

A Closer Look at Lightweight Graph Reordering

Priyank Faldu
The University of Edinburgh
priyank.faldu@ed.ac.uk

Jeff Diamond
Oracle Labs
jeff.diamond@oracle.com

Boris Grot
The University of Edinburgh
boris.grot@ed.ac.uk

Abstract—Graph analytics power a range of applications in areas as diverse as finance, networking and business logistics. A common property of graphs used in the domain of graph analytics is a *power-law distribution* of vertex connectivity, wherein a small number of vertices are responsible for a high fraction of all connections in the graph. These richly-connected (*hot*) vertices inherently exhibit high reuse. However, their sparse distribution in memory leads to a severe underutilization of on-chip cache capacity. Prior works have proposed lightweight skew-aware vertex reordering that places hot vertices adjacent to each other in memory, reducing the cache footprint of hot vertices and thus improving cache efficiency. However, in doing so, they may inadvertently destroy the inherent community structure within the graph, which may negate the performance gains achieved from the reduced footprint of hot vertices.

In this work, we study existing reordering techniques and demonstrate the inherent tension between reducing the cache footprint of hot vertices and preserving original graph structure. We quantify the potential performance loss due to disruption in graph structure for different graph datasets. We further show that reordering techniques that employ fine-grain reordering significantly increase misses in the higher level caches, even when they reduce misses in the last level cache.

To overcome the limitations of existing reordering techniques, we propose *Degree-Based Grouping (DBG)*, a novel lightweight reordering technique that employs a coarse-grain reordering to largely preserve graph structure while reducing the cache footprint of hot vertices. Our evaluation on 40 combinations of various graph applications and datasets shows that, compared to a baseline with no reordering, DBG yields an average application speed-up of 16.8% vs 11.6% for the best-performing existing lightweight technique.

Index Terms—cache, graph analytics, graph reordering, power-law graphs

I. INTRODUCTION

Graph analytics is an exciting and rapidly growing field with applications spanning diverse areas such as optimizing routes, uncovering latent relationships, pinpointing influencers in social graphs and many more. Graph analytics exhibit highly irregular memory access patterns. As a result, when processing large graphs, graph analytics tend to lack both spatial and temporal locality, leading to frequent misses in on-chip caches, which limits their performance [1], [2], [3], [4], [5], [6].

A distinguishing property of graph datasets common in many graph-analytic applications is that the vertex degrees follow a skewed *power-law* distribution, in which a small fraction of vertices have many connections while the majority of vertices have relatively few connections [5], [6], [7], [8], [9]. Graphs characterized by such a distribution are known as *natural* or *scale-free* graphs and are prevalent in a variety

of domains, including social networks, computer networks, financial networks, semantic networks and airline networks.

Power-law skew in the degree distribution means that a small set of vertices with the largest number of connections is responsible for a major share of memory accesses. The fact that these richly-connected vertices (henceforth referred to as *hot vertices*) comprise a small fraction of the overall footprint while exhibiting high reuse makes them prime candidates for caching. Meanwhile, the rest of the vertices (henceforth referred to as *cold vertices*) comprise a large fraction of the overall footprint while exhibiting low or no reuse.

For a typical graph application, each cache block (or line) contains multiple vertices, as vertex properties usually require just 4 to 16 bytes whereas cache block size in modern processors is typically 64 or 128 bytes. Since hot vertices are sparsely distributed throughout the memory space, they inevitably share cache blocks with cold vertices, leading to underutilization of a considerable fraction of useful cache capacity.

Prior works have proposed lightweight reordering techniques that leverage application-visible graph properties, such as skew in vertex degree distribution, to improve cache block utilization [5], [6]. We collectively refer to these techniques as *skew-aware* techniques. Skew-aware techniques reorder vertices in memory such that hot vertices are adjacent to each other in a contiguous memory region. As a result, each cache block is comprised of exclusively hot or cold vertices, reducing the total footprint (i.e., number of cache blocks) required to store hot vertices. Blocks that are exclusively comprised of hot vertices are far more likely to be retained in the cache due to higher aggregate hit rates, leading to higher utilization of existing cache capacity.

A straightforward way to pack vertices with similar degree into each cache block is to apply *Sort Reordering*, which sorts vertices in memory based on their degree. However, Sort is not always beneficial, because many real-world graph datasets exhibit a strong internal structure, e.g., clusters of webpages within the same domain in a web graph, or communities of common friends in a social graph [10], [11]. In such datasets, vertices within the same community that are accessed together often reside nearby in memory, exhibiting spatio-temporal locality that should be preserved. Fine-grain vertex reordering, such as Sort and Hub Sorting [5], destroys the spatio-temporal locality, which limits the effectiveness of such reordering techniques on graph datasets that exhibit structure.

In this work, we quantify potential performance loss

due to disruption of graph structure on various datasets. We further characterize locality at all three levels of the cache hierarchy, and show that all skew-aware techniques are generally effective at reducing LLC misses. However, the techniques that employ fine-grain reordering significantly disrupt graph structure, which increases misses in higher level caches. Our results highlight a tension between reducing the cache footprint of hot vertices and preserving graph structure, which limits the effectiveness of existing skew-aware reordering techniques.

To overcome the limitations of existing techniques, we propose *Degree-Based Grouping (DBG)*, a novel reordering technique that largely preserves graph structure while reducing the cache footprint of hot vertices. Like prior skew-aware techniques, DBG segregates hot vertices from the cold ones. However, to preserve existing graph structure, DBG employs *coarse-grain* reordering. DBG partitions vertices into a small number of groups based on their degree but maintains the original relative order of vertices within each group. As DBG *does not sort* vertices within any group to minimize structure disruption, DBG also incurs a very low reordering overhead.

To summarize, we make the following contributions:

- We study existing skew-aware reordering techniques on a variety of multi-threaded graph applications processing varied datasets. Our characterization reveals the inherent tension between reducing the cache footprint of hot vertices and preserving graph structure.
- We propose DBG, a new skew-aware reordering technique that employs lightweight coarse-grain reordering to largely preserve existing graph structure while reducing the cache footprint of hot vertices.
- Our evaluation on a real machine shows that DBG outperforms existing skew-aware techniques. Averaging across 40 datapoints, DBG yields a speed-up of 16.8%, vs 11.6% for the best-performing existing skew-aware technique over the baseline with no reordering.

II. BACKGROUND AND MOTIVATION

A. Properties of Natural Graphs

Natural graphs have two distinguishing properties – skew in their degree distribution and community structure.

Skew in Degree Distribution: Natural graphs exhibit *skew* in their degree distribution [5], [6], [7], [8], [9]. The skew follows a power-law, with the vast majority of the vertices having relatively few edges and a small fraction of vertices featuring a large number of edges. Such skewed distribution is prevalent in many domains and found, for instance, in nodes in large-scale communication networks (e.g., the internet), web pages in web graphs, and individuals in social graphs.

Table I quantifies the skew for the datasets evaluated in this work (Table IX in Sec. V contains more details about these datasets). For example, in the *sd* dataset, 11% of total vertices are classified as hot vertices in terms of their in-degree (13% for out-degree) distribution. These hot vertices are connected to 88% of all in-edges (88% of all out-edges) in the graph.

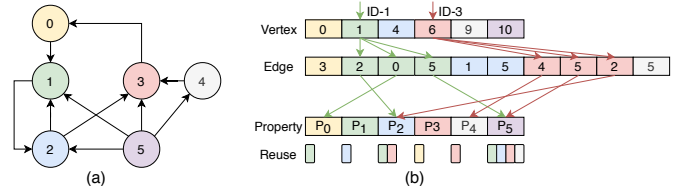


Fig. 1: (a) An example graph. (b) CSR format encoding in-edges. Elements of the same colors in all arrays, correspond to the same destination vertex. Number of bars (labeled *Reuse*) below each element of the Property Array shows the number of times an element is accessed in one full iteration, where the color of a bar indicates the vertex making an access.

		<i>kr</i>	<i>pl</i>	<i>tw</i>	<i>sd</i>	<i>lj</i>	<i>wl</i>	<i>fr</i>	<i>mp</i>
In	Hot Vertices (%)	9	16	12	11	25	12	24	10
Edges	Edge Coverage (%)	93	83	84	88	81	88	86	80
Out	Hot Vertices (%)	9	13	10	13	26	20	18	12
Edges	Edge Coverage (%)	93	88	83	88	82	94	92	81

TABLE I: Rows #2 and #4 show number of hot vertices (vertices with degree equal or greater than the average degree) as a percentage of total vertices, with respect to in-edges and out-edges, respectively; the higher the skew, the lower the percentage. Rows #3 and #5 show number of in-edges and out-edges connected to the hot vertices as a percentage of total edges, respectively; the higher the skew, the higher the percentage.

Similarly, in other datasets, 9%-26% of vertices are classified as hot vertices, which are connected to 80%-94% of all edges.

Community Structure: Real-world graphs often feature clusters of highly interconnected vertices such as communities of common friends in a social graph [10], [11]. Such community structure is often captured by vertex ordering within a graph dataset by placing vertices from the same community nearby in the memory space. At runtime, vertices that are placed nearby in memory are typically processed within a short time window of each other. Thus, by placing vertices from the same community nearby in memory, both temporal and spatial locality is improved at the cache block level for such datasets.

B. Graph Processing Basics

The majority of shared-memory graph processing frameworks are written using a vertex-centric model, wherein an application computes some information for each vertex based on the properties of its neighboring vertices [12], [13], [14], [15], [16], [17]. Applications may perform pull- or push-based computations, or both. In pull-based computations, a vertex pulls (reads) property values of its in-neighbors (vertices whose edges point to this vertex). In push-based computations, a vertex pushes (writes) its property values to its out-neighbors (vertices pointed to by the edges of this vertex). This process may be iterative, and all or only a subset of vertices may participate in a given iteration.

The *Compressed Sparse Row (CSR)* format is commonly used to represent graphs in a storage-efficient manner. CSR

Dataset	<i>kr</i>	<i>pl</i>	<i>tw</i>	<i>sd</i>	<i>lj</i>	<i>wl</i>	<i>fr</i>	<i>mp</i>
Avg.	1.3	1.6	1.5	1.8	3.5	3.1	2.7	2.6

TABLE II: Average number of hot vertices per cache block. Calculation assumes 8 bytes per vertex and 64 bytes per cache block, and counts only cache blocks that contain at least one hot vertex. As a result, any cache block can contain between 1–8 hot vertices.

encodes in-edges for pull-based computations and out-edges for push-based computations. In this discussion, we focus on pull-based computations and note that the observations also hold for push-based computations. CSR uses a pair of arrays, *Vertex* and *Edge*, to encode the graph. For every vertex, the *Vertex Array* maintains an index that points to its first in-edge in the *Edge Array*. The *Edge Array* stores all in-edges, grouped by destination vertex ID. For each in-edge, the *Edge Array* entry stores the associated source vertex ID.

Graph applications additionally use one or more *Property Arrays* to hold partial or final results for every vertex. For example, the *Pagerank* application maintains two rank values for every vertex; one computed from the previous iteration and one being computed in the current iteration. Implementation may use either two separate arrays (each storing one rank per vertex) or may use only one array (storing two ranks per vertex). Fig. 1(a) shows a simple graph and Fig. 1(b) shows its CSR representation for pull-based computations along with one *Property Array*.

C. Cache Behavior in Graph Analytics

At the most fundamental level, a graph application computes a property for a vertex based on the properties of its neighbors. To find the neighboring vertices, an application traverses the portion of the *Edge Array* corresponding to a given vertex, and then accesses elements of the *Property Array* corresponding to these neighboring vertices. Fig. 1(b) shows the elements accessed while processing vertex ID-1 and ID-3.

As the figure shows, each element of the *Vertex* and *Edge* Arrays is accessed exactly once during an iteration, exhibiting no temporal locality at LLC. In contrast, the *Property Array* does exhibit temporal reuse. However, reuse is not consistent for all elements, being proportional to the number of out-edges for pull-based computations. Thus, high out-degree vertices exhibit high reuse. Fig. 1(b) shows the reuse for high out-degree (i.e., hot) vertices P_2 and P_5 of the *Property Array* assuming pull-based computations; the other elements do not exhibit reuse. The same observations apply to high in-degree vertices for push-based computations.

D. Poor Cache Efficiency for the Property Array

As discussed in the previous section, the *Property Array* exhibits reuse and thus, should be the prime target for caching. However, not all elements of the array exhibit high reuse. Elements associated with hot vertices are the ones responsible for large amount of reuse. Despite such high reuse, on-chip cache capacity is severely underutilized as hot vertices are sparsely distributed throughout the memory space.

A cache block is typically comprised of multiple vertices as the properties associated with a vertex are much smaller than the size of a cache block. Thus, inevitably, hot vertices share space in a cache block with cold vertices, which poses a challenge for cache efficiency. Even when such a cache block is retained in the cache, it leads to underutilization of cache capacity as a considerable fraction of the cache block is occupied by cold vertices that exhibit low or no reuse.

Table II shows the average number of hot vertices per cache block, assuming typical values of 8 bytes per vertex and 64 bytes per cache block. While, at best, 8 hot vertices can be packed together in a cache block, in practice, only 1.3 to 3.5 hot vertices are found per cache block across the datasets. As the footprint (i.e., number of cache blocks) to store hot vertices is inversely proportional to the average number of hot vertices per cache block, the data shows significant opportunity in reducing the cache footprint of hot vertices, and in turn, improving cache efficiency.

E. Vertex Reordering to Improve Cache Efficiency

The order of vertices in memory is under the control of a graph application. Thus, the application can reorder vertices in memory before processing a graph to improve cache locality. To accomplish this, researchers have proposed various reordering techniques [4], [5], [6], [18], [19], [20], [21], [22], [23], [24]. Reordering techniques only relabel vertices (and edges), which does not alter the graph itself and does not require any changes to the graph algorithms. Following the relabeling, vertices (and edges) are reordered in memory based on the new vertex IDs.

The most powerful reordering techniques like Gorder [4] leverage community structure, typically found in real-world graphs, to improve spatio-temporal locality. Gorder comprehensively analyzes the vertex connectivity and reorders vertices such that vertices that share common neighbors, and thus are likely to belong to the same community, are placed nearby in memory. While Gorder is effective at improving application performance, it requires a staggering reordering time that is often multiple orders of magnitude higher than the application runtime, rendering Gorder impractical [6].

To keep reordering cost affordable, recent works have argued for skew-aware reordering techniques that reorder vertices solely based on vertex degrees [5], [6]. By relying on lightweight analysis, these techniques can speed-up applications even after accounting for the reordering time.

Skew-aware reordering techniques seek to reduce the cache footprint of hot vertices to improve cache efficiency. However, as a side-effect of reordering, they may destroy a graph’s community structure, which could negate the performance gains achieved from the reduced footprint of hot vertices. Thus, there exists a tension between reducing the cache footprint of hot vertices and preserving graph structure when reordering vertices. In this context, we next detail existing skew-aware reordering techniques.

III. SKEW-AWARE REORDERING TECHNIQUES

A. Objectives for High Performance Reordering

In order to provide high performance for graph applications, skew-aware reordering techniques should achieve all of the following three objectives:

O1. Low Reordering Time: Reordering time plays a crucial role in deciding whether a technique is viable in providing end-to-end application performance after accounting for the reordering time. Lower reordering time facilitates amortizing the reordering overhead in a fewer graph traversals.

O2. High Cache Efficiency: As explained in Sec. II-D, a cache block is comprised of multiple vertices. Problematically, hot vertices are sparsely distributed throughout the memory space, which leads to cache blocks containing vertices with vastly different degrees. To address this, vertex reordering should ensure that hot vertices are placed adjacent to each other in the memory space, thus reducing the cache footprint of hot vertices, and in turn, improving cache efficiency.

O3. Structure Preservation: As explained in Sec. II-A, many real-world graph datasets have vertex ordering that results in high spatio-temporal cache locality. For such datasets, vertex reordering should ensure that the original structure is preserved as much as possible. If structure is not preserved, reordering may adversely affect the locality, negating performance gains achieved from the reduced footprint of hot vertices.

B. Implications of Not Preserving Graph Structure

In this section, we characterize how important it is to preserve graph structure for different datasets. To quantify potential performance loss due to reduction in spatio-temporal locality arising from reordering, we randomly reorder vertices, which decimates any existing structure. Randomly reordering all vertices would cause a slowdown for two potential reasons: (1) By destroying graph structure, thus reducing spatio-temporal locality. (2) By further scattering hot vertices in memory, thus increasing the cache footprint of hot vertices. To isolate performance loss due to the former, we also evaluate random reordering at a cache block granularity. In such a reordering, cache blocks (not individual vertices) are randomly reordered in memory, which means that the vertices within a cache block are moved as a group. As a result, the cache footprint of hot vertices is unaffected, and any change in performance can be directly attributed to a change in graph structure. Fig. 2(a) illustrates vertex placement in memory after Random Reordering at a vertex and at a cache block granularity.

Fig. 3 shows performance slowdown for Random Reordering for the Radii application on all datasets listed in Table IX. The figure shows four configurations – Random Vertex (RV) that reorders at a granularity of one vertex and Random Cache Block- n (RCB- n) that reorders at a granularity of n cache blocks, where n is 1, 2 or 4.

Performance difference between RV and RCB-1 is very large for the four right-most datasets. Recall from Table II that these datasets have relatively high number of hot vertices per

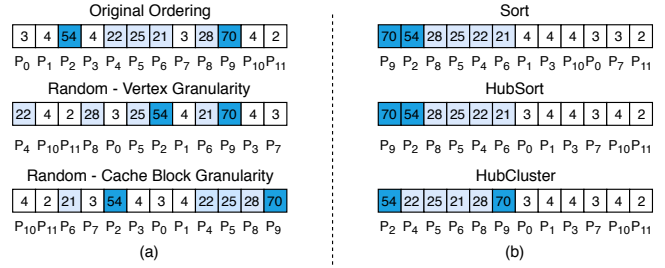


Fig. 2: Vertex ordering in memory for different techniques. Vertex degree is shown inside the box while original vertex ID is shown below the box. Hot vertices (degree ≥ 20) are shown in color. Hottest among the hot vertices (degree ≥ 40) are shown in darker shade. Finally, Random (Cache Block Granularity) assumes two vertices per cache block.

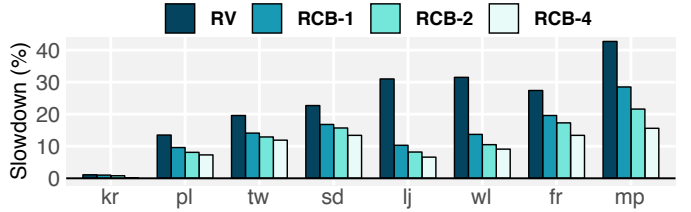


Fig. 3: Application slowdown after random reordering at different granularity for the Radii application. The lower the bar, the better the application performance.

cache block. RV scatters the hot vertices in memory, incurring large slowdowns for these datasets.

Performance slowdown for RCB-1 is significant on all real-world datasets (i.e., all but kr), and ranges from 9.6% to 28.5%. This slowdown can be attributed to disruption in spatio-temporal locality for the real-world datasets, confirming existence of community structure in the original ordering of the datasets. As reordering granularity increases, disruption in graph structure reduces, which also reduces the slowdown. For example, on the mp dataset, most affected by the Random Reordering, performance slowdown is 28.5% for RCB-1, which reduces to 21.6% for RCB-2 and 15.6% for RCB-4.

Results for kr , the only synthetic dataset in the mix, are in stark contrast with that of the real-world datasets. As kr is generated synthetically, kr does not have any structure in the original ordering. Thus, the performance on the kr dataset is largely oblivious to random reordering at any granularity.

The results show that the real-world graph datasets exhibit some structure in their original ordering, which, if not preserved, is likely to adversely affect the performance. The results also indicate that structure can be largely preserved by applying reordering at a coarse granularity.

C. Limitations of Skew-Aware Reordering Techniques

This section describes the existing skew-aware techniques and how they fare in achieving the three objectives listed in Sec. III-A. As skew-aware techniques solely rely on vertex degrees for reordering, they all incur relatively low reordering

Per-Vertex Property	<i>kr</i>	<i>pl</i>	<i>tw</i>	<i>sd</i>	<i>lj</i>	<i>wl</i>	<i>fr</i>	<i>mp</i>
8 Bytes	44	51	56	80	9	16	115	39
16 Bytes	88	102	112	160	18	32	230	78

TABLE III: Cache size in MB needed to store all hot vertices, for 8 and 16 bytes per property, respectively. Vertex is classified hot if its degree is equal or greater than the average degree of the dataset.

Degree Range	$[1\bar{A}, 2\bar{A})$	$[2\bar{A}, 4\bar{A})$	$[4\bar{A}, 8\bar{A})$	$[8\bar{A}, 16\bar{A})$	$[16\bar{A}, 32\bar{A})$	$[32\bar{A}, \infty)$
Vertices (%)	45%	28%	15%	7%	3%	2%
Footprint	35.8	22.3	12.0	5.7	2.2	1.8

TABLE IV: Degree distribution of hot vertices for the *sd* dataset, whose Average Degree (\bar{A}) is 20. Row #2 shows percentage of total hot vertices while row #3 shows the footprint requirement in MB.

time, achieving objective O1. However, for the two remaining objectives – reducing the cache footprint of hot vertices and preserving existing graph structure – existing techniques trade one for the other, hence failing to achieve at least one of the two objectives.

Sort reorders vertices based on the descending order of their degree. Sort requires the least possible number of cache blocks to store hot vertices without explicitly classifying individual vertices as hot or cold. However, sort reorders all vertices, which completely destroys the original graph structure. Fig. 2(b) shows vertex placement in memory after the Sort Reordering.

Hub Sorting [5] (also known as Frequency-based Clustering) was proposed as a variant of Sort that aims to preserve some structure while reducing the cache footprint of hot vertices. Hub Sorting uses a threshold (typically, average degree of the dataset) to classify vertices as hot or cold, and only sorts the hot vertices.

Hub Sorting does preserve partial structure by not sorting the cold vertices, but problematically, the hot vertices are fully sorted. While hot vertices constitute a smaller fraction compared to the cold ones, recall from Table I that hot vertices account for up to 26% of the total vertices. Moreover, hot vertices are connected to the vast majority of edges (80%-94%) and thus, are responsible for the majority of reuse. Consequently, preserving structure for hot vertices is also important, at which Hub Sorting fails.

Hub Clustering [6] is a variant of Hub Sorting that only segregates hot vertices from the cold ones but does not sort them. While Hub Clustering was proposed as an alternative to Hub Sorting that has lower reordering time, we note that Hub Clustering is also better than Hub Sorting at preserving existing graph structure as Hub Clustering does not sort any vertices. However, by not sorting hot vertices, Hub Clustering sacrifices significant opportunity in improving cache efficiency as discussed next.

For large graph datasets, it is unlikely that all hot vertices fit in the LLC. For example, the *sd* dataset requires at least 80MB to store all hot vertices assuming only 8 bytes per vertex (refer to Table III for requirements of the remaining datasets).

The required capacity significantly exceeds typical LLC size of commodity server processors. As a result, all hot vertices compete for the limited LLC capacity, causing cache thrashing.

Fortunately, not all hot vertices have similar reuse, as vertex degree varies vastly among hot vertices. Table IV shows the degree distribution for just the hot vertices of the *sd* dataset. Each column in the table represents a degree range as a function of \bar{A} , the average degree of the dataset. For instance, the first column covers vertices whose degree ranges from \bar{A} to $2\bar{A}$; these are the lowest-degree vertices among the hot ones (recall that a vertex is classified as hot if its degree is equal or greater than \bar{A}). For a given range, the table shows number of vertices (as a percentage of total hot vertices) whose degree is within that range. The table also shows cache capacity needed for those many vertices assuming 8 bytes per vertex property. Unsurprisingly, given the power-law degree distribution, the table shows that the least-hot vertices are the most numerous, representing 45% of all hot vertices and requiring 35.8MB capacity, yet likely exhibiting the least reuse among hot vertices. In contrast, vertices with degree above $8\bar{A}$ (three right-most columns) are the hottest of all, constituting just 12% of total hot vertices (< 10 MB footprint). Naturally, these hottest vertices are the ones that should be retained in the cache. However, by not sorting hot vertices, Hub Cluster fails to differentiate between the most- and the least-hot vertices, hence denying the hottest vertices an opportunity to stay in the cache in the presence of cache thrashing.

To summarize, Sort achieves the maximum reduction in the cache footprint of hot vertices. However, in doing so, Sort completely decimates existing graph structure. Hub Sorting and Hub Clustering both classify vertices as hot or cold based on their degree and preserve the structure for cold vertices. However, in dealing with hot vertices, they resort to inefficient extremes. At one extreme, Hub Sorting employs fine-grain reordering that sorts all hot vertices, destroying existing graph structure. At the other extreme, Hub Clustering does not apply any kind of reordering among hot vertices, sacrificing significant opportunity in improving cache efficiency.

IV. DEGREE-BASED GROUPING (DBG)

To address the limitations of existing skew-aware reordering techniques, we propose *Degree-Based Grouping (DBG)*, a novel skew-aware technique that applies coarse-grain reordering such that each cache block is exclusively comprised of vertices having similar degree, and in turn, similar hotness, while also preserving graph structure at large.

Unlike Hub Sorting and Hub Clustering, which rely on a single threshold to classify vertices as hot or cold, DBG employs a simple binning algorithm to *coarsely* partition vertices into different groups (or bins) based on their hotness level. Groups are assigned exclusive degree ranges such that the degree of any vertex falls within a degree range of exactly one group. Within each group, DBG maintains the original relative order of vertices to preserve graph structure at large. To keep the reordering time low, DBG maintains only a small

$G(V, E)$ where Graph G has V vertices and E edges.
Input: Degree Distribution $D[]$, where $D[v]$ is degree of vertex v .
Output: Mapping $M[]$, where $M[v]$ is the new ID of vertex v .
DBG: Binning algorithm to reorder vertices into K groups ($K > 0$).

- 1: Assign contiguous range $[P_k, Q_k)$ to every Group_k such that,
 - $Q_1 > \max(D[])$ &
 - $P_K \leq \min(D[])$ &
 - $Q_{k+1} = P_k < Q_k$, for every $k < K$
- 2: For every vertex v from 1 to V
 - Append v to the Group_k for which $D[v] \in [P_k, Q_k)$.
- 3: Assign new IDs to all vertices as follows:
 - $\text{id} := 1$
 - For every Group_k from 1 to K
 - For every vertex v in Group_k
 - $M[v] := \text{id}++$, where v is the original ID

Listing 1: DBG algorithm. Degree can be in-degree or out-degree or sum of both.



Fig. 4: Vertex ordering in memory after DBG. In this example, DBG partitions vertices into three groups with degree ranges $[0, 20)$, $[20, 40)$ and $[40, 80)$. DBG maintains a relative order of vertices within a group. As a result, many vertices are placed nearby the same vertices as before the reordering such as vertex sets (P_4, P_5, P_6) , (P_0, P_1) and (P_{10}, P_{11}) .

number of groups and does not sort vertices within any group. Listing 1 presents the formal DBG algorithm.

To assign degree ranges to different groups, DBG leverages the power-law distribution of vertex connectivity in natural graphs. For example, recall Table IV, which shows distribution of vertices across different degree ranges. Vertices with the smallest degree range constitute the largest fraction of hot vertices. As degree range doubles, the number of vertices are roughly halved, exhibiting the power-law distribution. Thus, geometrically-spaced degree ranges provide a natural way to segregate vertices with different levels of hotness. At the same time, using such wide ranges to partition vertices facilitates reordering at a very coarse granularity, preserving structure at large. Meanwhile, by not sorting vertices within any group, DBG incurs a very low reordering time. Thus, DBG successfully achieves all three objectives listed in Sec. III-A. Fig. 4 shows vertex placement in memory after the DBG Reordering, for a synthetic example.

Finally, we note that the DBG algorithm (Listing 1) provides a general framework to understand trade-offs between reducing the cache footprint of hot vertices and preserving graph structure just by varying a number of groups and their degree ranges. Indeed, Table V shows how different skew-aware techniques can be implemented using the DBG algorithm. For example, Hub Clustering can be viewed as an implementation of DBG algorithm with two groups – one containing hot vertices and another one containing cold

Reordering	#groups	Degree Range
Sort	$M+1$	$[n, n+1)$ where $n \in [0, M]$
Hub Sorting	$M-A+2$	$[0, A)$, $[n, n+1)$ where $n \in [A, M]$
Hub Clustering	2	$[0, A)$, $[A, M]$
DBG	$\lfloor \log_2 \frac{M}{C} \rfloor + 2$	$[0, C)$, $[2^n C, 2^{n+1} C)$ where $n \in [0, \lfloor \log_2 \frac{M}{C} \rfloor]$

TABLE V: Implementation of various skew-aware techniques using DBG algorithm. A is the average and M is the maximum degree of the dataset. For DBG, C is some threshold such that $0 < C < M$.

Technique	Structure Preservation	Reordering Time	Net Performance
Sort	✗	✓	✓
Hub Sorting [5]	✓	✓	✓
Hub Clustering [6]	✓✓	✓✓	✓
DBG (proposed)	✓✓	✓✓	✓✓
Gorder [4]	✓✓	✗	✗

TABLE VI: Qualitative performance of different reordering techniques for graph analytics on natural graphs.

vertices. Similarly, Sort can be seen as an implementation of DBG algorithm with as many number of groups as many unique degrees a given dataset has. Consequently, for a given unique degree, the associated group contains all vertices having the same degree, effectively sorting vertices by their degree. In general, as the number of groups is increased, the degree range gets narrower and vertex reordering gets finer, causing more disruption to existing structure. Table VI qualitatively compares DBG to prior techniques.

V. METHODOLOGY

A. Graph Processing Framework, Applications and Datasets

For the evaluation, we use *Ligra* [12], a widely used shared-memory graph processing framework that supports both pull- and push-based computations, including switching from pull to push (and vice versa) at the start of a new iteration. We evaluate various reordering techniques using five iterative graph applications listed in Table VII, on eight graph datasets listed in Table IX, resulting in 40 datapoints for each technique.

We obtained the source code for the graph applications from *Ligra* [12]. Implementation of the graph applications is unchanged except for an addition of an array to keep a mapping between the vertex ID assignments before and after reordering. The mapping is needed to ensure that root-dependent traversal applications running on the reordered graph datasets use the same root as the baseline execution running on the original graph dataset. We compile the applications using g++6.4 with O3 optimization level on Ubuntu 14.04.1 booted with Linux kernel 4.4.0-96-lowlatency and use OpenMP for parallelization. To utilize memory bandwidth from both sockets, we run every application

Application	Brief description
Betweenness Centrality (BC)	finds the most central vertices in a graph by using a BFS kernel to count the number of shortest paths passing through each vertex from a given root vertex.
Single Source Shortest Path (SSSP)	computes shortest distance for vertices in a weighted graph from a given root vertex using the Bellman Ford algorithm.
Pagerank (PR)	is an iterative algorithm that calculates ranks of vertices based on the number and quality of incoming edges to them [25].
PageRank-Delta (PRD)	is a faster variant of PageRank in which vertices are active in an iteration only if they have accumulated enough change in their PageRank score.
Radii Estimation (Radii)	estimates the radius of each vertex by performing multiple parallel BFS's from a small sample of vertices [26].

TABLE VII: A list of graph applications evaluated in this work.

Graph App.	Computation Type	Per-Vertex Property Size (bytes)		Degree Type used for Reordering
		All Properties	Only Properties with Irregular Accesses	
BC	pull-push	17	8	out
SSSP	push-only	8	8	in
PR	pull-only	20	12	out
PRD	push-only	20	8	in
Radii	pull-push	20	8	out

TABLE VIII: Properties of graph applications. In addition to the vertex properties, all graph applications require 4 bytes to encode a vertex and 8 bytes to encode an edge.

under NUMA interleave memory allocation policy. Table VIII lists various properties for the Ligra implementation of the evaluated graph applications.

B. Evaluation Platform and Methodology

Evaluation is done on a dual-socket server with two Broadwell based *Intel Xeon CPU E5-2630* [27], each with 10 cores clocked at 2.2GHz and a 25MB shared LLC. Hyper-threading is kept on, exposing 40 hardware execution contexts across both CPUs. Server has 128GB of DRAM provided by eight DIMMs clocked at 2133MHz. Applications use 40 threads, and the threads are pinned to avoid performance variations due to OS scheduling. To further reduce sources of performance variation, DVFS features are disabled. Finally, *Transparent Huge Pages* is kept on to reduce TLB misses.

We evaluate each reordering technique on every combination of graph applications and graph datasets 11 times, and record the average runtime of 10 executions, excluding the timing of the first execution to allow the caches to warm up. We report the speed-up over the entire application runtime (with and without reordering cost) but exclude the graph loading time from the disk. For iterative applications – PR and PRD – we run them until convergence and consider the aggregate runtime over all iterations. For root-dependent traversal applications – SSSP and BC – we

Dataset	Vertex Count	Edge Count	Avg. Degree	Type	Original Ordering
Kron (<i>kr</i>) [14]	67M	1,323M	20	Synthetic	Unstructured
PLD (<i>pl</i>) [28]	43M	623M	15	Real	Unstructured
Twitter (<i>tw</i>) [29]	62M	1,468M	24	Real	Unstructured
SD (<i>sd</i>) [28]	95M	1,937M	20	Real	Unstructured
LiveJournal (<i>lj</i>) [30]	5M	68M	14	Real	Structured
WikiLinks (<i>wl</i>) [31]	18M	172M	9	Real	Structured
Friendster (<i>fr</i>) [32]	64M	2,147M	33	Real	Structured
MPI (<i>mp</i>) [33]	53M	1,963M	37	Real	Structured

TABLE IX: Properties of the evaluated graph datasets. We empirically label those datasets as *structured* on which Random Reordering (RV) causes more than 25% slowdown (Fig. 3).

Dataset	Vertex Count	Edge Count	Avg. Degree	Type
Uniform (<i>uni</i>) [34]	50M	1,000M	20.0	Synthetic
USA Road Network (<i>road</i>) [35]	24M	29M	1.2	Real

TABLE X: Properties of the no-skew graph datasets. The *uni* dataset is generated using R-MAT [36] methodology with parameter values of $A=B=C=25$.

run them from eight different root vertices for each input dataset and consider the aggregate runtime over all eight traversals. Finally, we note that the application runtime is relatively stable across executions. For each reported datapoint, coefficient of variation is at most 2.3% for PRD and at most 1.6% for other applications.

C. Reordering Techniques

We evaluate DBG and compare it with all three existing skew-aware techniques described in Sec. III-C (Sort, HubSort [5] and HubCluster [6]) along with Gorder [4] – the state-of-the-art structure-aware reordering technique.

For Gorder, we use the source code available from the authors. As Gorder is only available in a single-thread implementation, while reporting the reordering time of Gorder for a given dataset, we optimistically divide the reordering time by 40 (maximum number of threads supported on the server) to provide a fair comparison with skew-aware techniques whose reordering implementation is fully parallelized.

For DBG, we use 8 groups with the ranges $[32\mathbb{A}, \infty)$, $[16\mathbb{A}, 32\mathbb{A})$, $[8\mathbb{A}, 16\mathbb{A})$, $[4\mathbb{A}, 8\mathbb{A})$, $[2\mathbb{A}, 4\mathbb{A})$, $[1\mathbb{A}, 2\mathbb{A})$, $[\mathbb{A}/2, \mathbb{A})$ and $[0, \mathbb{A}/2)$, where \mathbb{A} is the average degree of the graph dataset. Note that we also partition cold vertices into two groups. We developed a multi-threaded implementation of DBG, which is available at <https://github.com/faldupriyank/dbg>.

Finally, we implement HubSort and HubCluster using the DBG algorithm as shown in Table V. We found our implementations to be more effective than the original implementations (referred to as HubSort-O and HubCluster-O) provided by the authors of HubCluster. Fig. 5 shows application speed-up over the baseline with no reordering. Table XI shows reordering time normalized to that of Sort. As our implementation of both techniques provides better speed-

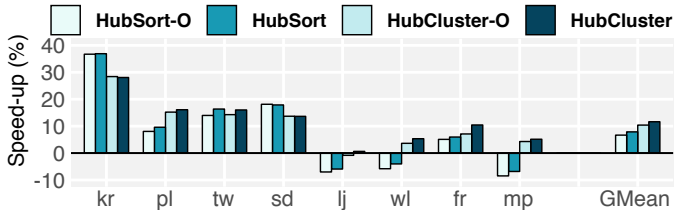


Fig. 5: Application speed-up over the baseline with no reordering. Techniques with suffix O use their original implementations whereas techniques without any suffix are implemented using DBG algorithm as per Table V. The bars for the datasets show geometric mean of speed-ups across five applications for a given dataset.

Technique	kr	pl	tw	sd	lj	wl	fr	mp
HubSort-O	1.02	1.04	1.01	1.02	1.09	0.79	1.04	1.01
HubSort	0.80	0.82	0.84	0.84	0.87	0.91	0.90	0.89
HubCluster-O	0.78	0.79	0.81	0.81	0.78	0.56	0.88	0.87
HubCluster	0.77	0.74	0.81	0.78	0.76	0.81	0.84	0.82

TABLE XI: Reordering time for existing skew-aware techniques, normalized to that of Sort. Lower is better.

ups and lower reordering time, we use our implementations in the main evaluation.

VI. EVALUATION

In this section, we evaluate the effectiveness of DBG against the state-of-the-art reordering techniques. In Sec. VI-A, we compare the application speed-up for these techniques *without considering the reordering time*. In Sec. VI-B and Sec. VI-C, we analyze different levels of cache hierarchy to understand the sources of performance variation. Subsequently, to understand the effect of the reordering time on end-to-end performance, we compare the application speed-up *after accounting for the reordering time* in Sec. VI-D.

A. Performance Excluding Reordering Time

Fig. 6 shows application speed-up excluding reordering time for various datasets. Averaging across all 40 datapoints (combining all structured and unstructured), DBG provides 16.8% speed-up over the baseline with no reordering, outperforming all existing skew-aware techniques: Sort (8.4%), HubSort (7.9%) and HubCluster (11.6%). Gorder, which comprehensively analyzes graph structure, yields 18.6% average speed-up, marginally higher than that of DBG. We next analyze performance variations across datasets and applications.

1) *Unstructured vs Structured*: As shown in Fig. 6(a), on unstructured datasets, all reordering techniques provide positive speed-ups for all applications except for PRD. Sec. VI-C explains the reasons for slowdowns for the PRD application. Among skew-aware techniques, DBG provides the highest average speed-up of 28.1% in comparison to 22.1% for Sort, 19.8% for HubSort and 18.3% for HubCluster.

On synthetic dataset *kr*, all techniques except HubCluster provide similar speed-ups as *kr* is largely insensitive

to structure preservation. Similarly, on other unstructured datasets, as hot vertices are relatively more scattered in memory (see Table II), the benefit of vertex packing outweighs potential slowdown due to structure disruption. Thus, Sort, despite completely decimating the original graph structure, outperforms HubSort and HubCluster on more than half of the datapoints. Meanwhile, DBG, which also preserves graph structure while reducing the cache footprint of hot vertices, provides higher performance than Sort on more than half of the datapoints.

Overall, DBG provides more than 30% speed-up over the baseline on half of the datapoints. DBG outperforms or matches skew-aware techniques on nearly all datapoints. Over the best performing skew-aware technique, DBG provides the highest performance improvements on the SSSP application, with maximum speed-up of 18.0% on the *tw* dataset.

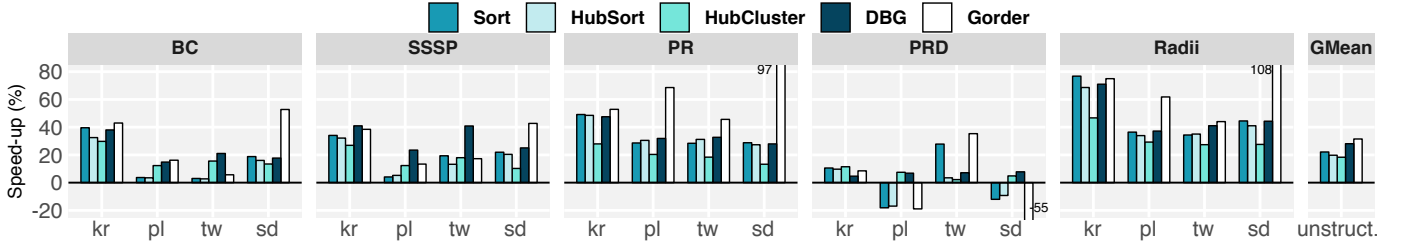
Structured datasets exhibit high spatio-temporal locality in their original ordering. Thus, any technique that does not preserve the graph structure is likely to yield only a marginal speed-up, if any. Among skew-aware techniques, DBG provides the highest average speed-up of 6.5% in comparison to -3.7% for Sort, -2.8% for HubSort and 5.3% for HubCluster.

On structured datasets, performance gains from reduction in footprint of hot vertices are negated by disruption in graph structure. Thus, Sort and HubSort, which preserve graph structure the least, cause slowdown (up to 38.4%) on more than half of the datapoints. DBG, in contrast, successfully avoids slowdown on almost all datapoints and causes a marginal slowdown (up to 4.9%) only on 4 datapoints.

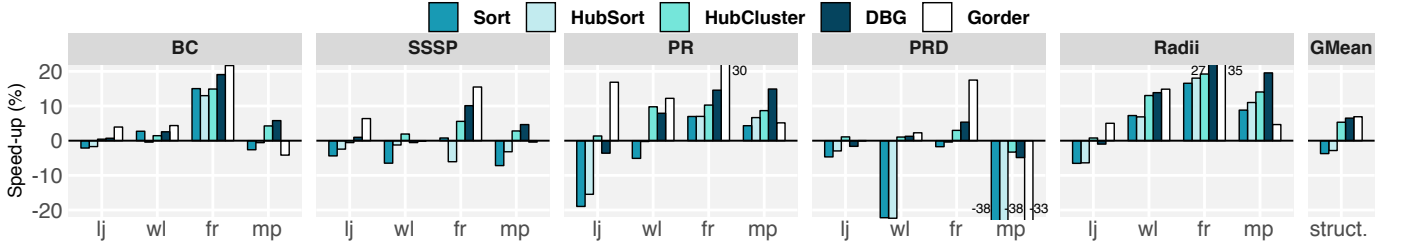
2) *DBG vs Gorder*: Gorder comprehensively analyzes vertex connectivity to improve cache locality whereas DBG reorders vertices solely based on their degrees. Thus, it is expected for Gorder to outperform DBG (and other skew-aware techniques). On average, Gorder yields a speed-up of 31.5% (vs 28.1% for DBG) for unstructured datasets and 6.9% (vs 6.5% for DBG) for structured datasets.

Specifically, difference in speed-ups for DBG and Gorder is very small for datasets *kr*, *tw*, *wl* and *mp*. These datasets have relatively small clustering coefficient compared to other datasets [37], which makes it difficult for Gorder to approximate suitable vertex ordering. On other datasets, Gorder provides significantly higher speed-ups than any skew-aware techniques. Problematically, Gorder incurs staggering reordering overhead and thus, causes severe slowdowns when accounted for its reordering time (see Sec. VI-D), making it impractical.

3) *Reordering on No-Skew graphs*: In this section, we evaluate the effect of reordering techniques on graph datasets that have no skew. Skew-aware techniques are not expected to provide significant speed-up for these datasets due to lack of skew in their degree distribution. More importantly, these techniques are also not expected to cause any significant slowdown due to a nearly complete lack of locality in the baseline ordering to begin with.



(a) Unstructured datasets.



(b) Structured datasets.

Fig. 6: Application speed-up (excluding reordering time) for reordering techniques over the baseline with no reordering.

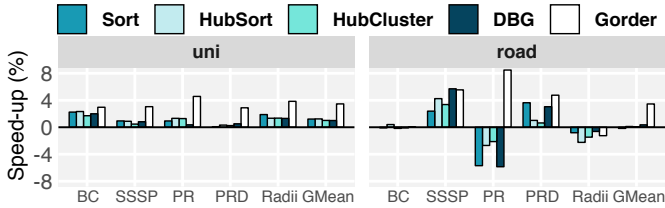


Fig. 7: Effect of reordering techniques on graph datasets having no skew.

Fig. 7 shows speed-ups for reordering techniques on two datasets – *uni* and *road* – listed in Table X. As expected, all skew-aware techniques have a relatively neutral effect, with an average change in execution time within 1.2% on the *uni* dataset and within 0.4% on the *road* dataset. Meanwhile, Gorder yields slightly more speed-up (3.5% on both *uni* and *road* datasets), as it can exploit fine-grain spatio-temporal locality, which is not entirely skew dependent.

B. MPKI Across Cache Levels

In this section, we explain the sources of performance variations for different reordering techniques by analyzing their effects on all three levels of the cache hierarchy. Fig. 8 plots *Misses Per Kilo Instructions (MPKI)* for L1, L2 and L3 cache, measured using hardware performance counters, for the PR application as a representative example.

In the baseline with the original ordering, on all datasets except *lj* and *wl*, L1 MPKI is more than 100 (i.e., at least 1 L1 miss for every 10 instructions on average), which confirms the memory intensive nature of graph applications. For the original ordering, L2 MPKI is only marginally lower than L1 MPKI across datasets, which shows that almost all memory accesses that miss in the L1 cache also miss in the L2 cache.

As L3 cache is significantly larger than L2 cache, L3 MPKI is much lower than L2 MPKI; nonetheless, L3 MPKI is very high for the original ordering, ranging from 56.2 to 82.9 across large datasets (excluding *lj* and *wl*).

While all skew-aware techniques target L3 cache, we observe that analyzing the effect of reordering on all three cache levels is necessary to understand application performance. For example, for *wl* dataset, Sort yields 5.5% reduction in L3 MPKI over the baseline and yet causes a slowdown of 5.1%. In fact, the slowdown is caused by 15.3% and 19.6% increase in L1 and L2 MPKI, respectively, over the baseline.

All skew-aware techniques are generally effective in reducing L3 MPKI on all datasets but *lj*. On unstructured datasets (the left-most four datasets), all skew-aware techniques reduce L1 and L2 MPKI, with the highest reduction on the *sd* dataset. Meanwhile, on structured datasets (the right-most four datasets), Sort and HubSort, which do not preserve graph structure, significantly *increase* L1 and L2 MPKI (increase of 5.7 to 27.6 over original ordering). In contrast, HubCluster and DBG, which largely preserve existing structure, only marginally increase L1 and L2 MPKI (difference of -2.0 to 7.5) on structured datasets.

Results also show that small datasets are not good candidates for skew-aware reordering. For example, on *lj* and *wl* datasets, L3 MPKI is relatively small in the original ordering. As these datasets have relatively fewer number of vertices (5M and 18M vertices, respectively), hot vertices are largely cached in L3, even for the original ordering. Consequently, the opportunity to further exploit skew is small.

C. Performance Analysis of Push-dominated Applications

As seen in Fig. 6, all reordering techniques slowdown the PRD application on many datasets, the cause of which can be

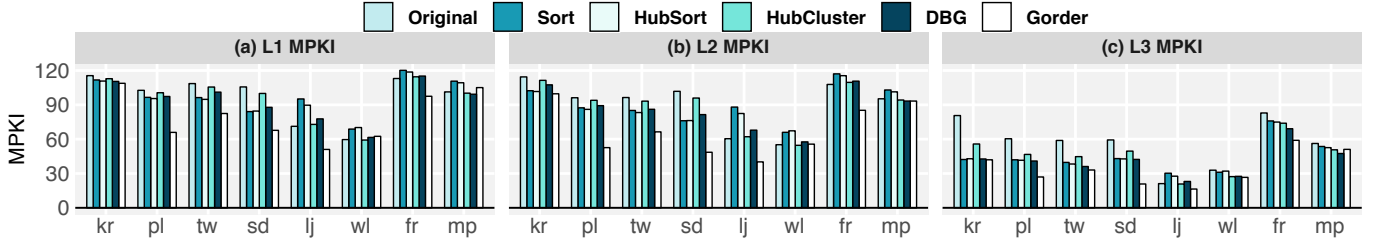


Fig. 8: Misses Per Kilo Instructions (MPKI) for the PR application across datasets. Lower is better.

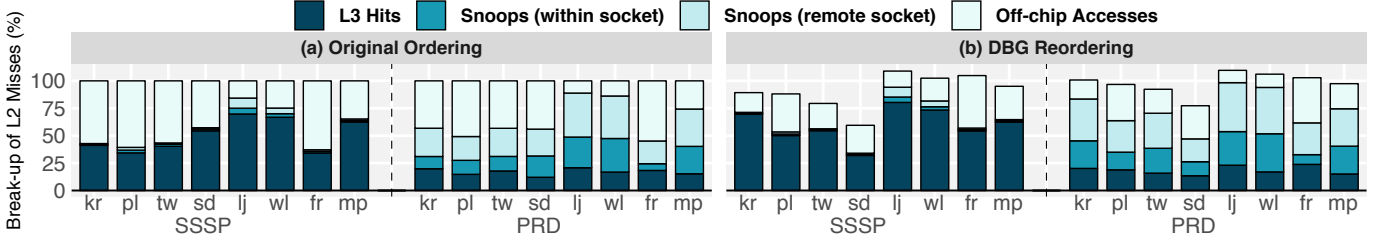


Fig. 9: Break-up of L2 misses for the push-dominated applications (SSSP and PRD) for datasets with original and DBG ordering, normalized to the L2 misses of the original ordering.

attributed to the push-based computation model employed by PRD. In push-based computations, when a vertex pushes an update through the out-edges, it generates scattered or irregular write accesses (as opposed to irregular read accesses in pull-based computations). As different threads may concurrently update the same vertex (true sharing) or update different vertices in the same cache block (false sharing), the push-based model leads to read-write or write-write sharing, hence generating on-chip coherence traffic.

Fig. 9 quantifies coherence traffic on both push-dominated applications – SSSP and PRD. The figure shows the break-up of L2 misses into four categories – L3 Hits (served by L3 without requiring any snoops to other cores), Snoop to other cores within the same socket, Snoops to another socket and off-chip accesses. For the first three categories, data is served by on-chip caches whereas for the last category, data is served from the main memory.

The two push-dominated applications have strikingly different fraction of coherence traffic while processing the datasets with the original ordering (middle two stacked bars in Fig. 9(a)). For SSSP, a relatively small fraction of L2 misses (14.5% for *lj* and below 9% for other datasets) required snoops whereas for PRD, a considerable fraction of L2 misses (from 26.9% for *fr* to 69.4% for *wl*) required snoops.

While processing a vertex using push-based computations, an application pushes updates (writes) to some or all destination vertices of the out-edges. In the case of PRD, it unconditionally pushes an update (i.e., a PageRank score) to all destination vertices while processing a vertex. In contrast, SSSP pushes an update to an out-edge only if it finds a shorter path through that edge. Thus, SSSP has much fewer number of irregular writes, and in turn, less coherence traffic,

in comparison to PRD.

Fig. 9(b) shows a similar break-up for SSSP and PRD on the datasets after DBG reordering. For PRD, DBG consistently reduces off-chip accesses (top stacked bar) across datasets, thus, a significantly higher fraction of requests are served by on-chip caches. However, most of these requests (37.8% to 77.0% of L2 misses) incur a snoop latency. For example, for DBG, while processing the *pl* dataset, 65.4% (vs 49.2% for the original ordering) of L2 misses are served by on-chip caches (bottom three stacked bars combined). However, most of these on-chip hits required snooping to other cores, incurring high access latency. Specifically, only 18.9% (vs 14.8% for the original ordering) of total L2 misses are served without requiring snooping. For most of the datasets, increase in L3 hits (i.e., no snooping) due to DBG is relatively small despite a significant reduction in off-chip accesses, which explains the marginal speed-up for DBG for the PRD application (Fig. 6).

For SSSP, most of the savings in off-chip accesses directly translate to L3 hits (i.e., no snooping) as the application does not exhibit high amount of coherence traffic even in the baseline. Thus, DBG is highly effective on SSSP, despite being dominated by push-based computations.

D. Performance Including Reordering Time

Fig. 10 shows end-to-end application speed-up for different reordering techniques after accounting for the reordering time. Due to space constraints, we show only four datasets (two largest unstructured and two largest structured datasets).

Gorder, while more effective at improving application speed-up (Fig. 6), when accounted for its reordering time, causes severe slowdowns (up to 96.5%) across datasets, corroborating prior work [6]. In contrast, all skew-aware

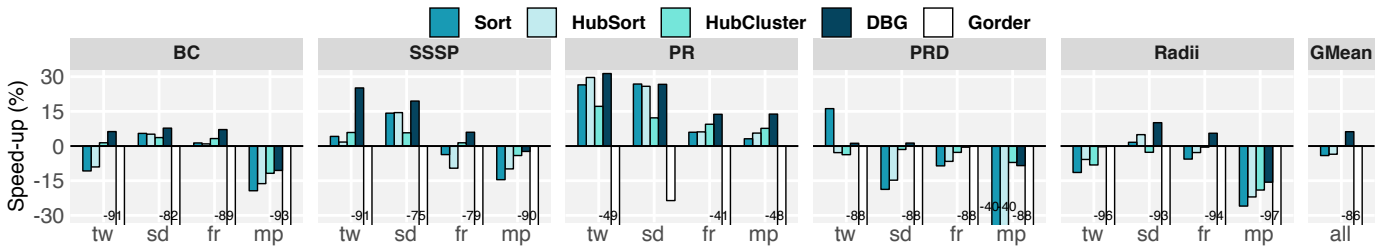


Fig. 10: Net speed-up for software reordering techniques over the baseline with original ordering of datasets. GMean shows geometric mean across speed-ups for all five applications on four datasets.

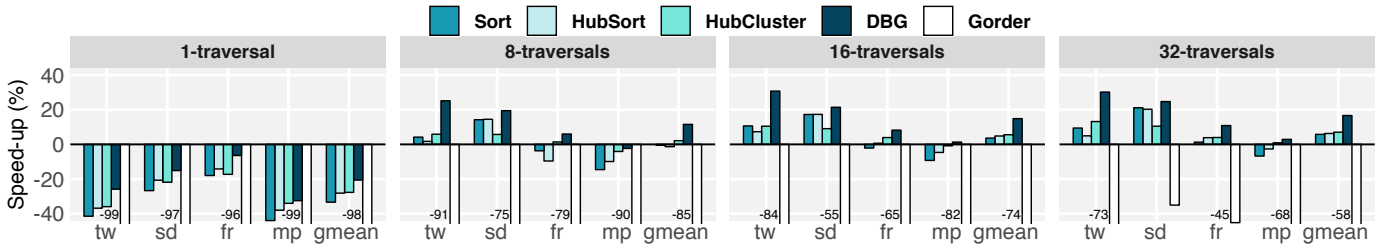


Fig. 11: Net speed-up for reordering techniques over the baseline with no reordering for SSSP with different number of traversals.

Dataset	Sort	HubSort	HubCluster	DBG	Gorder
<i>tw</i>	3.3	2.4	3.5	1.9	258.6
<i>sd</i>	3.7	3.0	5.0	2.4	112.2
<i>fr</i>	8.6	7.4	4.7	3.2	254.9
<i>mp</i>	18.2	10.3	7.5	4.4	1359.4

TABLE XII: Minimum number of iterations of PR application needed to amortize the reordering time of different reordering techniques.

techniques provide a net speed-up on at least some of the datapoints.

DBG outperforms all prior techniques on 17 out of 20 datapoints. DBG provides a net speed-up (up to 31.4%) on 14 out of 20 datapoints, even after accounting for its reordering time. On the remaining 6 datapoints, DBG reduces slowdown when compared to prior techniques, with maximum slowdown of 15.6% for the Radd application on the *mp* dataset and below 10% for others. In contrast, existing skew-aware techniques cause slowdown of up to 40.2% on half of the datapoints. Overall, DBG is the only technique that yields an average net speed-up (6.2%) by providing high performance while incurring low reordering overhead.

We next study how long it takes to amortize the reordering cost for an iterative application (PR) and a root-dependent traversal application (SSSP).

1) *Amortization point for PR*: The PR application has the largest runtime among all five applications for any given dataset, thus all skew-aware techniques are highly effective for the PR application and yield a net speed-up on all four datasets. Averaging across four datasets for the PR application, DBG outperforms all reordering techniques with 21.2% speed-up vs 15.1% for Sort, 16.3% for HubSort, 11.6% for HubCluster and -41.3% for Gorder.

Table XII lists the minimum number of iterations needed for the PR application to amortize the cost of different reordering techniques. For all four datasets, DBG is quickest in amortizing its reordering time, providing a net speed-up for all four datasets after just 2-5 iterations.

2) *Amortization point for SSSP*: We now evaluate net performance sensitivity to the number of successive graph traversals for different techniques for the SSSP application. The runtime for root-dependent applications depends on the number of traversals (or queries) performed from different roots. The exact number of traversals required depends on the specific use case. Thus, we perform a sensitivity analysis by varying the number of traversals from 1 to 32 in multiples of 8.

As shown in Fig. 11, with the increase in the number of traversals, performance for each technique also increases, as the reordering needs to be applied only once and its cost is amortized over multiple graph traversals. Thus, a single traversal is the worst-case scenario, with all techniques causing slowdown due to their inability to amortize the reordering cost. Of all the techniques, DBG causes the minimum slowdown (20.6% on average vs 27.7% for the next best) and is the quickest in amortizing the reordering cost, providing an average speed-up of 11.5% (vs 2.1% for the next best) with as few as 8 graph traversals.

VII. RELATED WORK

A significant amount of research has focused on designing high performance software frameworks for graph applications (e.g., [12], [13], [14], [15], [16], [17]). In this section, we highlight the most relevant works that focus on improving cache efficiency for graph applications.

Graph Slicing: Researchers have proposed graph slicing that slices the graph in LLC-size partitions and processes one

partition at a time to reduce irregular memory accesses [5], [15], [38]. While generally effective, slicing has two important limitations. First, it requires invasive framework changes to form the slices (which may include replicating vertices to avoid out-of-slice accesses) and manage them at runtime. Secondly, for a given cache size, the number of slices increases with the size of the graph, resulting in greater processing overheads in creating and maintaining partitions for larger graphs. In comparison, DBG only requires a preprocessing pass over the graph dataset to relabel vertex IDs and does not require any change in the graph algorithms.

Traversal Scheduling: A recent work proposed Bounded Depth-First Scheduling (BDFS) to exploit cache locality for graphs exhibiting community structure [37]. However, the software implementation of BDFS introduces significant book-keeping overheads, causing slowdowns despite improving cache efficiency. To avoid software overheads, the authors propose an accelerator that implements BDFS scheduling in hardware. In comparison, DBG is a software technique that can improve application performance without any additional hardware support.

Hardware Cache Management: Many researchers have proposed domain-agnostic hardware cache management schemes to reduce cache thrashing [39], [40], [41], [42], [43]. These hardware schemes improve cache utilization by keeping high reuse cache blocks in the LLC based on inter-block reuse and thus are orthogonal to DBG (and other skew-aware software techniques) that improve cache block utilization by focusing on intra-block reuse.

A recent work have proposed a domain-specialized cache management scheme for graph analytics [44], which requires hot vertices to be located in a contiguous memory region. DBG is inherently compatible with such a scheme. DBG can also facilitate a wider adoption of such a hardware scheme over more complex reordering techniques such as Gorder (assuming the cost of complex techniques can be tolerated). To achieve this, DBG can be applied after applying Gorder on an original dataset to further reorder vertices, which results in a vertex order that retains most of the Gorder ordering while also segregating hot vertices in a contiguous region. The combined reordering retains most of the performance of Gorder due to its structure preserving property (e.g., Gorder+DBG achieves 17.2% average application speed-up across 40 datapoints vs 18.6% for Gorder alone), while making dataset compatible with such a hardware scheme to gain further performance improvement.

VIII. FUTURE WORK

We believe that there are two potential future research directions for the wider domain of vertex reordering in the context of graph generation and dynamic graphs.

A. Integrating Reordering Techniques with Graph Generation

In this work, we assumed that the graph datasets are readily available and thus assumed that spatio-temporal locality in real-world datasets (specifically for the structured datasets)

exist without any overhead. In practice, such ordering may be a positive side effect of dataset generation algorithm (e.g., crawling webpages in certain order) or it may have been achieved by post-processing a dataset (e.g., graph datasets available from The Laboratory for Web Algorithmics have been ordered with the Layered Label Propagation technique [45]). Thus, there exist an opportunity to integrate skew-aware reordering techniques with the dataset generation process in order to avoid regenerating CSR-like structure post reordering, which dominates the reordering cost. At the very least, cost of reordering should be compared to the cost of a post-processing technique used over the raw dataset to understand the cost-benefit trade-offs of techniques from different domains.

B. Amortizing Reordering Costs on Dynamic Graphs

In this work, we assumed that graphs are static and thus, have evaluated a net speed-up conservatively assuming only one graph application (or query) over the reordered dataset (Fig. 10). In practice, a graph may evolve over time and a stream of graph updates (i.e., addition or removal of vertices or edges) are interleaved with graph-analytic queries. For such a deployment, graph reordering may provide an even greater benefit as the reordering cost can be amortized not only over multiple graph traversals of a single query, but also over multiple graph queries. Intuitively, addition or removal of some vertices or edges in a large graph would not lead to a drastic change in the degree distribution, and thus unlikely to change which vertices are classified hot in a short time window. Therefore, reordering techniques may need to be re-applied at large periodic intervals (i.e., after a series of updates has been made to a graph) to improve cache behavior, amortizing the cost of reordering over multiple graph queries performed in a given interval.

IX. CONCLUSION

In this work, we study skew-aware reordering techniques that seek to improve cache efficiency for graph analytics. Our work demonstrates the inherent tension between reducing the cache footprint of hot vertices and preserving original graph structure, which limits the effectiveness of existing skew-aware reordering techniques. To overcome the limitations of existing techniques, we propose Degree-Based Grouping (DBG), a novel lightweight reordering technique that employs coarse-grain reordering to preserve graph structure while reducing the cache footprint of hot vertices. DBG outperforms all existing skew-aware techniques, yielding an average speed-up of 16.8% vs 11.6% for the best-performing skew-aware technique.

ACKNOWLEDGMENT

We thank Artemiy Margaritov, Amna Shahab and the anonymous reviewers for their valuable feedback on an earlier draft of this work. This work was supported in part by a research grant from Oracle Labs.

REFERENCES

- [1] S. Beamer, K. Asanovic, and D. Patterson, "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, Oct. 2015.
- [2] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect Memory Prefetcher," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec. 2015.
- [3] S. Ainsworth and T. M. Jones, "Graph Prefetching Using Data Structure Knowledge," in *Proceedings of the International Conference on Supercomputing (ICS)*, Jun. 2016.
- [4] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup Graph Processing by Graph Ordering," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, Jun. 2016.
- [5] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making Caches Work for Graph Analytics," in *International Conference on Big Data (Big Data)*, Dec. 2017.
- [6] V. Balaji and B. Lucia, "When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, Sep. 2018.
- [7] A.-L. Barabási and R. Albert, "Emergence of Scaling in Random Networks," *Science*, vol. 286, no. 5439, 1999.
- [8] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On Power-law Relationships of the Internet Topology," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, Aug. 1999.
- [9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *Proceedings of the International Conference on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.
- [10] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, Jun. 2002.
- [11] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Statistical Properties of Community Structure in Large Social and Information Networks," in *Proceedings of the International Conference on World Wide Web (WWW)*, 2008.
- [12] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *Proceedings of the International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Feb. 2013.
- [13] D. Nguyen, A. Lenharth, and K. Pingali, "A Lightweight Infrastructure for Graph Analytics," in *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.
- [14] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP Benchmark Suite," *CoRR*, 2015, <http://arxiv.org/abs/1508.03619>.
- [15] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High Performance Graph Analytics Made Productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, Jul. 2015.
- [16] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A New Framework For Parallel Machine Learning," in *Proceedings of the International Conference on Uncertainty in Artificial Intelligence (UAI)*, Jul. 2010.
- [17] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Proceedings of the International Conference on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.
- [18] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis," in *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016.
- [19] K. Lakhota, S. Singapura, R. Kannan, and V. Prasanna, "ReCALL: Reordered Cache Aware Locality Based Graph Processing," in *International Conference on High Performance Computing (HiPC)*, Dec. 2017.
- [20] I. Stanton and G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, Aug. 2012.
- [21] G. Karypis and V. Kumar, "Multilevel k-way Partitioning Scheme for Irregular Graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, Jan. 1998.
- [22] U. Kang and C. Faloutsos, "Beyond 'Caveman Communities': Hubs and Spokes for Graph Compression and Mining," in *Proceedings of the International Conference on Data Mining (ICDM)*, Dec. 2011.
- [23] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," in *Proceedings of the National Conference*, Aug. 1969.
- [24] J. Banerjee, W. Kim, S. Kim, and J. F. Garza, "Clustering a DAG for CAD databases," *IEEE Transactions on Software Engineering*, vol. 14, no. 11, Nov. 1988.
- [25] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford InfoLab, Technical Report 1999-66, November 1999, <http://ilpubs.stanford.edu:8090/422/>.
- [26] C. Magnien, M. Latapy, and M. Habib, "Fast Computation of Empirically Tight Bounds for the Diameter of Massive Graphs," *Journal of Experimental Algorithmics (JEA)*, vol. 13, no. 10, Feb. 2009.
- [27] "Intel Xeon Processor E5-2630 v4," Intel Corporation, https://ark.intel.com/products/92981/Intel-Xeon-Processor-E5-2630-v4-25M-Cache-2_20-GHz.
- [28] "Web Data Commons - Hyperlink Graphs," Web Data Commons, <http://webdatacommons.org/hyperlinkgraph/>.
- [29] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?" in *Proceedings of the International Conference on World Wide Web (WWW)*, Apr. 2010.
- [30] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," Jun. 2014, <http://snap.stanford.edu/data>.
- [31] "Wikipedia, English network dataset - KONECT," Apr. 2017, <http://konect.uni-koblenz.de/networks/dbpedia-link>.
- [32] "Friendster network dataset - KONECT," The Koblenz Network Collection, Oct. 2016, <http://konect.uni-koblenz.de/networks/friendster>.
- [33] "Twitter (MPI) network dataset - KONECT," Apr. 2017, http://konect.uni-koblenz.de/networks/twitter_mpi.
- [34] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-Efficient Graph Processing on GPUs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015.
- [35] R. A. Rossi and N. K. Ahmed, "The Network Data Repository with Interactive Graph Analytics and Visualization," in *Proceedings of the International Conference on Artificial Intelligence*, 2015, <http://networkrepository.com/road-road-usa.php>.
- [36] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *Proceedings of the International Conference on Data Mining (SDM)*, 2004.
- [37] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Oct. 2018.
- [38] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Oct. 2016.
- [39] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, "SHiP: Signature-based Hit Predictor for High Performance Caching," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec. 2011.
- [40] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling Dead Block Prediction for Last-Level Caches," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec. 2010.
- [41] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Jun. 2016.
- [42] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron Learning for Reuse Prediction," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Oct. 2016.
- [43] P. Faldu and B. Grot, "Leeway: Addressing Variability in Dead-Block Prediction for Last-Level Caches," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2017.
- [44] P. Faldu, J. Diamond, and B. Grot, "POSTER: Domain-Specialized Cache Management for Graph Analytics," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2019.
- [45] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered Label Propagation: A Multiresolution Coordinate-free Ordering for Compressing Social Networks," in *Proceedings of the International Conference on World Wide Web (WWW)*, Mar. 2011.