
[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.
[9] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.
[10] A. Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, 2009.
[11] J. Nishimura and J. Ugander. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *SIGKDD*, pages 1106–1114, 2013.
[12] Orkut Community Archive. https://orkut.google.com/en.html.
[13] E. Papalexakis, B. Hooi, K. Pelechrinis, and C. Faloutsos. Power-Hop: A Pervasive Observation for Real Complex Networks. *PloS one*, 2016.
[14] K. Schloegel, G. Karypis, and V. Kumar. *Graph partitioning for high performance scientific simulations*. AHPCRC, 2000.
[15] http://www.labri.u-bordeaux.fr/perso/pelegrin/scotch/.
[16] Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *ICDE*, 2013.
[17] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *SIGKDD*, 2012.
[18] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM*, 2014.
[19] N. Xu, L. Chen, and B. Cui. LogGP: a log-based dynamic graph partitioning method. *VLDB*, 2014.
[20] N. Xu, B. Cui, L.-n. Chen, Z. Huang, and Y. Shao. Heterogeneous Environment Aware Streaming Graph Partitioning. *TKDE*, 2015.
[21] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Architecture-Aware Graph Repartitioning for Data-Intensive Scientific Computing. In *BigGraphs*, 2014.
[22] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Planar: Parallel Lightweight Architecture-Aware Adaptive Graph Repartitioning. In *ICDE*, 2016.
[23] A. Zheng, A. Labrinidis, P. K. Chrysanthis, and J. Lange. Argo: Architecture-Aware Graph Partitioning. In *IEEE BigData*, 2016.
[24] A. Zheng, A. Labrinidis, P. Pisciuneri, P. K. Chrysanthis, and P. Givi. Paragon: Parallel Architecture-Aware Graph Partitioning Refinement Algorithm. In *EDBT*, 2016.