Structure Preserved Graph Reordering for Fast Graph Processing Without the Pain

Baofu Huang, Zhidan Liu*, Kaishun Wu
College of Computer Science and Software Engineering, Shenzhen University, China
E-mails:huangbaofu2018@email.szu.edu.cn, {liuzhidan, wu}@szu.edu.cn

***Abstract*—By optimizing the data layout ahead-of-time, graph reordering can effectively improve the memory access locality in graph processing. The reordered graphs derived by sophisticated graph reordering approaches can greatly speedup the executions of most graph algorithms, while they incur huge computation overheads. Although lightweight approaches indeed reduce the reordering costs, they cannot achieve the best speedup performances. This is because they merely operate vertices of high-degrees and inadvertently destroy the community structures hidden in the graph. In this paper, we thus propose *Sorder* to balance the speedup performance and the reordering overhead. *Sorder* achieves better locality by preserving structural properties of graphs. Specifically, it mainly exploits neighborhood relations to renumber vertices and preferentially reorders vertices of high-degree ahead of the other vertices. We further enhance the design with the *hypernode* concept, which gathers neighboring vertices of low-degree to form a virtual vertex. Therefore, *Sorder* can consecutively rearrange more neighboring vertices, such that protecting the community structures. Extensive experiments with 5 representative graph algorithms and 7 real-world graphs demonstrate that *Sorder* can achieve comparable speedup performance as *Gorder*, the state-of-the-art approach, while significantly reducing the reordering overhead by the maximum $435\times$.**

I. Introduction

Graphs have been frequently used to model tremendous real-world data, which are generated from a variety of applications, *e.g.*, social networks, web pages, citation networks, and *etc.* Previously, people have deployed distributed systems to process such large-scale graphs for knowledge discovery [10]. In recent years, especially due to the rapid increase of memory capacity and core counts, many graphs can fit on one server's memory for the convenient processing. A number of optimized shared-memory computing frameworks, *e.g.*, Ligra [20], have been developed to process large graphs in a single machine, which can avoid the expensive cross-machine communications and the need to maintain a complex distributed system [4].

Due to inherently irregular memory access patterns of graph algorithms, however, it is still non-trivial to achieve efficient graph processing in shared-memory systems. Random memory accesses will inevitably cause low cache utilization, resulting in long CPU cache latency [2], [3], [22]. Prior works report that on average 70% of CPU time are stalled for the desired data to be accessed in caches in many graph algorithms [21].

As an effective technique to improve cache locality, *graph reordering* aims to optimize the data layout and the computation order by altering the indexing orders of vertices yet not changing their underlying connections. Graph reordering can speedup graph processing for (*almost*) all graph algorithms without modifying each algorithm itself, and thus has attracted considerable research efforts [2], [3], [7], [12], [13], [21], [22]. The sophisticated graph reordering approaches [2], [12], [21] greatly speedup graph processing, while incurring extremely huge computation overheads. The lengthy reordering time severely affects *end-to-end* graph processing performance, and thus limits their practicability. As a result, some lightweight approaches [3], [7], [22], [13] have been recently proposed. These techniques mainly exploit the power-law degree distribution of graphs, and preferentially reorder the *hub vertices*, which have relatively much more connections than others, with a high priority while retaining other vertices unchanged. The hub vertices are frequently accessed by numerous neighbors, leading to high temporal locality. Due to the simplicity, they indeed reduce the reordering time, while they cannot achieve the optimal orderings as *Gorder* [21], a state-of-the-art approach.

In addition to the property of power-law degree distribution, real-world graphs generally reveal the community structures, where vertices of a community have dense inner-connections [14]. During graph processing, a vertex will frequently access to other vertices of the same community, exhibiting the high spatial locality. These degree-based approaches [3], [7], [22], [13], however, omit or even severely destroy such structures, which potentially determine access patterns among vertices.

In this paper, we present a structure preserved graph reordering approach, named *Sorder*, which balances the speedup performance and reordering overhead to achieve better efficiency of graph processing. By analyzing the indexing patterns of graphs reordered by *Gorder*, we observe that hub vertices usually have smaller IDs and meanwhile neighboring vertices are contiguously renumbered. Such an indexing preserves the structural properties of both power-law degree distribution and community structures. Inspired by these observations, *Sorder* heuristically reorders a graph by primarily exploiting neighborhood relations, so as to achieve comparable speedup performance as *Gorder* while avoiding huge computation overhead. Specifically, for a given seed vertex, *Sorder* classifies its neighbors into high-degree and low-degree, and then assigns consecutive IDs to each group separately. In particular, high-degree neighbors are always renumbered ahead of the low-degree ones. By selecting a neighbor as the seed of next round,

* Corresponding author.

Sorder repeats the operations until all vertices are renumbered. We enhance *Sorder* with the concept of *hypernode* that clusters low-degree neighbors of the seed vertex as one virtual vertex. With such an optimization, *Sorder* can not only preserve more complete community structures, but also consecutively renumber more neighboring vertices. Therefore, the temporal-spatial cache locality of a graph is largely improved.

The contributions of our work are summarized as follows:

- We have comprehensively analyzed the indexing patterns of *Gorder*, the state-of-the-art approach, and derived two insights for devising effective reordering approaches.
- We present *Sorder* that can well preserve the structural properties of graphs for better graph reordering efficiency.
- We conduct extensive experiments with 5 typical graph algorithms on 7 real-world graphs. Experimental results demonstrate that *Sorder* can achieve comparable speedup performance as *Gorder* with maximum speedup as $2.56\times$, while significantly reducing the pre-processing overhead, *e.g.*, by $435\times$ at most.

The rest of the paper is organized as follows. We present the background and motivation of graph reordering in Section II. The design of *Sorder* is detailed and evaluated in Section III and Section IV, respectively. We review the related works in Section V. Finally, Section VI concludes this paper.

II. Background and Motivation

In this section, we will introduce the background of graph processing, and then discuss the necessity of graph reordering. We motivate our design by analyzing the indexing patterns of graphs reordered by *Gorder*.

A. Preliminary

Graph modeling. The real-world data can be modelled as a directed graph $\mathtt{G} = (\mathtt{V}, \mathtt{E})$, where $\mathtt{V}$ represents a set of vertices (*e.g.*, users in a social network) and $\mathtt{E}$ is a set of edges (*e.g.*, indicating relationships among users). Given a vertex $v \in \mathtt{V}$, we denote its in-neighbors and out-neighbors as $\mathcal{N}_{in}(v)$ and $\mathcal{N}_{out}(v)$, respectively. In addition, vertex v is associated with an attribute d_v. Most of graph algorithms process and analyze the attributes of vertices for knowledge discovery [10].

Graph representation. In shared-memory frameworks, the *compressed sparse row* (CSR) format is widely used to represent a graph in a storage-efficient manner [2], [3], [4], [7], [12], [21], [22]. CSR utilizes two arrays, *i.e.*, a *coordinate array* (CA) and an *offset array* (OA), to efficiently encode a graph's edges (sorted by the edge source/destination) [4]. Specifically, CA contiguously stores the neighbors of each vertex, and OA stores the offset of each vertex's first neighbor in the CA. In order to access the neighbors of the i-th vertex, a program accesses the i-th entry of OA to find this vertex's first neighbor in the CA. In addition, the number of neighbors for the i-th vertex is the difference between the entries $(i+1)$ and i in the OA. To represent a directed graph G, two CSRs can be used to encode vertices' out-neighbors and in-neighbors, respectively.

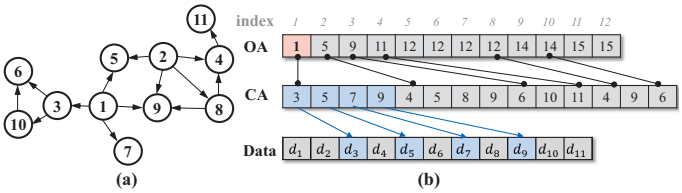

Fig. 1. (a) A sample directed graph G; (b) The CSR representation of vertices' out-neighbors for graph G.

TABLE I
Statistics on the skewness of real-world graphs.

Graph	In-neighbors		Out-neighbors	
	Hubs	*Involved edges*	*Hubs*	*Involved edges*
flickr	4.0%	84.8%	4.1%	85.1%
orkut	19.2%	74.5%	21.1%	72.0%
livej	6.1%	62.7%	6.3%	62.8%
it	4.8%	85.4%	14.2%	64.7%
pld	4.5%	79.0%	4.1%	82.6%
twitter	5.7%	89.5%	4.3%	84.3%
sd	4.8%	85.6%	5.0%	83.7%

Figure 1(b) illustrates the CSR representation for out-neighbors of the sample graph shown in Figure 1(a).

Graph properties. The real-world graphs commonly have two distinguish structural properties described as follows.

- *Power-law degree distribution.* The majority of vertices have relatively few neighbors while a few vertices have many neighbors. The degree distribution is skewed and nearly follows the *power-law* distribution [10]. The vertices with more neighbors are usually called as *hub vertices*, and are frequently accessed by other vertices during the graph processing. Table I presents the percentages of hub vertices and their involved edges of 7 real-world graphs (See more details about these graphs in Section IV-A). Here we consider a vertex with degree ≥ 50 as a hub vertex. Table I shows that for all graphs, although hub vertices only account for $4\% \sim 21\%$ of all vertices, they connect $62\% \sim 89\%$ edges for both *in* and *out* direction.
- *Community structure.* The vertices of a real-world graph can be easily grouped into clusters. Specifically, vertices of the same cluster are densely connected, while vertices of different clusters may be sparsely connected [7]. For example in a social network, users sharing the common interests will usually form different communities [14].

Algorithm 1: *Typical Graph Processing Kernel*

```
1 for v ∈ frontier do
2     for u ∈ N_out(v) do
3         Update(d_v, d_u, ···);
```

Graph processing. Algorithm 1 sketches the typical graph processing kernel. Generally, graph algorithms process an input graph by iteratively visiting the vertices and their neighbors until some convergence criterion is achieved. During each iteration, all vertices or only a subset of them will be visited, and these active vertices are called *frontier*. The attribute data of their neighbors are accessed to update some information, *e.g.*, *PageRank* value of a target vertex. The vertices of next iteration's frontier are identified with the application-specific logic. When implementing a specific graph algorithm, a vertex

can either *push* its attribute to update its out-neighbors, or *pull* its in-neighbors' data to update its own value. The efficiency of push- or pull-based implementations varies by different algorithms [6], and a wise switch in each iteration may achieve the better performance for some graph algorithms [20].

B. Why Graph Reordering

As illustrated in **Algorithm 1**, graph algorithms sequentially access to a given vertex's edges (which are stored in the CSR format), but will randomly access its neighbors' data, resulting in irregular memory accesses. As an example in Figure 1(b), when the program processes vertex v_1, it needs to access the attribute data of v_1's neighbors, *i.e.*, $\{d_3, d_5, d_7, d_9\}$, which are randomly distributed in the data array. Since the frontier of real-world graphs is pretty larger than cache size of current machines [22], irregular vertex data accesses will cause high cache miss ratio. According to a recent study, the representative graph algorithms may even waste $55\% \sim 90\%$ of their CPU time stalled on the memory accesses [21].

To improve cache locality, graph reordering is proposed to rearrange all vertices in the optimal ordering that is aligned with the CPU accesses of a graph, so as to reduce CPU cache misses. Since graph reordering merely renumbers the vertices, without changing the graph itself, implementations of graph algorithms, or the data structures used, it has attracted many research efforts recently [2], [3], [7], [12], [13], [21], [22]. As one of the state-of-the-art graph reordering approach, *Gorder* can achieve the best > 2 speedups in testing typical graph algorithms over large real-world graphs [21].

To understand the benefits of graph reordering, we assume that a program executes **Algorithm 1** over the sample graph in Figure 1(a) by visiting all vertices once. Figure 2(a) illustrates the cache behaviors, where we assume the machine has two cache blocks (CB), each of which can contain 2 vertex data, and the least recently used (LRU) caching scheme is adopted here. From Figure 2(a), we see that **Algorithm 1** suffers from high cache misses, with only 3 cache hits in one iteration. As a comparison, we run this testing over the same graph yet in an alternative ordering as shown in Figure 4(d), the cache hits increase to 9 with speedup by 300%, as shown in Figure 2(b).

Therefore, it is necessary to reorder a graph before inputting it to the graph algorithms for better computation efficiency. We formally define graph reordering problem as follows.

Problem statement: *Given a graph* $\mathtt{G} = (\mathtt{V}, \mathtt{E})$ *and letting* $\Phi(\cdot)$ *be a permutation function that assigns a vertex a unique ID in* $[1, \cdots, n]$ *where* $n = |\mathtt{V}|$*, graph reordering aims to find the optimal permutation of vertices* $\Phi : \mathtt{V} \rightarrow \mathtt{N}$*, which minimizes the total cache misses produced by Algorithm 1.*

C. Motivation

Despite the good speedup performance, *Gorder* incurs extremely huge computation overhead to find the optimal vertex ordering. For example, it reduces the execution time of *PageRank* over graph *twitter*, which consists of 61.6 million vertices and 1.5 billion edges, from 218 seconds to 144 seconds,

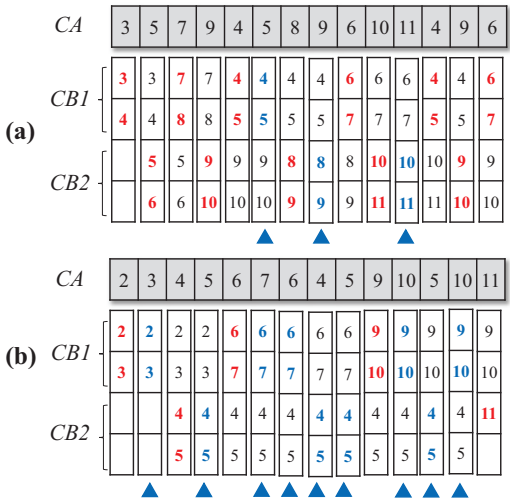

Fig. 2. (a) and (b) show cache behaviors for the executions of **Algorithm 1** on the original vertex ordering (as shown in Figure 1(a)) and an alternative vertex ordering (as shown in Figure 4(d)), respectively. Red numbers represent loaded data from memory (*i.e.*, cache misses), and blue numbers represent the cache hit CBs, which are further indicated by the blue triangles.

however, at the expense of spending about 2.8 hours to reorder the graph. *Gorder* assumes that the expensive pre-processing overhead could be amortized over multiple executions on the reordered graph, however, in practice many graph applications require to conduct timely analysis on the evolving graphs (*a.k.a* temporal graph mining) [19], *e.g.*, executing *PageRank* on dynamically changing social networks.

Thus, we turn to design a graph reordering approach that can not only achieve comparable speedup performance as *Gorder*, but also incurs slight reordering cost. To this end, we carefully analyze the design of *Gorder* and its reordering patterns, in hope of discovering some helpful insights.

Understanding *Gorder*. By analyzing **Algorithm 1**, *Gorder* observes that graph processing kernel mainly involves neighborhood relationship and sibling relationship among vertices, which together determine the data access patterns. *Gorder* thus defines a score function to measure the closeness of any two vertices, *e.g.*, u and v, in terms of locality as follows:

$$\mathcal{S}(u, v) = \mathcal{S}_s(u, v) + \mathcal{S}_n(u, v), \tag{1}$$

where $\mathcal{S}_s(u, v)$ indicates the number of common in-neighbors of u and v (*i.e.*, they are sibling), and $\mathcal{S}_n(u, v)$ indicates the number of times that u and v are direct neighbors [21]. With this score function, *Gorder* aims to find the optimal ordering $\Phi(\cdot)$ by maximizing the accumulated locality score $\mathcal{F}(\cdot)$ over a sliding window of size ω. Specifically, $\mathcal{F}(\cdot)$ is defined as

$$\mathcal{F}(\Phi) = \sum_{0 < \Phi(v) - \Phi(u) \leq \omega} \mathcal{S}(u, v). \tag{2}$$

Gorder has proved that maximizing $\mathcal{F}(\cdot)$ is NP-hard [21], and proposed a greedy algorithm to iteratively search the best solution. The time complexity of *Gorder* is $O(\omega \cdot d_{max} \cdot n^2)$, where d_{max} is the maximum in-degree of the graph.

Although *Gorder* has high computation overheads, its practical performance is close to the optimal in experiments [21].

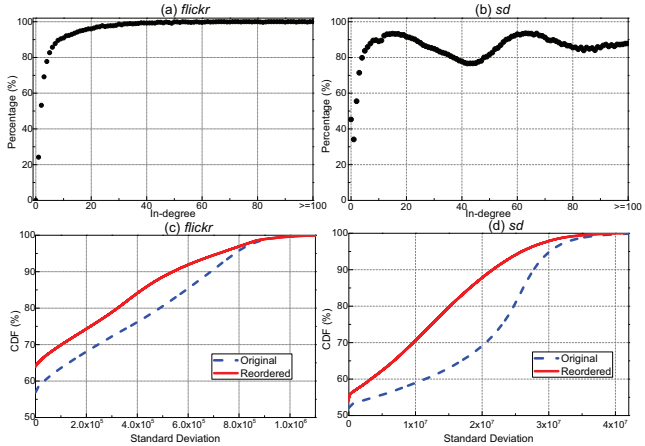

Fig. 3. With respect to in-degree, (a) and (b) present the percentage of vertices with smaller IDs in reordered graph than original graph for graph $flickr$ and sd, respectively. (c) and (d) present the CDF of standard deviations of neighboring vertices' IDs for graph $flickr$ and sd, respectively.

Therefore, we run *Gorder* on two real graphs (*i.e.*, $flickr$ and sd), and compare the vertex orderings of original graphs and reordered graphs to expose possible hints for a better design.

At first, we compare each vertex's IDs in both graphs with respect to the in-degree. We classify all vertices into groups according to their in-degrees, and for each group we calculate the percentage of vertices that has smaller IDs in reordered graph than original graph. Figure 3(a)(b) show the statistics for graph $flickr$ and sd, respectively. In general, a vertex owning higher in-degree tends to be assigned with a smaller ID (when compared to its original ID) by *Gorder* with a higher probability. When the in-degree of a vertex is higher than 50, its ID is reduced in the new ordering with probability of near 100% and $> 80\%$ for $flickr$ and sd, respectively. This can be explained using Equation (1), *Gorder* prefers to reorder vertices of high in-degree in advance, since they can derive much higher scores. Once these vertices are reordered in advance, they will be processed earlier and their cached attributes benefit data accesses of the numerous neighbors.

Next, we further study the indexing patterns among vertices of the same community by comparing the ID variances of each vertex's neighbors for both original and reordered graphs. In theory, if a vertex's neighbors have contiguous IDs, their ID variance should be small. As shown in Figure 3(c)(d), the standard deviations of reordered graphs for both $flickr$ and sd are greatly reduced when compared to the original graphs. These results imply that *Gorder* will potentially aggregate the neighboring vertices by assign them adjacent IDs. Thus, *Gorder* indirectly preserves the community structures as well.

Key insights. Based on above analysis, we derive two useful guidelines to effectively reorder a real-world graph.

(1) *Reordering hub vertices first.* Since data of hub vertices will be frequently accessed by numerous vertices, they should be reordered earlier so that their data can be cached for reuse by the other vertices. In essence, reordering hub vertices early can improve the temporal locality of memory accesses.

(2) *Reordering neighboring vertices together.* The neighboring vertices possibly belong to the same community and thus they may be accessed together, exhibiting high spatial locality of memory accesses. Therefore, the neighboring vertices should be assigned with consecutive IDs, so as to make their data reside nearby in the memory.

III. Design of *Sorder*

Inspired by above insights, we propose *Sorder* to reorder a graph with balanced speedup performance and pre-processing overhead. Different from previous works [3], [4], [7], [22] that merely reorder a few hub vertices, *Sorder* not only guarantees the reordering priority of hub vertices, but also preserves the community structures among vertices.

Basic design. *Sorder* sequentially processes each vertex and its out-neighbors by exploiting the CSR representation. Specifically, for each vertex $v \in \mathtt{V}$, it assigns a new ID to v, and then assigns consecutive IDs to v's out-neighbors. To guarantee that the vertices of high degree can be loaded into the cache early for facilitating later data accesses, *Sorder* preferentially assigns consecutive IDs to v's hub-vertex neighbors ahead of v's non-hub neighbors. *Sorder* identifies a vertex with *in-degree* higher than the threshold λ as the *hub vertex*. During the reordering, *Sorder* also keeps an eye on the un-renumbered neighbors of v, and selects the last un-renumbered neighbor as the seed vertex to trigger next round of numbering. As a result, *Sorder* spreads reordering operations along with the neighborhood relations most of the time, such that the community structures would be reserved. *Sorder* repeats above operations until all vertices are renumbered. During the reordering, both hub neighbors and non-hub neighbors are sequentially renumbered according to their orders in the CSR representation.

Enhancement. The basic design will make hub vertices and non-hub vertices alternatively appear in the new ordering. Such a permutation may only preserve partial community structures, and thus cannot achieve the best performance. Detecting and reserving the intact community structures of a graph, however, is non-trivial and will incur huge pre-processing overheads [2]. To make more neighboring vertices reside adjacent in the new permutation and retain the most community relations among vertices at little expense, we thus propose a concept named as *hypernode*, which aggregates adjacent non-hub vertices as one virtual vertex, to enhance the basic design.

A hypernode begins from a seed vertex and expands itself by including neighboring non-hub vertices. Specifically, for seed vertex v, we retrieve its κ-hop out-neighbors from the CSR representation and selectively collect its non-hub neighbors to form a hypernode $\mathcal{H}_v$. Initialized as an empty set, $\mathcal{H}_v$ adds elements in an iterative manner:

1) Vertex v's out-neighbors, which are non-hub vertices and not renumbered yet, are included into $\mathcal{H}_v$;
2) For each element $u \in \mathcal{H}_v$, we add its out-neighbors, which also should be non-hub vertices and not renumbered yet, to into $\mathcal{H}_v$.

We repeat step 2) until the κ-th hop neighbors of v are reached.

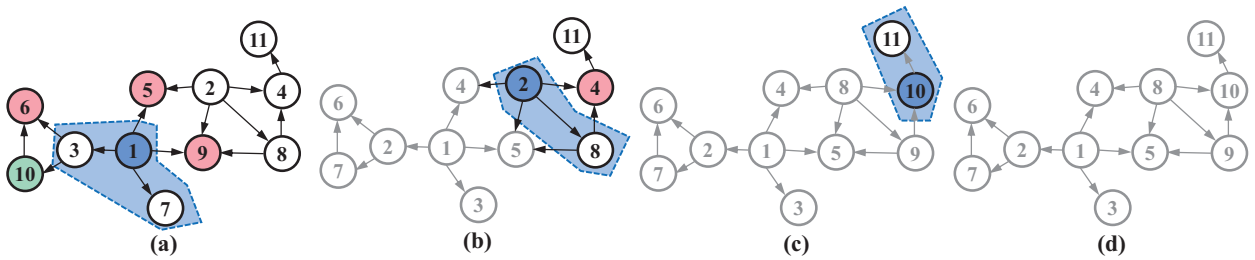

Fig. 4. Illustrations of applying *Sorder* on the sample graph in Figure 1(a), with step by step in (a-c). The blue vertex is the seed vertex of each round, the vertices covered by blue area form a hypernode (here we set $\kappa = 1$). Besides, red and green vertices are *hub* and *non-hub* neighbors of the hypernode, respectively, where we set $\lambda = 1$ for this example. The already renumbered vertices are marked by gray. Subfigure (d) shows the final reordered graph.

Algorithm 2: Structure-Preserved Graph Reordering

Input: Graph $\mathtt{G} = (\mathtt{V}, \mathtt{E})$, $n = |\mathtt{V}|$
Output: A reordered graph $\mathtt{G}$ with permutation function $\Phi(\cdot)$

1 *Assigned*[·] = {*false*, ···, *false*}, *move_id* = 1, *seed* = -1;
2 **for** $v \in \mathtt{V}$ **do**
3 **if** !$Assigned(v)$ **then**
4 *seed* = v;
5 **while** *seed != -1* **do**
6 $\mathcal{H}_v$ = *Fusion*(v, $\mathtt{V}$);
7 **for** $i \in \mathcal{H}_v$ & !$Assigned(i)$ **do**
8 $\Phi(i)$ = *move_id*++;
9 *Assigned(i)* = *true*;
10 *seed* = -1;
11 *NonHubs* = {};
12 *Retrieve out-neighbors of* $\mathcal{H}_v$ *to form set* $\mathbb{N}_{\mathcal{H}_v}$;
13 **for** $u \in \mathbb{N}_{\mathcal{H}_v}$ **do**
14 **if** !$Assigned(u)$ **then**
15 *seed* = u;
16 **if** $|\mathcal{N}_{in}(u)| \geq \lambda$ **then**
17 $\Phi(u)$ = *move_id*++;
18 *Assigned(u)* = *true*;
19 **else**
20 *NonHubs* = *NonHubs* $\cup \{u\}$;
21 **for** $i \in$ *NonHubs* **do**
22 $\Phi(i)$ = *move_id*++;
23 *Assigned(i)* = *true*;

Therefore, the enhanced design is as follows. For the generated hypernode $\mathcal{H}_v$, we implicitly classify its out-neighbors, which are actually the immediate out-neighbors of any vertex belonging to $\mathcal{H}_v$, into two groups as hub vertices and non-hub vertices, and then assign consecutive IDs to the vertices of each group in turn. Again, the hub vertices are renumbered ahead of these non-hubs. Among $\mathcal{H}_v$'s neighbors, the last one is chosen as the seed vertex to produce another hypernode to continue the graph reordering.

The holistic design. Algorithm 2 sketches the pseudocode of *Sorder*. At the beginning, all vertices are not assigned with new IDs, and we use $seed$ to store the seed vertex. For each seed vertex $v \in \mathtt{V}$, *Sorder* will generate a hypernode by invoking the function $Fusion(v, \mathtt{V})$ (line 6), and then assigns consecutive IDs to the vertices belonging to the hypernode $\mathcal{H}_v$ (line 8-9). Next, *Sorder* gathers the out-neighbors of hypernode

Algorithm 3: Form a Hypernode

1 **Function** `Fusion`(v, $\mathtt{V}$)**:**
2 $\mathcal{H}_v = \{v\}$, *list* = $\{v\}$, *hop* = κ;
3 **while** $hop > 0$ **do**
4 *temp* = {};
5 **for** $i \in list$ **do**
6 **for** $j \in \mathcal{N}_{out}(i)$ **do**
7 **if** *!Assigned(j)* & $|\mathcal{N}_{in}(j)| < \lambda$ **then**
8 *temp* = *temp* $\cup \{j\}$;
9 $\mathcal{H}_v = \mathcal{H}_v \cup$ *temp*;
10 *list.**swap**(temp)*;
11 $hop = hop - 1$;
12 **return** $\mathcal{H}_v$;

$\mathcal{H}_v$ into set $\mathbb{N}_{\mathcal{H}_v}$ (line 12), and assigns them with new IDs (line 13-23). For a vertex $u \in \mathbb{N}_{\mathcal{H}_v}$, if it is not assigned yet and is a hub-vertex (*i.e.*, its in-degree is greater than λ), it will be renumbered (line 16-18). The vertices with small in-degree ($< \lambda$) are stored in the array *NonHubs*. After all hub vertices in $\mathbb{N}_{\mathcal{H}_v}$ have been renumbered, the non-hub vertices in *NonHubs* are consecutively indexed (line 21-23). During the reordering procedure, $seed$ records the next seed vertex to continue next round of ID assignments. If all the out-neighbors of hypernode $\mathcal{H}_v$ are already renumbered, *Sorder* will continue the reordering from next vertex of v in the set $\mathtt{V}$.

Algorithm 3 presents the pseudocode of $Fusion(\cdot)$. For a given seed vertex v, *Sorder* iteratively add the vertices, which are within κ-hop of v and not renumbered yet, to set $\mathcal{H}_v$.

Figure 4 illustrates the procedure of applying *Sorder* on the sample graph in Figure 1(a). In Figure 4(a), *Sorder* starts from vertex v_1, and forms hypernode $\mathcal{H}_1 = \{v_1, v_3, v_7\}$ by including the non-hub vertices within 1 hop of v_1. The vertices in $\mathcal{H}_1$ are assigned with new IDs (*i.e.*, $1, 2, 3$). Then, out-neighbors of $\mathcal{H}_1$ are retrieved to form the set $\mathbb{N}_{\mathcal{H}_1} = \{v_5, v_9, v_6, v_{10}\}$. In particular, the three hub vertices $\{v_5, v_9, v_6\}$ are renumbered preferentially with new IDs (*i.e.*, $4, 5, 6$) before assigning 7 as the new ID to non-hub vertex v_{10}. Meanwhile, $seed$ is set as v_{10} during above reordering. However, all out-neighbors of v_{10} have been already renumbered. Thus, *Sorder* takes the formal v_2 as the seed vertex for next round, as shown in Figure 4(b). Similarly, *Sorder* forms hypernode $\mathcal{H}_2 = \{v_2, v_8\}$ and assigns them with new ID $\{8, 9\}$ in turn. Then, *Sorder* derives $\mathbb{N}_{\mathcal{H}_2} = \{4\}$ from $\mathcal{H}_2$, and assigns the un-renumbered neighbor

TABLE II
SUMMARY OF THE USED REAL-WORLD GRAPHS (*M: million*)

Graph	Description	#vertices	#edges	$\bar{d}$	Disk size
$flickr$	Social network	$2.3\,M$	$33.1\,M$	14	$0.4\,GB$
$livej$	Social network	$4.8\,M$	$68.5\,M$	14	$1.1\,GB$
$orkut$	Social network	$3.0\,M$	$106.3\,M$	35	$1.5\,GB$
pld	Hyperlinks	$42.9\,M$	$623.1\,M$	15	$10.9\,GB$
it	Hyperlinks	$41.3\,M$	$1135.7\,M$	28	$19.0\,GB$
$twitter$	Social network	$61.6\,M$	$1468.4\,M$	24	$25.0\,GB$
sd	Hyperlinks	$94.9\,M$	$1937.5\,M$	20	$34.4\,GB$

with new ID as 10. During this round, the formal vertex v_4 is selected as the seed to enable the last round, as shown in Figure 4(c). The last vertex v_{11} is assigned with 11 as the new ID. Figure 4(d) presents the final reordered graph by *Sorder*.

IV. PERFORMANCE EVALUATION

In this section, we conduct extensive experiments to evaluate *Sorder* using typical graph algorithms and real-world graphs.

A. Experimental Setup

For performance evaluation, we compare *Sorder* with other four graph reordering approaches on seven large real-world graphs using five representative graph algorithms. We conduct all experiments with a powerful machine, which is equipped with a dual-socket Intel(R) Xeon(R) E5-2630 v4 10-core processors @2.20GHz and 192GB memory, running the Ubuntu 20.04. In addition, the L1, L2, and L3 cache size of the machine are 640KB, 5MB, and 50MB, respectively.

Graph algorithms. We select five typical graph algorithms to test various graph reordering approaches. Specifically, we adopt their implementations from Ligra [20] benchmark suites. All implementations are complied using g++ -9.3 with the highest optimization -O3 option. We briefly introduce these graph algorithms as follows.

- *Radii Estimation* (*Radii*) approximates the diameter of a graph by performing parallel breadth-first-search (BFS) traversals from a set of randomly selected sources [16].
- *Betweenness Centrality* (*BC*) searches the most central vertices in a graph by exploiting a BFS kernel to count the number of shortest paths passing through each vertex from a given source vertex [9].
- *Single Source Shortest Path* (*SSSP*) calculates the shortest paths for all vertices in a weighted graph from the given source using the Bellman Ford algorithm [20].
- *PageRank* (*PR*) computes the ranks of vertices based on both quantity and quality of their incoming edges in an iterative manner [18].
- *PageRank-delta* (*PR-delta*), proposed as a faster variant of *PageRank*, only lets a subset of vertices, whose ranks are sufficiently changed, to be active in an iteration [15].

For each graph algorithm, we adopt the default settings in Ligra implementation for experiments.

Input graphs. We present the key statistics of input graphs in Table II. These graph data are collected from real-world applications, including social networks and hyperlinks among web pages. All graphs contain millions of vertices and edges. In particular, $flickr$ is the smallest graph and sd is the largest one. In addition, we have three billion-edge graphs, *i.e.*, it, $twitter$, and sd. The average degrees (*i.e.*, the column of $\bar{d}$) range from 14 to 35, and their disk sizes are in the range from 0.4GB to 34.4GB. We utilize the original vertex ordering of each graph for the baseline executions of all graph algorithms.

Compared approaches. We compare *Sorder* with the following four graph reordering approaches.

1) *Sort* derives the permutation by simply sorting vertices in descending order of in-degrees. It has also been selected as a baseline approach by previous works [2], [3], [7], [21].

2) *DBG* is a coarse-grain reordering approach that partitions vertices into a number of groups based on their degrees while retaining the relative order of vertices within each group [7].

3) *Norder* is a recent proposal that has also exploited the neighborhood relations [13]. It firstly arranges all vertices in descending order of their in-degrees, and then performs a BFS search that also assign a vertex ID in the traversed order.

4) *Gorder* [21] renumbers vertices according to their scores, which are calculated using Equation (1). As the sophisticated approach, it can achieve the best speedup performance, while it introduces significant pre-processing overhead.

Evaluation methodology. For fair comparisons, we directly adopted the open-sourced codes of the compared approaches, all of which are written in C++. We implemented our approach *Sorder* in C++ as well. All the codes are complied using g++ -9.3 with the highest optimization option. For compared approaches, we test them and set their parameters to achieve the best performances. For *Sorder*, we set $\lambda = 50$ and $\kappa = 2$ by default. The efficiency of each graph reordering approach is measured by three metrics as *reordering time*, *cache miss ratio*, and *execution speedup* of graph algorithms. In particular, given a graph algorithm the speedup is calculated as the ratio between execution time over the original graph and execution time over the graph reordered by a graph reordering approach. We evaluate each approach on every combination of graph algorithms and input graphs 6 times, and report the average of the results from the last 5 times. Similar as previous works [3], [4], [7], [8], we let the first execution to warm up cache.

B. Results

In this subsection, we will discuss the comparison results of the five approaches, and then study the impacts of parameter κ on *Sorder*'s performances.

Comparison on execution time speedup. Figure 5 presents the speedup comparisons of all approaches on 35 combinations of graph algorithms and input graphs. For a clear comparison, for each graph algorithm we calculate geometric mean speedups of the five approaches (*i.e.*, *GMean* in the last column of each subfigure in Figure 5). Since *Gorder* reorders vertices by comprehensively analyzing their connections to effectively improve cache locality, it can thus achieve the best speedup performances. Specifically, *Gorder* has the highest speedups over 21 out of 35 combinations, and our approach *Sorder* takes the second place with 9 times of the best speedup. In fact, we

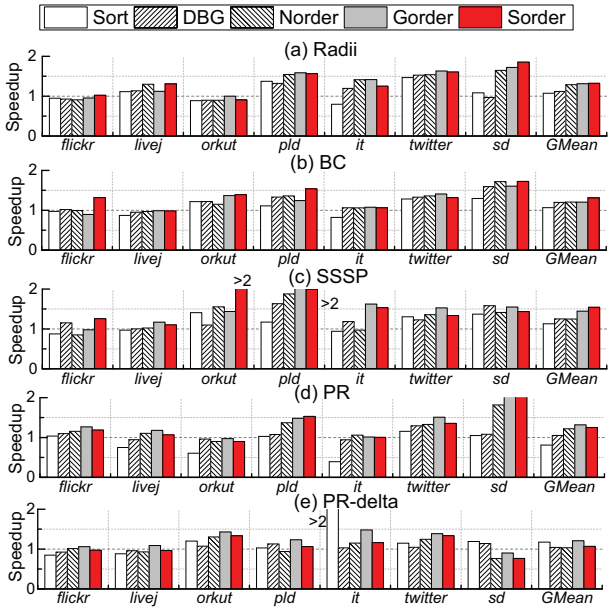

Fig. 5. The speedup comparisons of different graph reordering approaches on various graph algorithms and input graphs. For each graph algorithm, the *GMean* results of all reordering approaches are provided in the last column.

observe that *Sorder* have quite close speedups as *Gorder* in the majority cases in Figure 5. In addition, we find that *Sort*, *DBG*, and *Norder* can only perform well on few scenarios, as they win the best with 2, 2, and 1 times, respectively.

Regarding on comprehensive speedup performances, we see that *Sorder* slightly outperforms *Gorder* on graph algorithms of *Radii*, *BC*, and *SSSP* by achieving higher *GMean* results. In contrary, *Gorder* performs the best on the graph algorithms of *PR* and *PR-delta*. Figure 5 demonstrates that reordered graphs can generally yield better graph processing performances (*i.e.*, with speedup > 1), while most of the reordering approaches relatively perform not good on the *PR-delta* algorithm. This is possibly because the subset of active vertices are continuously changing when *PR-delta* proceeds, and at last iterations only the hub vertices have not been converged. That may be the reason why *Sort* can work well for *PR-delta*.

Different combinations of algorithms and graphs will result in diverse memory access patterns, and the speedups of *Sorder* thus vary from case to case. In general, *Sorder* works well for most of algorithms and graphs, where 30 out of 35 reordered graphs have obtained positive speedups (*i.e.*, > 1). Among the five failures, three cases come from the executions of *PR-delta*, where the vertices of frontier dynamically change along iterations. It is difficult to predict the subset of vertices that will keep active in each iteration. From Figure 5, we find that *Sorder* can achieve the maximum speedup as $2.56\times$, and its *GMean* speedups of the five algorithms are $1.32\times$, $1.31\times$, $1.54\times$, $1.25\times$, and $1.07\times$, respectively.

Lastly, we calculate the *GMean* over all the 35 combinations for the five approaches. Specifically, the *GMean* speedups of *Sort*, *DBG*, *Norder*, *Gorder*, and *Sorder* are 104.2%, 112.8%, 119.5%, 129.4%, and 129.0%, respectively. We see that *Sorder* can achieve comparable speedup performance as *Gorder*.

TABLE III
CACHE STATISTICS BY ALGORITHM *PR* OVER GRAPH $flickr$.

Approach	L1-ref	L1-mr	L3-ref	L3-r	Cache-mr
Original	2.74E+10	34.38%	6.37E+09	23.23%	1.83%
Sort	2.75E+10	34.64%	6.57E+09	23.92%	1.87%
DBG	2.74E+10	30.95%	5.53E+09	20.13%	1.61%
Norder	2.75E+10	29.70%	5.23E+09	19.06%	1.41%
Gorder	2.75E+10	24.99%	3.77E+09	13.72%	**1.11%**
Sorder	2.74E+10	27.91%	4.89E+09	17.85%	**1.27%**

TABLE IV
CACHE STATISTICS BY ALGORITHM *PR* OVER GRAPH sd.

Approach	L1-ref	L1-mr	L3-ref	L3-r	Cache-mr
Original	1.29E+12	55.27%	6.03E+11	46.94%	17.52%
Sort	1.28E+12	51.32%	5.62E+11	43.84%	16.99%
DBG	1.28E+12	50.84%	5.53E+11	43.10%	15.84%
Norder	1.28E+12	34.00%	3.11E+11	24.29%	7.92%
Gorder	1.28E+12	28.30%	2.30E+11	17.97%	**6.20%**
Sorder	1.28E+12	30.09%	2.59E+11	20.24%	**6.56%**

Comparison on cache miss ratio. We utilize perf tool [1] to collect CPU cache statistics of running graph algorithm *PR* on the smallest graph $flickr$ and the largest graph sd, and summarize the results in Table III and Table IV, respectively. Here *Original* means we run *PR* on the original vertex ordering of each graph. L1-ref denotes the number of L1 cache references. Since all cache accesses must firstly check L1 cache, thus L1-ref indicates the total number of cache references. All approaches have similar L1-refs, because running the same graph algorithm on the same graph (regardless of the vertex ordering) will involves similar number of cache accesses. L1-mr denotes the L1 cache miss ratio that is calculated as the ratio between L1 cache misses and L1-ref. Similarly, L3-ref denotes the number of L3 cache references. L3-r is the ratio of cache references checked in L3 cache, *i.e.*, $= \frac{\text{L3-ref}}{\text{L1-ref}}$. A small L3-r implies that most cache references are hit by the L1 and L2 cache [21]. The cache-mr denotes the percentage of cache reference misses in all three levels over L1-ref.

Table III shows that all approaches have small cache miss ratios, *e.g.*, $< 2\%$. This is because $flickr$ is relatively small, while the caches of our machine are sufficiently large. The reordering approaches except *Sort* can still reduce cache miss ratios through better data layout. For graph sd, the cache miss ratios of all approaches become much greater, as shown in Table IV. As an example, the original ordering has L3-r as 46.94% and cache miss ratio as 17.52%. *Sort* and *DBG* slightly reduce the miss ratio, while *Gorder* and *Sorder* produce much better vertex orderings that reduce the miss ratio by about 10%. *Gorder* achieves the smallest cache miss ratios in both graphs, and *Sorder* has quite close cache miss ratios as *Gorder* in both tables, with a small gap as 0.16% and 0.36%, respectively.

Comparison on reordering cost. We compare the reordering time in Table V. When the graph size becomes larger, the reordering time of all approaches increases. We also observe that *Gorder* spends the most time than other four approaches to reorder a graph, with much more overheads about two orders of magnitude. This is because *Gorder* needs to calculate locality scores for all vertices using Equation (1), resulting in high

TABLE V
COMPARISONS ON THE REORDERING TIME (UNIT: SECONDS).

Graph	*Sort*	*DBG*	*Norder*	*Gorder*	*Sorder*	*Speedup*
$flickr$	0.08	0.47	0.38	40.07	0.39	103×
$livej$	0.20	0.73	0.89	59.37	0.82	72×
$orkut$	0.13	0.64	0.40	83.78	0.38	223×
pld	2.07	6.89	18.34	4882.27	11.22	435×
it	1.60	2.50	7.41	178.08	4.67	38×
$twitter$	2.76	5.78	64.12	10134.90	24.07	421×
sd	4.44	12.06	35.30	10630.40	39.72	268×

computation complexity. The two lightweight approaches, *i.e.*, *Sort* and *DBG*, indeed have relatively smaller pre-processing overheads. Since sorting all vertices in descending order of in-degrees takes extra time, *Norder* introduces more reordering costs than *Sorder* in most cases. Among the five approaches, *Sorder* has the moderate reordering cost. We calculate the speedup rations of *Sorder* over *Gorder* on the reordering time, and list the results in the last column of Table V. With the comparable speedup performance as *Gorder*, *Sorder* instead greatly reduces the reordering overhead by $38 \sim 435$ times.

Impact of parameter κ. We study the impact of parameter κ on *Sorder* using graph pld, it, $twitter$, and sd, and present the results in Figure 6. A larger κ will aggregate more neighbors to form a hypernode, and thus incurs more computation time. When κ becomes larger, reordering time of pld, $twitter$, and sd also increase. On the contrary, it is less influenced by κ. We also find that the reordering time of sd is quite stable when $\kappa \leq 3$. The impact of κ on reordering time maybe mainly determined by the graph structures, which calls for further researches. We also test the impact on speedup by varying κ. Specifically, we take the execution time of $\kappa = 1$ for each combination of algorithms and graphs as the baseline, and calculate a relative speedup for other settings of κ. Figure 6 shows that $\kappa = 2$ can achieve the best speedup performance.

V. RELATED WORK

Reordering vertices of a graph ahead-of-time could improve the cache locality of most graph algorithms, and thus attracts many research efforts in recent years. As introduced in Section II-B, *Gorder* can achieve the best speedup performance, while incurring extremely huge computation overheads. ReCALL [12] operates on the graph reordered by *Gorder* to rearrange blocks of consecutive vertices to further improve spatial locality, while at the cost of introducing even more computations. Rabbit Order [2] primarily exploits the community structures to reorder a graph, while its reordering cost is still unacceptable. To avoid such a huge pre-processing cost, some lightweight approaches that heavily rely on the skewed degree distribution of graphs have been proposed [3], [4], [7], [13], [22]. They renumber hub vertices with a high priority, while keeping other vertices almost unchanged. As a result, they may badly destroy the community structures among vertices, resulting in sub-optimal orderings. Different from these works, *Sorder* well preserves the structural properties of real-world graphs, and can achieve the comparable speedup performance as *Gorder*, while introducing the moderate reordering cost.

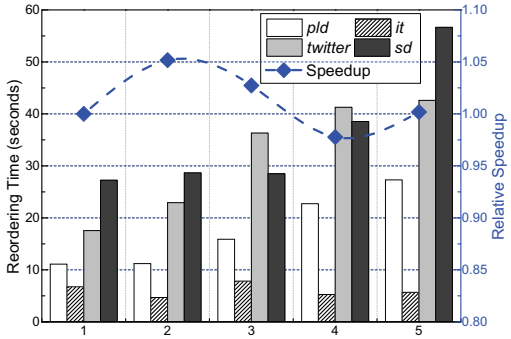

Fig. 6. The impacts of k on the reordering time and speedup performances.

There exist other alternative techniques to improve locality, *e.g.*, cache blocking [5], graph partitioning [11], vertex scheduling [17]. Different from *Sorder*, cache blocking and graph partitioning require to modify graph algorithms or data structures. Besides, graph reordering is complementary with vertex scheduling, and thus *Sorder* can be employed to further improve the performances of vertex scheduling techniques.

VI. CONCLUSION

This paper presents *Sorder* to well preserve the structural properties of real-world graphs for more effective graph reordering. *Sorder* consecutively renumbers vertices mainly by exploiting the neighborhood relations among vertices, and is further enhanced by the hypernode design. The vertex ordering derived by *Sorder* thus largely improves the temporal-spatial locality. Extensive experiments with typical graph algorithms and graphs demonstrate that *Sorder* achieves similar speedup performances as the sophisticated approach, while significantly reducing the pre-processing overheads, *e.g.*, at most by $435\times$.

REFERENCES

[1] perf tool. https://perf.wiki.kernel.org/index.php. [Online; accessed 28-August-2020].

[2] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit Order: just-in-time parallel reordering for fast graph analysis. In *IEEE IPDPS*, 2016.

[3] V. Balaji and B. Lucia. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *IEEE IISWC*, 2018.

[4] V. Balaji and B. Lucia. Combining data duplication and graph reordering to accelerate parallel graph processing. In *ACM HPDC*, 2019.

[5] S. Beamer, K. Asanović, and D. Patterson. Reducing pagerank communication via propagation blocking. In *IEEE IPDPS*, 2017.

[6] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. To push or to pull: on reducing communication and synchronization in graph computations. In *ACM HDPC*, 2017.

[7] P. Faldu, J. Diamond, and B. Grot. A closer look at lightweight graph reordering. In *IEEE IISWC*, 2019.

[8] P. Faldu, J. Diamond, and B. Grot. Domain-specialized cache management for graph analytics. In *IEEE HPCA*, 2020.

[9] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, 2008.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. In *USENIX OSDI*, 2012.

[11] G. Karypis and V. Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[12] K. Lakhotia, S. Singapura, R. Kannan, and V. Prasanna. ReCALL: reordered cache aware locality based graph processing. In *IEEE HiPC*, 2017.

[13] E. Lee, J. Kim, K. Lim, S. H. Noh, and J. Seo. Pre-select static caching and neighborhood ordering for BFS-like algorithms on disk-based graph engines. In *USENIX ATC*, 2019.

[14] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In *ACM WWW*, 2008.

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

[16] C. Magnien, M. Latapy, and M. Habib. Fast computation of empirically tight bounds for the diameter of massive graphs. *Journal of Experimental Algorithmics (JEA)*, 13:1–10, 2009.

[17] A. Mukkara, N. Beckmann, and D. Sanchez. Cache-guided scheduling: exploiting caches to maximize locality in graph processing. In *1st International Workshop on Architecture for Graph Processing*, 2017.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation algorithm: bringing order to the web. In *WWW*, 1998.

[19] P. Rozenshtein and A. Gionis. Mining temporal networks. In *ACM SIGKDD*, 2019.

[20] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM PPoPP*, 2013.

[21] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *ACM SIGMOD*, 2016.

[22] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *IEEE BigData*, 2017.