Accelerating Graph Processing With Lightweight Learning-Based Data Reordering

Mo Zou, Mingzhe Zhang, Rujia Wang, *Member, IEEE*, Xian-He Sun, *Fellow, IEEE*, Xiaochun Ye, Dongrui Fan, and Zhimin Tang, *Member, IEEE*

Abstract—Graph processing is a vital component in various application domains. However, a good graph processing performance is hard to achieve due to its intensive irregular data accesses. Noticing that in real-world graphs, a small portion of vertices occupy most connections, several techniques are proposed to reorder vertices based on their access frequency for better data access locality. However, these approaches can be further improved by identifying reordered data more effectively, which will reduce reordering overhead and improve overall performance. In this letter, we propose *Learning-Based Reordering (LBR)*, a novel lightweight framework that identifies and reorders hot data adaptively for given graphs, algorithms, and threads. Our experimental evaluation indicates that *LBR* decreases reordering overhead by 24.7% while improves performance by 9.9% compared to the best-performing existing scheme.

Index Terms—Graph processing, reordering technique, learning-based prediction model

1 Introduction

GRAPH is a powerful data structure to describe real-world relations and is widely used in various application domains. However, graph processing workloads suffer from poor performance due to their irregular memory access patterns, making it hard to utilize locality-based performance enhancement designs. Moreover, the large footprint of graph datasets, which is far beyond the cache capacity, magnifies the impact of random data accesses on the system performance by causing frequent cache misses.

On the other hand, most of the real-world graphs follow a *power-law* distribution, indicating that a small portion of vertices contributes to most connections [2]. Based on this observation, previous works focus on packaging the vertices with higher access probabilities (i.e., hot vertices) in successive memory blocks before the execution phase. A common way to determine the hot vertices is to utilize the vertex degree. Therefore, the hot vertices identification problem is converted to determine a degree threshold that separates high-degree and low-degree vertices [6].

Previous works [1], [2], [7] calculate the average degree (i.e., the ratio of edges and vertices number) as the threshold for hot data identification. Such a static estimation may cause two problems: First, the selected threshold may be too low, and some *non-hot data* may be reordered, which will not only increase the reordering overhead but also make the scale of reordered vertices exceed the cache capacity, limiting the benefit of such an optimization. Second, since graph processing performance is also sensitive to dynamic factors like algorithms and threads, the static degree calculated from the graph dataset may not be enough.

Therefore, we are motivated to find a balance between improving cache performance and reducing reordering overhead. Our contributions can be summarized as follows:

- We quantify the reordering overhead increasing trend with the reordered vertices scale growth and further prove that reordering a small portion of vertices can achieve significant performance improvements.
- We propose *Learning-Based Reordering (LBR)*, a novel framework contains a learning-based prediction model for hot data identification and a lightweight reordering scheme for improving data locality.
- We evaluate the effectiveness of *LBR* on a real machine across 80 datapoints, showing that *LBR* improves performance by 9.9% while reduces reordering overhead by 24.7% over the best-performing existing reordering technique.

2 Motivation

Vertex reordering is a straightforward optimization in graph processing based on the *power-law* distribution. However, previous studies either fail to control reordering overhead or fail to achieve the highest performance improvement.

Sort replaces all vertices in a descending or ascending order, resulting in significant reordering overhead. HubCluster [1] and HubSort [7] categorize vertices with an equal or higher degree than the average as hot ones and reorder them successively in the memory. Unfortunately, in this case, the scale of the hot vertices exceed the cache capacity for most real-world graphs, which limits the ability of reordering. Table 1 quantifies the portion of hot vertices determined by the average in-degree and its storage scale. On average, 18.3% of vertices are marked as hot ones, occupying 125.1 MB across fourteen datasets evaluated in our work. DBG [2] divides vertices into multiple groups according to their degrees and maintain vertices within any group in their original order, which reduces reordering overhead and preserves graph structures. But since DBG still utilizes static parameters (i.e., the average degree and its multiples) as the group boundaries, its performance is not stable facing dynamic factors like various thread configurations. RCM [3] is effective in reducing memory bandwidth but can not solve the irregular memory access patterns well.

To summarize, nearly all previous works choose the static average degree to guide the reordering scheme. However, our evaluations find that the over-estimated reordered data set increases the reordering time significantly, while has a minimum impact on execution time. Fig. 1 demonstrates the reordering and execution time of the application SSSP on the datasets *pk* and *ll* with different reordered vertices scale. We change the degree thresholds for reordered vertices identification and thus vary the reordered vertices portion from 5% to 30%. We make the following observations:

- Not surprisingly, a larger reordered data set requires longer reordering time. In particular, when running with *ll*, the reordering time grows from 8s to 15s as the proportion of the reordered data increases from 5% to 30%.
- Meanwhile, the execution time changes slightly as the scale of reordered data grows. For instance, in *ll*, when the

- *Mo Zou is with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100045, China, and also with the University of Chinese Academy of Sciences, Beijing 101408, China. E-mail: zoumo@ict.ac.cn.*
- *Mingzhe Zhang is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100045, China. E-mail: zhangmingzhe@iie.ac.cn.*
- *Rujia Wang and Xian-He Sun are with the Illinois Institute of Technology, Chicago, IL 60616 USA. E-mail: {rwang67, sun}@iit.edu.*
- *Xiaochun Ye, Dongrui Fan, and Zhimin Tang are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100045, China. E-mail: {yexiaochun, fandr, tang}@ict.ac.cn.*

Manuscript received 6 Jan. 2022; revised 7 Feb. 2022; accepted 8 Feb. 2022. Date of publication 14 Feb. 2022; date of current version 3 Mar. 2022.
This work was supported in part by the National Science Foundation under Grants CCF-2008907 and CCF-2029014 and in part by the National Natural Science Foundation of China under Grants 61732018, 61872335, and 61802367, in part by Austrian-Chinese Cooperative R&D Project (FFG and CAS) under Grant 171111KYSB20200002.
(Corresponding author: Mo Zou.)
Digital Object Identifier no. 10.1109/LCA.2022.3151087

TABLE 1
Hot Vertices Percentage and Scale Guided by the Average In-Degree

Graph	Perc.	Scale (MB)	Graph	Perc.	Scale (MB)
pl	20%	70	*bd*	12%	59
pt	16%	59	*ru*	17%	111
ja	19%	70	*fr*	16%	111
pk	32%	119	*ol*	33%	232
fl	12%	47	*de*	17%	126
it	17%	73	*hd*	9%	40
lj	25%	278	*ll*	30%	357

proportion of the reordered vertices increases from 5% to 30%, the execution time only decreases from 12.39s to 12.37s.

The over-estimated reordered data scale affects system performance from two aspects. First, the scale of hot vertices often exceeds cache capacity, limiting the performance enhancement. Second, real-world graphs often exhibit community characteristic [4], indicating that vertices placed nearby in the memory tend to be accessed successively. Changing the location of too many vertices cannot preserve the original structure and will damage the performance.

Based on the above analysis, A smaller group of reordered vertices will lower reordering overhead and preserve the original graph features. As an alternative to choose a static average degree, we prefer an adaptive framework considering dynamic factors and identifying a relatively smaller portion of hot vertices, motivating us to exploit our proposed framework, *LBR*.

3 Framework Design and Implementation

3.1 Characteristic Space

We use Machine Learning to identify the appropriate degree threshold dynamically. In the training phase, the prediction model learns from data samples, whose input is a vector as shown in Equation (1) and the output is a degree threshold.

In input vector, the vertex and edge numbers change with various graph datasets, demonstrating static graph density. The application ID and thread number reflect dynamic runtime features. The output of the model is the degree threshold of hot vertices, which is also a reflection of hot vertices scale as well. Those vertices with a higher degree than the predicted degree will be replaced successively in the memory to improve locality while the others will stay in the original place to decrease reordering overhead and pursue community features

$$input\ space = \begin{bmatrix} \text{vertex number} \\ \text{edge number} \\ \text{application ID} \\ \text{thread number} \end{bmatrix}. \qquad (1)$$

3.2 Training Phase

To generate training samples, we implement fine-grained experiments to find optimal degrees for distinct [vertex, edge, application, thread]^T configurations. Noticing that most real-world graphs are directed graphs, we choose in-degree or out-degree thresholds

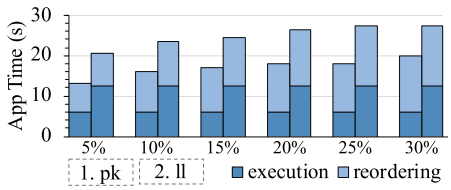

Fig. 1. Application time with different reordered vertices scale.

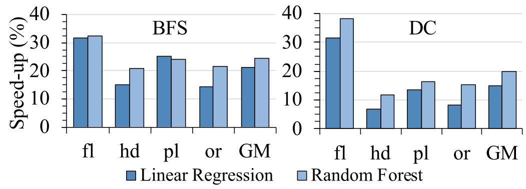

Fig. 2. Multi-variable linear regression *versus* random forest.

as outputs according to application behaviors. If an application traverses the graph following out-neighbors of each vertex, vertices with more in-neighbors are more likely to be accessed. Then we build the training set using in-degree as the outputs and predict in-degree threshold in the testing phase to group vertices into hot and cold ones.

For an input vector [vertex, edge, application, thread]^T, we change the degree threshold, reorder hot vertices identified by different degree thresholds and evaluate the application speed-ups. We choose the top ten degrees with the best performance as the outputs, generating ten samples for an input vector. We further vary the application ID and thread configurations respectively so as to obtain dissimilar samples facing dynamic factors. It takes months to generate input samples and train the model. The model is reusable, not requiring re-train for future workloads.

3.3 Model Selection

To build a lightweight degree prediction model with the highest performance improvement, we explore two learning-based models: *multi-variable linear regression* and *random forest*, and estimate application speed-ups with their predicted degree thresholds. We predict degree thresholds with the same input samples on the two learning-based models. Fig. 2 exhibits the application BFS and DC speed-ups achieved by *linear regression* and *random forest*. BFS provides 24.3% speed-up over the baseline averaging across four testing datasets when *random forest* is adopted, outperforming 20.9% acceleration from *linear regression*. Similarly, *random forest* supplies an outstanding average speedup of 20.1% in comparison to 14.8% for *linear regression* in DC. We find that overfitting in *linear regression* is critical since there exist no obvious linear relationships between various threads in the characteristic space. On the other hand, *random forest* enhances the overfitting problem through several disjoint decision trees. Therefore, we select *random forest* in *LBR* framework as our learning-based model since the predicted degree consistently achieves a higher application speed-up over the baseline than *multi-variable linear regression*.

3.4 Prediction-Based Hot Data Reordering

Our proposed *LBR* framework knows which vertex ought to be rearranged with the help of the degree threshold predicted by the learning-based model. As shown in Fig. 3, we train the prediction model offline with generated samples. In the testing phase, we input the vertex and edge numbers of the testing graph, the application ID, and the thread number to the trained model. The model outputs a predicted degree, which guides the reordering phase.

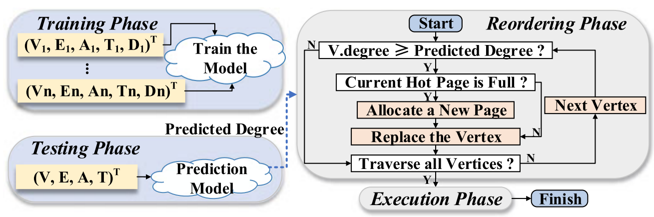

Fig. 3. *LBR* implementation and workflow.

TABLE 2
Configuration of the Simulated System

		Configuration
Processor	Type	Intel Xeon Silver 4114
	Frequency	2.2 GHz
	Core	2 sockets, 10 cores per socket
	L1	32KB private, 8-way
	L2	1MB private, 16-way
	L3	13.75MB shared, 11-way
Other	Memory	48 GB
	OS	CentOS 7

TABLE 3
Training Graphs

Dataset	|V|	|E|	Dataset	|V|	|E|
Wiki_pt (*pt*)	1.6	49.02	Japan (*ja*)	1.61	71.06
Pokec (*pk*)	1.63	30.62	InterLinks (*it*)	1.87	91.56
Baidu (*bd*)	2.14	17.8	Wiki_ru (*ru*)	2.85	82.06
Friends (*fr*)	3.02	102.38	England (*de*)	3.23	81.63
LiJournal (*lj*)	4.85	68.48	LiveLinks (*ll*)	5.2	49.17

|V| and |E| are all in millions.

At the beginning of the reordering phase, *LBR* allocates an empty page waiting for hot vertices. Then *LBR* traverses the graph and compares each vertex's degree with the predicted degree. Only vertices with an equal or larger degree than the predicted one are replaced in the allocated page. Once that page is full, another page will be allocated to keep hot vertices. *LBR* performs the execution phase until all vertices are traversed.

We reorder hot vertices together to improve cache performance but allocate one page each time to decrease the demand for contiguous memory space. In such a case, hot vertices are located in concentrated pages without influenced by cold ones. Since the predicted degree is higher than the average degree, the scale of hot vertices is much smaller, benefiting the application from a lower reordering overhead and a cluster-friendly representation.

4 Evaluation

4.1 Experimental Setup

We implement *LBR* on GraphBIG [5], a widely used vertex-centric graph framework. Rather than CSR format, the graph is denoted by a vertex list containing pointers to all vertices. Each vertex is an independent unit composed of vertex ID, property value, and in/out-edge lists. Table 2 shows the hardware details of the evaluated system.

We choose four classical graph algorithms (i.e., BFS, CCMP, SSSP, and DC), in our experiments. Since all algorithms traverse graphs following out-neighbors, *LBR* learns and predicts in-degree thresholds for all applications.

In the training phase, we select a set of real-world graphs from different fields with various vertex and edge numbers, as shown in Table 3, and run fine-grained experiments to find the best-performing degree thresholds to generate input samples. We consider uniform random datasets to generate our input samples. They have different vertex and edge numbers, representing various density features. They come from different domains including social networks, twitter followers, and so on, so that we can safely consider different graphs to have uniform behavior. In the testing phase, we generate input vectors on four graphs (shown in Table 4) using their vertex and edge numbers combing with application and thread configurations. We predict in-degree thresholds for the four unseen graphs and evaluate the performance improvements guided by predicted degrees. These testing graphs have significant different vertex and edge numbers with the training datasets, ensuring the effective test of *LBR*.

TABLE 4
Testing Graphs

Dataset	|V|	|E|	Dataset	|V|	|E|
Wiki_pl (*pl*)	1.53	57.49	Flicker (*fl*)	1.72	15.55
Hudong (*hd*)	1.95	14.87	Orkut (*or*)	3.07	117.18

|V| and |E| are all in millions.

4.2 Experimental Results and Analysis

Performance: We evaluate *LBR* and compare it with Sort, HubCluster [1], RCM [3], HubSort [7], and DBG [2] - the-state-of-the-art reordering technique, over the LRU baseline without reordering.

Fig. 4 demonstrates application speed-ups excluding reordering time for various testing graph. Each bar is a geometric mean across five thread configurations. Averaging across all 80 datapoints, *LBR* provides 17.3% speed-up over the baseline, outperforming HubCluster by 9.9%, DBG by 10%, RCM by 50.4%, Sort by 27.1%, HubSort by 20%. Through replacing hot vertices continuously in the memory, *LBR* creates a cache-friendly scenario, improving the locality of vertex property value.

As shown in Fig. 4, Sort replaces vertices in a descending degree order, destroying the original graph structure completely, thus causing application slowdown (-9.8%) averaging all the datapoints. HubCluster and HubSort narrow the reordered vertices by utilizing average degree as the threshold. However, in some cases, like the application CCMP on the dataset *fl*, they reorder too many vertices and can not preserve graph community features very well, leading to significant performance decrements (-37.1% and -9.7%). On the other hand, *LBR* labels much fewer vertices determined by the predicted degree threshold, protecting the structure most and benefiting the performance. RCM is a classical algorithm to reduce graph bandwidth. However, the poor performance in graph applications mainly comes from irregular data accesses, so RCM does not solve the performance bottleneck. DBG receives an average performance improvement by 7.3% over the baseline. However, we find that facing dynamic factors like thread configurations, DBG can not achieve a stable benefit. For example, in the application CCMP on the dataset *pl*, DBG yields a speed-up by 36.7% with 20 threads but -64.7% with 40 threads. *LBR* is an adaptive

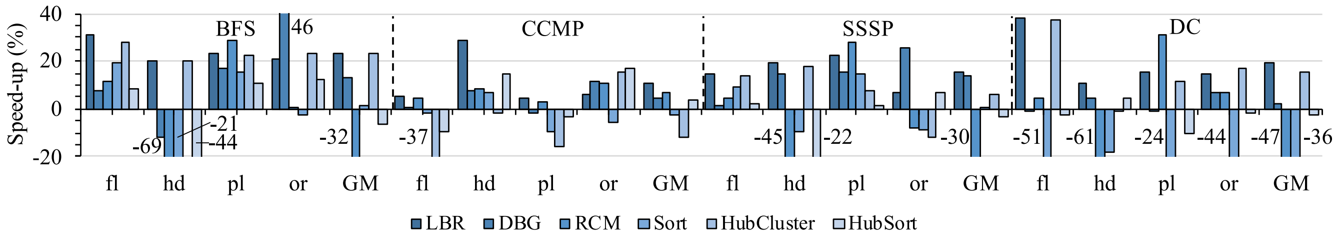

Fig. 4. Application speed-ups (excluding reordering time) for LBR and prior techniques over the baseline without reordering.

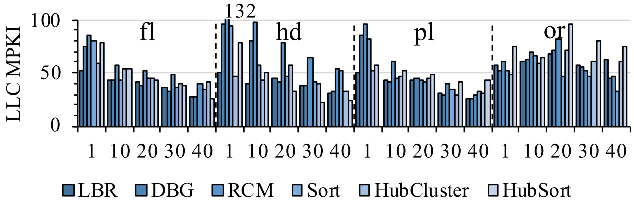

Fig. 5. LLC MPKI for the application BFS across graphs. Lower is better.

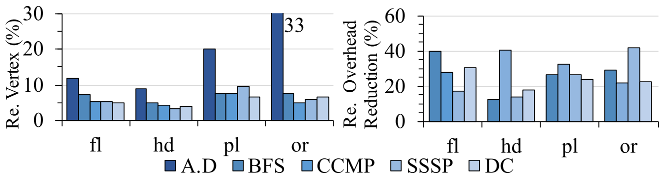

Fig. 6. Reordered vertices proportion and reordering overhead reduction.

framework considering runtime features, providing higher performance than DBG.

LLC Misses Per Kilo Instructions (MPKI): To further explain the performance increments, we analyze MPKI on LLC using various reordering techniques. Fig. 5 shows LLC MPKI on the application BFS configured by various threads. On average, LLC MPKI in *LBR* is the lowest (43) while in RCM is the highest (62), clarifying the huge performance gap between these two schemes. Although DBG generates the least LLC MPKI in some cases (i.e., on the dataset *pl* with ten-thread configuration), it can not work well facing dynamic factors, producing 47 LLC MPKI on average.

Hot Data Proportion and Reordering Overhead. Since HubCluster is the best-performing reordering technique in our evaluations, we quantify the reordered data proportion of average degree (employed by HubCluster) and our proposed *LBR*. Fig. 6 (left) shows, for all datasets, *LBR* can generate less reordered vertices. On average, the reordered hot vertices proportion reduces from 18.3% to 5.9% when the predicted degree guides the reordering technique instead of the average degree. In the worst case, 32.9% of vertices are categorized as hot ones by HubCluster in the application CCMP on the dataset *or*, indicating that nearly one-third of vertices will be rearranged in memory. Conversely, only 4.8% of vertices are reordered by *LBR*, protecting the community feature and thus improving performance.

Fig. 6 (right) indicates the reordering overhead reduction of *LBR* over HubCluster. Since *LBR* reduces the scale of identified reordered vertices significantly, the reordering overhead is decreased as well. As shown in Fig. 6 (right), the average reordering overhead reduction is from 11% to 40% for different datasets and can be up to 41.8% in the application SSSP on the dataset *or*. The mean overhead reduction is 24.7% using our *LBR* framework compared with HubCluster.

5 Conclusion

Graph analysis plays an essential role in big data applications today. However, cache performance of graph processing is poor due to frequent cache misses. Multiple reordering techniques have been proposed to improve cache utilization. Nevertheless, existing methods are inefficient in both hot vertices identification and the way of reordering. We find that average degree is not ideal to categorize hot vertices and its additional traversal is costly. In this paper, we propose *LBR* that includes a prediction-based model to find a just-right degree, which predicts less hot vertices but provides better performance than the average degree. The new scheme has achieved significant performance improvements comparing with prior works. Moreover, the framework is easy to combine with any degree-based software graph optimizations. With our approach, the application performance, in terms of execution time, is 9.9% faster and the overhead is reduced by 24.7% compared with the best-performing reordering scheme.

Acknowledgments

This work was performed while M. Zou was a visiting Ph.D student at Illinois Institute of Technology.

References

[1] V. Balaji and B. Lucia, “When is graph reordering an optimization? Studying the effect of lightweight graph reordering across applications and input graphs,” in *Proc. IEEE Int. Symp. Workload Characterization*, 2018, pp. 203–214.
[2] P. Faldu, J. Diamond, and B. Grot, “A closer look at lightweight graph reordering,” in *Proc. IEEE Int. Symp. Workload Characterization*, 2019, pp. 1–13.
[3] A. G. and J. W. Liu, *Computer Solution of Large Sparse Positive Definite*. Englewood Cliffs, NJ, USA: Prentice Hall, 1981.
[4] M. Girvan and M. EJ Newman, “Community structure in social and biological networks,” in *Proc. Nat. Acad. Sci. USA*, 2002, pp. 7821–7826.
[5] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, “Graphbig: Understanding graph computing in the context of industrial solutions,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–12.
[6] D. F. Nettleton, “Data mining of social networks represented as graphs,” *Comput. Sci. Rev.*, vol. 7, pp. 1–34, 2013.
[7] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, “Making caches work for graph analytics,” in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 293–302.