# Workload Balancing via Graph Reordering on Multicore Systems

YuAng Chen and Yeh-Ching Chung

**Abstract**—In a shared-memory multicore system, the intrinsic irregular data structure of graphs leads to poor cache utilization, and therefore deteriorates the performance of graph analytics. To address the problem, prior works have proposed a variety of lightweight reordering methods with focus on the optimization of cache locality. However, there is a compromise between cache locality and workload balance. Little insight has been devoted into the issue of workload imbalance for the underlying multicore system, which degrades the effectiveness of parallel graph processing. In this work, a measurement approach is proposed to quantify the imbalance incurred by the concentration of vertices. Inspired by it, we present *Cache-aware Reorder (Corder)*, a lightweight reordering method exploiting the cache hierarchy of multicore systems. At the shared-memory level, Corder promotes even distribution of computation loads amongst multicores. At the private-cache level, Corder facilitates cache efficiency by applying further refinement to local vertex order. Comprehensive performance evaluation of Corder is conducted on various graph applications and datasets. Experimental results show that Corder yields speedup of up to $2.59\times$ and on average $1.45\times$, which significantly outperforms existing lightweight reordering methods. To identify the root causes of performance boost delivered by Corder, multicore activities are investigated in terms of thread behavior, cache efficiency, and memory utilization. Statistical analysis demonstrates that the issue of imbalanced thread execution time dominates other factors in determining the overall graph processing time. Moreover, Corder achieves remarkable advantages in cross-platform scalability and reordering overhead.

**Index Terms**—Multicore system, cache locality, workload balance, graph processing

## 1 Introduction

Graph analytics has been a fast growing research field covering applications in diverse domains, such as protein structure identification [1], social network analysis [2], and fraud detection [3]. At the same time, data generated in these domains continuously grow in size and complexity, which require efficient processing systems for large-scale graphs.

To achieve high performance, various graph processing frameworks have been designed for different use scales. At a small scale where resources are limited (e.g., a laptop with 4 cores and 16 GB memory), the out-of-core frameworks utilize the *external* memory (e.g., disks) to temporarily offload the work [4], [5]. At a larger scale, such as a server-class multicore machine with 40 cores and 256 GB memory, *shared*-memory frameworks are able to exploit high parallelism as well as high capacity [6], [7]. When the graph size is too large (e.g., Terabytes) to fit into a single machine, the graph processing scales out to a cluster of multiple machines by deploying *distributed*-memory frameworks [8], [9].

In this paper, we narrow down our research scope within the shared-memory graph processing on the single-machine multicore system. Without the overheads of intensive disk accesses and network traffics, the shared-memory frameworks oftentimes outperform their disk-based and distributed counterparts [9], [10]. Nevertheless, suffering from highly randomized memory access patterns, the computing power of underlying multicore platforms still remains very much underutilized [11], [12].

The main reason accounting for the inefficiency of shared-memory multicore graph processing is the irregular pointer-based data structure of graphs. It behaves as the opposite of regular sequential data structure, e.g., arrays and matrices. When a graph is processed, a large volume of communication is invoked between computations at each vertex or edge. For the memory, the irregular pattern of communication is translated into random memory accesses in terms of data loads and stores.

Moreover, the irregular connectivity incurs a distinctive property commonly found in natural graph datasets: the *skewed* power-law degree distribution, in which a tiny fraction of vertices contribute to the majority of edges [13], [14]. These vertices are called *hot* vertices, while the remaining vertices with less connections are named *cold* vertices. Casting a graph into the memory, hot vertices are preferable for caching, because they comprise a large portion of computation but a minor portion of memory usage. As for the cold vertices, they scatter all over the memory and are randomly accessed, thereby causing high cache misses rate.

Hot vertices randomly spread throughout the whole graph but not evenly. Naturally, they tend to aggregate and occur in concentration, which we refer to as *graph locality*. The impact of graph locality on cache behaviors is interpreted as cache locality [11], [12], [15], [16]. Characterized by the reference pattern, cache locality can be further classified into two types: *temporal* and *spatial*. A group of high-degree

• *The authors are with The Chinese University of Hong Kong, Shenzhen, Guangdong 518172, China. E-mail: yuangchen@link.cuhk.edu.cn, ychung@cuhk.edu.cn.*

*Manuscript received 7 Feb. 2021; revised 7 Aug. 2021; accepted 9 Aug. 2021. Date of publication 18 Aug. 2021; date of current version 15 Oct. 2021. (Corresponding author: Yeh-Ching Chung.) Recommended for acceptance by R. Prodan. Digital Object Identifier no. 10.1109/TPDS.2021.3105323*

vertices shows high temporal locality if frequently accessed. Also, they exhibit high spatial locality if stored in nearby memory allowing for continuous reading and writing.

Besides the cache efficiency, the irregularity of graphs also affects the effectiveness of parallel computing scheme. Some local subgraphs (i.e., subsets of a graph that is partitioned according to certain criteria or algorithms) inside the graph accommodate more vertices or richer connections than others. Therefore, they tend to demand intensive computation. The difference among subgraphs raises imbalanced workloads amongst working threads in a parallel graph processing system [11].

To utilize graph locality and improve cache efficiency, a variety of lightweight graph reordering methods have been proposed [15], [17], [18]. These approaches, in principle, reorder vertices such that hot vertices and cold ones are separated into different memory locations. Therefore, hot vertices are allocated in contiguous memory space and frequently requested by processor cores. At a result, they have higher chances to be retained within the cache lines, which allows for better cache utilization.

However, graph reordering does not necessarily guarantee performance boost. In de facto, it often leads to a slowdown if an improper strategy is deployed [15]. The gathering of hot vertices aggravates the imbalanced distribution of computation load [18], forcing the main process to wait for the longest thread to synchronize. Moreover, reordering techniques inevitably disrupt the internal structure of graphs [17], such as communities of common friends in social graphs. The finer-grained the reordering is, the more structural information it loses.

To overcome aforementioned limitations, we propose *Cache-aware Reorder*, namely *Corder*, a lightweight reordering method based on the characteristics of cache. Our contributions can be summarized as three folds:

- We demonstrate that graph analytics in multicore systems suffers from imbalanced workloads, besides poor cache locality. A novel metric, namely Locality-Skew, is designed to quantify the inherent imbalance inside graphs. It incorporates the knowledge of graph locality and power-law skew. Further, it can be used to evaluate the impact of graph reordering techniques on the balance of a graph.
- Inspired by the analysis above, we present Corder, an efficient reordering method with the awareness of cache. First, a graph is partitioned by Corder to fit the vertex subsets into the private caches of multicores. Then, in the main memory, Corder facilitates load balance by evenly distributing hot vertices across the partitions. Finally, within the cache, it enhances graph locality via separating hot vertices from cold ones.
- To locate the source of performance gain achieved by Corder, we investigate the multicore activities in the aspects of thread behaviors, cache utilities and memory dynamics. Based on statistical approaches, a performance model is derived that profiles the relative importance of the aforementioned three factors. Numerical results show that the thread holds a dominant position in contributing the acceleration.

TABLE 1
Graph Properties in Terms of Hot Vertices Percentages ($V$), Edge Coverage of Hot Vertices ($E$), and Average Degree ($D$)

| Graphs | Descriptions | $V$ (%) | $E$ (%) | $D$ |
|---|---|---|---|---|
| *urand* | Synthetic Graph[19] | 51 | 59 | 32 |
| *kron* | Synthetic Graph[19] | 8 | 93 | 31 |
| *pld* | Pay-Level-Domain[20] | 12 | 88 | 14 |
| *live* | Live Journal[21] | 24 | 80 | 14 |
| *wiki* | Wiki Links[22] | 19 | 94 | 9 |
| *twitter* | Twitter Follower[23] | 9 | 79 | 35 |
| *mpi* | Twitter Influence[24] | 11 | 81 | 38 |

Following the introduction, Section 2 offers preliminaries and related state-of-the-art research. Section 3 describes the designs of Skew-Locality and Corder in detail. Section 4 presents extensive performance evaluations in both theory and experimentation. Finally, this paper is concluded in Section 5.

## 2 Background and Related Work

### 2.1 Basic Concepts

#### 2.1.1 Skewed Degree Distribution

It is rare to see that a real-world graph that is not skewed with respect to its degree distribution [13], [14]. The skewed degree distribution, following the power law, means that a small fraction of vertices are responsible for a major faction of edges. Mathematically, the portion of vertices that have $k$ connections to other vertices can be formulated as $p(k) \sim k^{-\gamma}$, where $\gamma$ is a parameter empirically ranging in [2,3] . Such inequality can be interpreted differently in various contexts. For instance, a celebrity has enormous social influence in social network graphs, or a search engine website sends billions of requests to other websites on web graphs.

In this paper, we use out-degree as the default measurement for the degree of a vertex. Vertices with degrees higher than the average degree of the graph is considered as the hot vertices, while the rest of vertices in the graph are the cold ones. Table 1 lists the percentage of hot vertices and their edge coverage across various graphs. Detailed descriptions regarding these graphs are provided in Table 4 in Section 4. In skewed graphs, 8-24 percent hot vertices compose 79-94 percent edges. Graph *urand* is not skewed, since over half of vertices in it are classified as hot.

#### 2.1.2 Graph Locality

The placement of graph vertices and the relationship among them pose non-negligible impact on graph locality. Consider the examples in Fig. 1 and assume the vertices are stored in an array in the order of vertex ID. We suppose that a graph algorithm examines the relationship between every vertex and its neighbors in sequence. Our goal is to observe the access pattern relating to vertices 0 and 1. In the random graph of Fig. 1a, vertex 0 and its neighboring vertices 2 and 5 are first evaluated, then followed by the vertex 1 and its adjacent vertices 3 and 4. Therefore, the access pattern is 0-2-5-1-3-4, where each vertex is visited only once (i.e., poor temporal locality), and vertices are not co-located in the array (i.e., poor spatial locality).

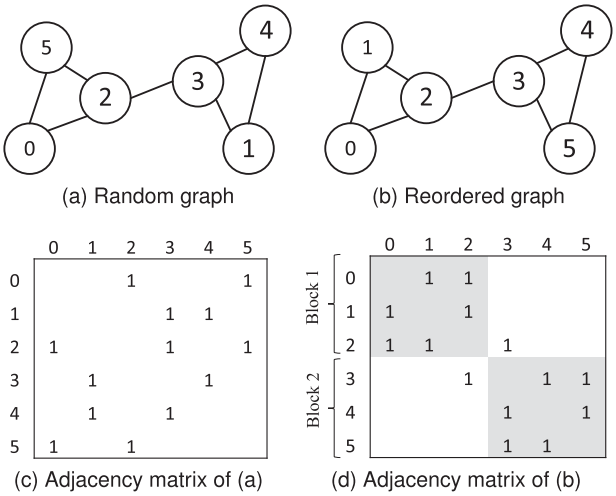



Fig. 1. Graphs and their belonging adjacency matrices. Graph (b) is reordered from graph (a) by assigning continuous ID numbers to neighboring vertices. The reordered vertices constitute two diagonal blocks, which are depicted with shadow color in matrix (d).

The graph in Fig. 1b is optimized on the vertex order, such that nearby vertices are labeled with close ID number. As a result, the access pattern corresponding to vertices 0 and 1 becomes 0-1-2-1-0-2 in the reordered graph. This pattern achieves good temporal-spatial locality, because vertices 0, 1, 2 are *repeatedly* accessed in *continuous* memory space.

The graph locality can also be directly observed from the adjacency matrix. Fig. 1c presents the adjacency matrix of the random graph. The elements of the matrix, representing the connections of vertices, are sparsely distributed. By contrast, the reordered adjacency matrix in Fig. 1d, containing two dense diagonal blocks, exhibits better graph locality.

## 2.2 Related Work

### 2.2.1 Graph Reordering

The random and sparse distribution of hot vertices in memory incurs severe underutilization of caches. To improving cache efficiency, prior works have proposed various graph reordering methods [12], [15], [17], [18]. These aim to search for an (approximately) optimal permutation of vertex order to improve graph locality, such that frequently accessed vertices are placed together. The examples of state-of-the-art graph reordering methods are described as follows:

*Gorder* [12] offers significant performance boost to graph applications by leveraging the internal structure of graphs. It undertakes a sophisticated analysis of the connectivity of vertices. Consecutive IDs are relabeled to vertices that share common neighbors. Hence, these neighboring data are reused in cache when vertices are consecutively processed. Nevertheless, Gorder tends to cost *hours* to accomplish the reordering procedure, while the graph applications are typically executed within minutes or even seconds (see Table 8). For this reason, it is impractical to deploy Gorder, especially when the graph is constantly changing.

Gorder can be considered as a representative example of heavyweight reordering methods that require prohibitively high overhead [12], [25], [26], [27]. Calling for pragmatism, lightweight degree-based reordering methods with low computation complexity are proposed [15], [17], [18]:

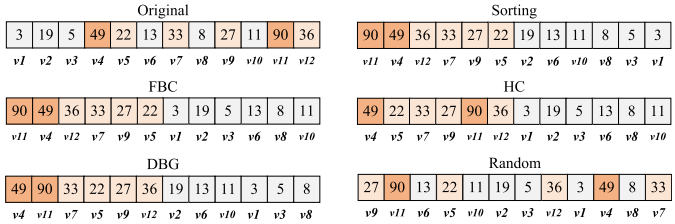



Fig. 2. Vertex placement by different reordering techniques. The degrees of vertices are filled inside the box. Cold vertices (degree $\leq$ 20) are painted with grey, hot vertices are colored, and very hot vertices (degree $>$ 40) are depicted with darker shade. Indexes below the box are the original vertex IDs.

*Sorting* places all vertices in descending order of degree. It is the most straightforward method for degree-based reordering. However, as illustrated in Fig. 2, the original graph structure is entirely reorganized, even including the relative order of the vertices. Moreover, sorting leads to extremely imbalanced workloads, imposing a considerable adverse effect on the overall performance of parallel graph processing.

*Frequency Based Clustering* (FBC) [18], also referred to as Hub Sorting, only applies sorting to hot vertices, while keeping remaining cold vertices unsorted. FBC improves the graph locality at the cost of fully rearranging the structure of hot vertices. Nonetheless, it still incurs high imbalance due to the sorting of hot vertices

*Hub Clustering* (HC) [15] is a variation of FBC. It separates hot and cold vertices into two memory spaces. HC does not employ sorting to any vertices. Therefore, the relative orders of hot vertices and cold vertices are maintained in their respective segments. HC reaches an optimal compromise between workload balance and graph locality.

*Degree Based Grouping* (DBG) [17] employs coarse-grain sorting to the graph. Vertices are allocated into different groups with exclusive degree ranges: $[0, \frac{\overline{D}}{2})$ and $[2^{n-1}\overline{D}, 2^n\overline{D})$, where $n = 0, 1, 2, \ldots$ and $\overline{D}$ is the average degree. Then, these groups are sorted in descending order. To preserve graph structure, the relative order of vertices inside each group remains intact. DBG can be viewed as an intermediate between Sorting and HC. Fig. 2 visualizes the degree-based vertex placement policy. In addition, the random reordering (RND) is used as a benchmark for later analysis. RND shuffles a graph, therefore completely eliminating the graph locality and structure. On the other hand, it generates a uniformly distributed random permutation of vertices, which yields a uniform distribution of workloads.

In brief, prior reordering methods mainly focus on enhancing the graph locality for performance boost by concentrating hot vertices. However, the experimental results in Section 4 shows that the scalability of these methods is seriously limited. They are effective only in a particular framework. In most cases, the performance of downstream graph applications is deteriorated due to imbalanced workloads caused by the concentration of hot vertices.

Besides graph reordering, various optimization techniques are proposed to enhance cache locality [28]. For example, graph-specialized cache management is designed to enhance the spatial-temporal locality at the last level cache [29]. Cache-aware partitioning/blocking methodologies subdivide a graph into cache-able subgraphs for the improvement of graph locality [11], [18], [30]. Oftentimes, the data layouts are

fine-tuned to cooperate with the proactive graph caching strategies [4], [10], [31].

#### 2.2.2 High-Performance Graph Processing

A diversity of graph analytics frameworks have been developed for high-performance graph processing. Many of these can be generalized into two types: *vertex-centric* and *edge-centric*.

The vertex-centric paradigm is widely adopted in contemporary frameworks [6], [32], [33], [34], [35]. It enables every thread to process arbitrary vertices in a graph. This paradigm propagates data through the graph in one of two directions: a vertex pushes data along its outgoing edges; otherwise, it pulls the data along its incoming edges. In the pushing flow, a race condition occurs when multiple threads access a common vertex. Hence, heavy use of synchronization primitives is involved during graph processing, including atomic operations and mutexes. In the pulling flow, locks are removed, but a full scan of incoming edges is required. Therefore, a redundancy is introduced when only a subset of vertices are active in an iteration [36].

To improve cache locality and alleviate thread contention, the edge-centric paradigm is utilized [4], [37]. Such paradigm streams on edges instead of vertices, so the graph data are prefetched before being processed. Nevertheless, it delivers suboptimal results to graph applications with dynamic active vertices, such as BFS. This is because all edges, including the unrelated ones, are streamed in every iteration [28].

With the awareness of cache hierarchy, a series of *partition-centric* graph processing frameworks are designed based on Compressed Sparse Row (CSR) segmentation [7], [18], [30], [38]. The input graph along the vertex array of the CSR format is first subdivided into partitions, such that the vertex set of each partition can fit into the cache. Then, each partition is exclusively processed by one core, so that random accesses are limited to local partition within the cache. Finally, updated results of each partition are propagated via inter-partition edges to the destination partitions. The partition-centric paradigm significantly improves cache performance and completely avoid atomic operations as well as mutexes.

Connecting two partitions in different cores, the inter-partition edge (inter-edge) is a distinct advantage of the partition-centric graph framework over the vertex-centric one [7]. An inter-edge transfers the message (e.g., updated rank value in PageRank application [39]) from a source vertex to a target partition. After being received, the message is then forwarded to destination vertices via local propagation. Through this process, multiple vertex-to-vertex edges are compressed into to a single inter-edges. The exercise of edge compression significantly reduces the memory traffic, and thus boosts the performance of graph processing [30].

## 3 Cache-Aware Reordering

We propose a cache-aware graph reordering method, named Corder, by following the partition-centric paradigm. In this paradigm, graph algorithms are parallelized by unrolling loops with dynamic schedulers [7]; that is, loop iterations are dynamically assigned to threads for workload balance. Based on the dynamic scheduling policy, Corder is aimed to further fine-tune the computational cost of each iteration via replacing the data (i.e., reordering the graph). The abstraction of Corder is illustrated in Fig. 3.

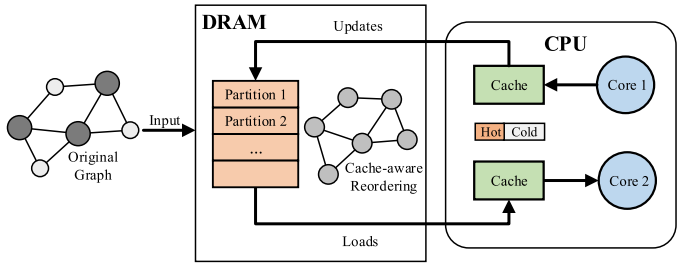



Fig. 3. Conceptual design of Corder. Grey cycles represent subgraphs. Different grey colors and sizes indicate imbalanced workloads. Corder reorders the graph such that the vertex subset of each subgraph can fit into the private caches of multicores with fairly balanced loads. Within the cache, hot vertices and cold ones are segmented to improve cache locality.

For the design of Corder, we first dig deeper into the property of graphs in Section 3.1. A metric, namely Locality-Skew, is formulated to estimate the inherent imbalance inside a graph. Accordingly, in Section 3.2, we present a swap-based reordering method that evenly distributes the hot vertices for load balance. Then, in Section 3.3, to optimize cache locality, hot vertices are locally concentrated within L2 cache. Finally, Section 3.4 generalizes the prior procedures and proposes Corder, which facilitates efficient implementation and exploits high parallelism.

Additionally, it is worth mentioning that finding the optimal partitioning as well as ordering scheme for a graph has been proved to be a NP-hard problem [12], [40]. Hence, the idea of Corder is developed based on the heuristics from the widely used graph processing frameworks and reordering methods. Though Corder is initially devised as an accelerator for the partition-centric graph processing paradigm, it exhibits strong scalability onto the vertex-centric frameworks too (see Section 4.7).

### 3.1 Estimating Imbalance

Despite the improvement in graph locality, the aggregation of hot vertices aggravates the uneven placement of hot vertices among subgraphs, and thus exacerbates load imbalance in a multicore system. To quantify the skewed distribution of *graph locality* incurred by concentration of vertices, we define the term "Locality-Skew".

To the best of our knowledge, there is no means to quantitatively measure the graph locality [11], [15]. The closest analogy is provided by Gorder, where the locality of two vertices is calculated for co-placement [12]. The extensive calculation of vertex pairs throughout the entire graph causes an excessively high computational cost (see Table 8). Hence, our primary task is to provide an efficient measurement approach for the graph locality. Given that the graph locality is mainly determined by the *connectivity* among the vertices in local *subgraphs*, the measurement task could be decomposed into two steps: (1) partition the graph, and (2) approximate the connectivity.

We perform the partitioning based on the characteristic of cache hierarchy. For a modern multicore machine, its cache is typically comprised of three level hierarchy according to the speed/storage. Level-1 (L1) cache operates at the

highest accessing speed but with the smallest storage capacity. Level-2 (L2) cache holds the medium-size storage by sacrificing the speed. Last Level cache (LLC) offers the lowest speed but the largest capacity. Every core is assigned with its own private L1 and L2 caches, whilst the LLC is shared between cores. According to these features, we split the vertex set of a graph into numerous partitions, the size of which is fixed to L2 cache size. Hence, each subgraph along its vertex array can be fitted in the private L2 cache of one core. The design choice of L2 cache achieves a balance between speed and storage.

Since the size of a vertex subset is predetermined, the graph locality now depends on the connectivity of vertices inside the subgraph. Here, we simply approximate the connectivity of a subgraph by accumulating the degrees of all vertices inside it. This is because, on one hand, the degree represents the number of edges one vertex is connected to. Vertices with higher degrees are likely to involve heavier computation. Thus, subgraphs with larger sum of degrees tend to associate with larger workloads. On the other hand, the summation of degrees is a simple arithmetic operation that requires no in-depth analysis, e.g., leveraging the internal structure of graphs.

In concise words, we evaluate the graph locality by *the cumulative total of vertex degrees in a subgraph with a fixed size of vertex array equal to L2 cache*. Furthermore, the localities (i.e., total sum of degrees) of different subgraphs vary with the skewed distribution of degrees. To measure the variation, we thereby introduce "Locality-Skew".

The calculation of Locality-Skew consists of three steps. First, the locality of each subgraph is obtained. Next, subgraphs are sorted by their localities in descending order. Therefore, the subgraphs at the top are loaded with much higher localities than these at the end. Last, the total locality of top $\lambda$ percent subgraphs is divided by the counterpart of end $\lambda$ percent to compute Locality-Skew

$$\text{Locality-Skew} \quad = \frac{\sum_{s \in top_\lambda} L(s)}{\sum_{s \in end_\lambda} L(s)}. \tag{1}$$

In Eq. (1), $s$ is the sorted subgraphs, $L(\cdot)$ stands for the locality of a subgraph (i.e., the sum of degrees). Table 2 presents the measurement of Locality-Skew with varied $\lambda$ (denoted as L-S$_\lambda$) across different graphs. *urand* is a perfectly unskewed and balanced graph. *kron* and *pld* are slightly imbalanced. The imbalances of *live* and *wiki* become severer as their L-S grow higher. As for *twitter* and *mpi*, hot vertices are concentrated in the 1 percent subgraphs, exhibiting extremely unequal distribution.

By absorbing the knowledge of graph locality and skewed degree distribution together, Locality-Skew offers us a new perspective to observe the features of a graph. For instance, the conventional measurement in Table 1 indicates *kron* is highly skewed, because 8 percent vertices compose 93 percent edges in this graph. Nevertheless, as listed in Table 2, its L-S$_{1\%}$ = 1.20 implies that it is, in fact, fairly balanced.

It should be emphasized that we aim to provide an efficient estimation regarding the graph property rather than a precise solution. Researchers have devoted much effort to designing fine-grained algorithms for the exploration of graph features, including graph reordering [12], graph partitioning [41], subgraph finding [42] and community detection [43]. However, the complexity of these algorithms prohibits their deployments in the context of "lightweight". Although our approaches, such as partitioning a graph by the L2 cache and sorting subgraphs instead of vertices, are rather coarse-grained, they considerably simplify implementation process and reduce computation complexity. Moreover, The estimation results clearly reveal the hidden property of graphs, and inspires a new direction in optimizing graph analytics: workload balance.

TABLE 2
Locality-Skew of Graphs With Varied $\lambda$

| $\lambda$ | 1% | 10% | 20% | 30% | 40% | 50% |
|---|---|---|---|---|---|---|
| *urand* | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| *kron* | 1.20 | 1.13 | 1.10 | 1.08 | 1.07 | 1.06 |
| *pld* | 2.83 | 1.86 | 1.61 | 1.46 | 1.36 | 1.29 |
| *live* | 25.89 | 21.24 | 13.19 | 9.34 | 6.54 | 4.48 |
| *wiki* | 39.12 | 17.60 | 12.04 | 9.03 | 7.17 | 5.69 |
| *twitter* | 192.12 | 30.61 | 15.56 | 10.47 | 7.90 | 6.35 |
| *mpi* | 432.43 | 56.12 | 22.69 | 12.82 | 8.21 | 5.93 |

*The larger a value, the more imbalanced a graph.*

### 3.2 Balancing Workloads

The summation of vertex degrees allows us to approximate the locality of L2-cache-sized subgraphs. More importantly, we are able to estimate the imbalance of workloads among these subgraphs based on Locality-Skew. To address the issue of imbalance, we develop a reordering strategy that swaps the hot vertices out of the overloaded subgraph.

**Algorithm 1.** Swap Reorder

```
Input:graph
Output:subgraphs
 1: subgraphs[N_sub] ← split(graph, N_sub)
 2: for all sub ∈ subgraphs do
 3:   /* is current sub overflowed with hot vertices? */
 4:   if isOverFlow(sub) then
 5:   /* swap hot vertices for cold ones with next subgraph */
 6:     swap(sub.hotVtx, sub.next.coldVtx)
 7:   else
 8:     swap(sub.coldVtx, sub.next.hotVtx)
 9:   end if
10: end for
```

Algorithm 1 generalizes the procedure of Swap Reorder, a swap-based reordering method. The initial step in line 1 is to split the graph into partitions in accordance with the L2 cache. The number of subgraphs (i.e., partitions) is determined by the number of vertices $N_v$ in the graph and the size of L2 cache: $N_{sub} = N_v/Size(L2)$. Then, based on the percentages of hot vertices $P_h$ listed in Table 1, we can calculate the total number of hot vertices in a graph dataset: $N_h = P_h \cdot N_v$. To equate the workloads, hot vertices are reallocated, such that each subgraph contains $n = N_h/N_{sub}$ hot vertices. If the number of hot vertices in a subgraph exceeds the threshold $n$, this subgraph is considered as overflowed (e.g., line 4). Once a subgraph is overflowed, its extra hot vertices will be swapped for cold ones with the next subgraph (e.g., line 6), and vice

TABLE 3
Locality-Skew of Graphs With Varied $\lambda$ After Applying Swap Reorder

| $\lambda$ | 1% | 10% | 20% | 30% | 40% | 50% |
|---|---|---|---|---|---|---|
| *urand* | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| *kron* | 1.23 | 1.14 | 1.11 | 1.09 | 1.07 | 1.06 |
| *pld* | 2.54 | 1.67 | 1.48 | 1.37 | 1.30 | 1.25 |
| *live* | 2.98 | 2.69 | 2.40 | 2.13 | 1.92 | 1.69 |
| *wiki* | 6.27 | 3.00 | 2.37 | 2.01 | 1.78 | 1.61 |
| *twitter* | 63.79 | 14.04 | 7.81 | 5.56 | 4.34 | 3.57 |
| *mpi* | 21.53 | 6.85 | 4.43 | 3.38 | 2.79 | 2.40 |

*The larger a value, the more imbalanced a graph.*

versa. The vertices to be modified are decided on a first-come-first-served basis, so their relative order is preserved.

Table 3 lists the results of Locality-Skew after swapping. The L-S of *urand* remains unchanged as 1.00, since it is already well balanced. The imbalance of *kron* is slightly enlarged as its L-$S_{1\%}$ increases from 1.20 to 1.23. The rest of graphs achieve a substantial reduction in L-S, such as the L-$S_{1\%}$ of *mpi* declining from 432.43 to 21.53.

### 3.3 Improving Graph Locality

Enlightened by aforementioned degree-based reordering techniques, we spot an opportunity for further optimization in graph locality. We adapt HC, which globally manipulates the vertex order at the level of main memory, to the level of local L2 cache. More specifically, after the deployment of Swap Reorder, the vertex order inside each subgraph within the L2 cache is further refined by classifying hot and cold vertices into two segments. As presented in Algorithm 2, it is simply just to append `hubCluster` to `swapReorder`. From the memory-cache hierarchy point of view, the workload is balanced in the main memory, and then the graph locality is improved in the private L2 cache.

**Algorithm 2.** Swap Reorder + Local HC

```
Input:graph
Output:subgraphs
1: subgraphs = swapReorder(graph)
2: /* local vertex reordering */
3: for all sub ∈ subgraphs do in parallel
4:   hubCluster(sub)
5: end for
```

However, a drawback of Swap Reorder is the interdependence of subgraphs, which leads to high computational complexity. To preserve the relative order, the swapping of vertices is incurred by every subgraph with the next one in a sequential order. This prohibits the swapping process from being parallelized. Considering the best case for Algorithm 1, the vertex distribution of a graph is so balanced that every subgraph contains exactly $n$ hot vertices. No subgraph is overflowed nor underflowed; hence, vertices are counted but never swapped. As a result, the time complexity of Algorithm 1 is $O(N_v)$. Then, in Algorithm 2, the local HC that classifies vertices into hot or cold vertices allows for parallelism. Its time complexity is $O(N_v/N_t)$, where $N_t$ is the number of threads. Adding these together, the time complexity of Algorithm 2 is $O(N_v + N_v/N_t)$ in the best case.

### 3.4 An Efficient Version - Corder

The sequential execution of Swap Reorder underutilizes the capacity of the multicore system. It induces high overhead costs and challenges the practical deployment of Algorithm 2. To parallelize the reordering procedure, an efficient version is thereby needed for pragmatism. Based on previous discussions, we organize the essential ideas of Swap Reorder plus local HC as follows:

1) The vertex set of a graph is subdivided into cacheable disjoint partitions of size equivalent to the L2 cache.
2) Each subgraph contains the same ratio of hot/cold vertices as the original graph.
3) Hot and cold vertices are segregated into two segments inside the subgraph.
4) The relative order of vertices in the hot segment as well as the cold one is preserved.

Accordingly, we propose *Cache-aware Reorder* (*Corder*), an efficient and effective algorithm fulfilling these ideas. The programming steps of Corder are outlined in Algorithm 3. It consists of two phases: *collecting* and *distributing*. In the collecting phase of lines 2-6, hot vertices and cold vertices are respectively collected into two distinct vectors. In the distributing phase of lines 10-12, $n$ hot vertices and $\bar{n}$ cold vertices are distributed from the vectors to all subgraphs, where $n = P_h \cdot N/N_{sub}$ and $\bar{n} = (1 - P_h) \cdot N/N_{sub}$.

**Algorithm 3.** Corder

```
Input:graph
Output:subgraphs
 1: /* collect phase */
 2: for all vertex ∈ graph.vertices do in parallel
 3:   if isHot(vertex) then
 4:     hotVertices.add(vertex)
 5:   else
 6:     coldVertices.add(vertex)
 7:   end if
 8: end for
 9: /* distribute phase */
10: for all sub ∈ subgraphs do in parallel
11:   move n vertices from hotVertices to sub
12:   move n̄ vertices from coldVertices to sub
13: end for
```

Compared with Algorithms 1 and 2, Corder exhibits its advantages in four aspects. First of all, the conditional statement `isHot(vertex)` of line 3 in Algorithm 3 is significantly simpler than the `isOverFlow(sub)` of line 4 in Algorithm 1. The former function verifies whether a single vertex is hot by simply comparing its degree with the average degree. By contrast, the later examines whether a subgraph is overflowed by checking all included vertices. This demands additional memory to record overflowing vertices. Second, distributing hot/cold vertices over all subgraphs (e.g., lines 11 and 12 in Algorithm 3) is easier than swapping them between subgraphs (e.g., lines 6 ad 8 in Algorithm 1). The distributing operation is nothing but to concatenate two subvectors, which is independent for every subgraph. Contrariwise, the swapping operation requires current subgraph to interact with the next subgraph(s) for vertex exchanging.

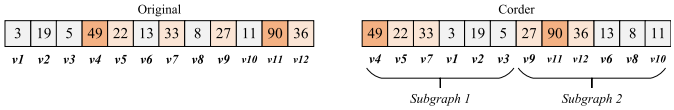



Fig. 4. Vertex placement after applying Corder. Assuming a private cache can hold six vertices, the graph of twelve vertices is partitioned into two subgraphs. In each subgraph, hot vertices (colored) are placed in the front of the cold ones (grey).

Third, the concatenation of hot and cold vertices (e.g., lines 11 and 12 in Algorithm 3) coincides with the completion of local HC (e.g., lines 3 and 4 in Algorithm 2). Therefore, local vertex reordering is not needed anymore. Last but not the least, the simplicity of Corder allows both *collect* and *distribute* phases to be parallelized, thereby significantly reducing overall reordering time.

The time complexity of the collecting phase and the distributing phase in Algorithm 3 are both $O(N_v/N_t)$. Therefore, the time complexity of Corder is $O(N_v/N_t + N_v/N_t) = O(N_v/N_t)$. It performs significantly faster than Algorithm 2, the best-case result of which is $O(N_v + N_v/N_t)$.

The vertex replacement result of Corder is exemplified in Fig. 4. A graph is partitioned into two subgraphs. Each subgraph contains the same number of hot vertices. Hence, the workload amongst two subgraphs (and therefore two processing cores) are roughly comparable. Inside the subgraph, hot vertices and cold vertices are segregated into two different memory locations. The grouping of hot vertices facilitates cache locality. At the meantime, the relative orders of hot vertices and cold ones are respectively maintained as original, so that the graph structure is preserved.

We expect that Corder can boost the performance of skewed graphs, e.g., all graphs except *urand*. In particular, the more "imbalanced" (i.e., higher Locality-Skew) a graph is, the better improvement Corder will lead to.

## 4 Evaluation

The performance of Corder is evaluated based on both theoretical and experimental approaches. First, a performance model for Corder is constructed, which incorporates multicore activities in diverse aspects. Then, we compare Corder with with state-of-the-art lightweight reordering methods. The behaviors of the multicore system are analyzed under the guideline of the proposed model. Besides, extensive experiments are undertaken to investigate the optimal partition size of Corder as well as its scalability onto vertex-centric paradigm.

### 4.1 A Performance Model for Corder

In this section, a theoretical evaluation is conducted to profile the correlation between Corder and various hardware activities in multicore systems. As previously described, Corder directly introduces two factors into the performance variability of graph processing on multicore systems: workload balance and cache utility. Besides, it might also raise memory traffic. This is because Corder balances the workload by evenly distributing hot vertices among partitions in different cores, thus elevating the inter-partition communication through the memory (verified in Section 4.4.3).

In the partition-centric paradigm, the number of inter-edges after compression poses substantial impact on the memory traffic (i.e., read + store) volume [30]. Since the access to memory is slower than the access to cache and CPU computation by orders of magnitude, it is considered as one of the most important factor in deciding the execution time of partition-centric graph processing [38]. Therefore, for a comprehensive analysis of underlying multicore systems, the memory dynamics is also taken into account as an indirect factor brought by Corder.

Thereby, the multicore activities are examined in three aspects: workload imbalance, cache utility, and memory dynamics. Particularly, we characterize the workload imbalance among multicores by the *the longest thread execution time* $T$, which signifies the most imbalanced load in a multithreaded task and bottlenecks the parallel computing. The cache utility is represented by the *L2 cache misses* $C$, as the cache efficiency is improved by Corder within the L2 caches of each processor core. Also, following prior works [30], [38], the memory dynamics is featured by *memory accesses volume* $M$. A simple linear model, which quantitatively portrays the contributions from the three factors, is proposed as follows:

$$speedup \propto w_1 \cdot r(T) + w_2 \cdot r(C) + w_3 \cdot r(M). \quad (2)$$

Function $r(\cdot)$ stands for the ratio of original result to the reordered result; that is, it accounts for the speedup of threads $r(T)$, the efficiency of cache $r(C)$ or the efficiency of memory $r(M)$ respectively. For example, $r(T)$ = {original thread execution time}/{thread execution time after reordering}. Thus, the model in Eq. (2) defines that the speedup of overall graph processing is proportional to the sum of improvements in thread performance, cache and memory efficiencies. Coefficients $w_1$, $w_2$ and $w_3$ express the weights of their corresponding factors in this model.

Based on the variations of their weights, the relationship of T, C, M can be classified into three typical cases:

Case 1: One factor plays a dominant role in deciding the graph processing time: $w_i >> w_{j\neq i} > 0\,(i, j \in \{1, 2, 3\})$.

Case 2: More than one factors offer comparable and non-trivial contributions to the performance: $w_i \approx w_{j\neq i} > 0$.

Case 3: At least one factor counteracts against others with a negative weight: $w_i < 0$.

It should be emphasized that the above cases do not list out all possible situations, nor occur in mutual exclusiveness. They are enumerated due to their representativeness in explaining the behavior of the model. Many other cases can be described by mixing these three.

Fig. 5 exemplifies the three cases in Kiviat charts. Cases 1 and 2 are straightforward: the performance gain delivered by Corder attributes to the shortened thread time, reduced cache misses and/or decreased memory traffic. However, in Case 3 of Fig. 5c, the negative weight $w_3 < 0$, belonging to $r(M)$, indicates a performance loss. The memory accesses are raised, while both thread time and cache misses are lowered. When the loss from memory outweighs the gains from thread and cache, the graph processing is expected to be decelerated, and vice versa.

The performance model of Eq. (2) depicts the relative importance of different multicore activities in building up the effectiveness of Corder. The coefficients in this model allow us to locate the root cause of the speedup offered by

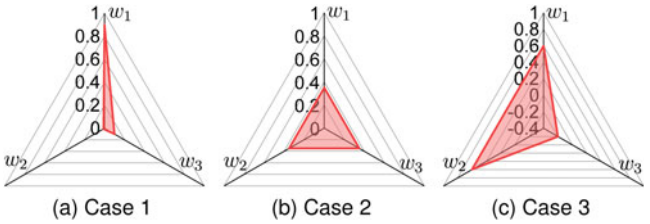



Fig. 5. The coefficients $w_1$, $w_2$, $w_3$ of performance model in three different cases. Case 1: one factor (with $w_1$) dominates others. Case 2: all factors contribute equally. Case 3: one factor (with $w_3$) is negatively correlated.

Corder; that is, the larger the weight value, the more important the corresponding factor. To obtain the values of those weights, comprehensive experiments are to be performed, which are detailed in following sections.

### 4.2 Experiment Setup

In our development environment, a dual-processor server is used. Each processor consists of 10 cores and each core has 2 threads. To eliminate contention on private caches, only 1 thread is enabled per core. In the memory hierarchy, the sizes of L1 cache, L2 cache, LLC and DRAM are 64 KB, 1 MB, 13.75 MB, 256 GB respectively.

Corder is developed in C++ on the basis of GPOP [7], a novel cache-aware graph processing framework designed for multicore systems. GPOP stores graphs in CSR for memory efficiency. It partitions a graph into subgraphs by equally splitting the vertex array into cache-able subsets. Each subgraph is exclusively processed by one core within its own cache. Following the authors' instruction, GPOP is parameterized with 20 threads and 1 MB partition size. A vertex occupies an unsigned 32-bit integer, so one subgraph contains 256K vertices. For reordering methods HC, FBC and DBG, we use the open-sourced code from the authors. All programs are compiled using G++ 9.3.0 with optimization level O3. The parallelization is realized with OpenMP. Our code is publicly accessible on GitHub.[1]

The performance of reordering techniques are evaluated on every pair of graph applications and graph datasets. The experimental results are obtained by averaging 5 rounds of executions. The activities of cache and memory are measured using the `perf` [44] and `likwid` [45] tools respectively on Linux. Also, the hottest vertex with the richest connections in every graph dataset is sorted out. They are used as the input for root-dependent applications that require a root vertex as the starting point for traversing, such as BFS and SSSP.

*Graph Datasets.* We select seven large graphs for evaluation. The statistic summary in Table 4 presents their numbers of vertices, numbers of edges, maximum degrees and references. In addition, the inter-edges per subgraph after compression is presented.

Graphs *live*, *twitter* and *mpi* are collected from social networks to depict user relationships [21], [23], [24]. Extracted by web crawlers, *pld* and *wiki* model the hyperlinks among the web pages [20], [22]. *kron* is a synthetic graph generated by Graph500 Kronecker generator with the scale of 23 [19]. Using the same tool, *urand* is generated, but follows uniform random distribution [19]. Here, *urand* serves as an example

1. https://github.com/yuang-chen/Corder-TPDS-21

TABLE 4
Statistic Summary of Graphs (K: Thousand, M: Million, B: Billion)

| Graphs | Vertices | Edges | Max Deg. | Refer. | Inter |
|---|---|---|---|---|---|
| *urand* | 67.1M | 2.1B | 66 | [19] | 7.9M |
| *kron* | 67.1M | 2.1B | 1.0M | [19] | 2.8M |
| *pld* | 42.9M | 0.6B | 3.9M | [20] | 1.6M |
| *live* | 4.8M | 68.5M | 2.0M | [21] | 0.8M |
| *wiki* | 18.3M | 0.2B | 9300 | [22] | 0.5M |
| *twitter* | 41.7M | 1.5B | 3.0M | [23] | 2.3M |
| *mpi* | 52.6M | 2.0B | 0.8M | [24] | 1.6M |

*Inter stands for the inter-edges per subgraph.*

of "not skewed and well balanced" graphs, though our work focuses on the optimization of skewed graphs. For this reason, the performance of *urand* is not taken into account when the we calculate the average result over the graphs.

*Graph Applications.* Performance experiments are applied on six representative applications:

- Pagerank (PR): a algorithm that iteratively computes the ranks of vertices based on their own attributes and edges. It is a fundamental algorithm widely applied in numerous academic and industrial scenarios [39].
- Pagerank Delta (PR-$\delta$): a variant of pagerank where vertices will be deactivated when their rank updates are smaller than a threshold $\delta$. Compared to PR, it allows faster converging rate and requires less memory traffic.
- Connected Component (CC): find subgraphs, or namely components, where any two vertices can be connected by a path. Components are isolated from others, in each of which vertices share the same label.
- Single Source Shortest Path (SSSP): from a given root vertex, compute the shortest distance to all vertices in a weighted graph by using Bellman Ford algorithm.
- Breadth-First Search (BFS): traverse the graph starting from a given root vertex layerwise.
- Parallel Nibble (PN): near a given vertex, discover a local cluster, the internal connections of which are enormously richer than its external connections [46]. The implementation is parallelized for optimization [47].

In particular, PR is the most sensitive application to the vertex order among the six. In every iteration of PR, all vertices stay active; that is, the property data of vertices are updated and propagated. By contrast, in other applications, only a subset of vertices participate in the computation. Since a vertex is deactivated and omitted from execution, its order becomes irrelevant. To this end, we select the hottest vertex as the root vertex as the input for those applications for the maximum activation.

### 4.3 Execution Time

Table 5 presents the graph processing time after applying different graph reordering methods. The processing time of graphs with original order (Orig.) is provided as the baseline for comparison. We can obtain the speedup by: speedup = {original processing time}/{reordered processing time} = 1/slowdown. Therefore, speedup $>$ 1 indicates performance

TABLE 5
Graph Processing Time of Graph Applications in Seconds After Applying Different Graph Reordering Methods

| | Orig. | Corder | HC | DBG | FBC | Sort | RND | Orig. | Corder | HC | DBG | FBC | Sort | RND |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Pagerank (1 iteration) | | | | | | | Pagerank Delta | | | | | | |
| *urand* | 1.23 | 1.19 | 1.13 | 1.16 | 1.14 | 1.21 | **1.15** | **7.24** | 7.92 | 7.92 | 7.59 | 8.58 | 10.54 | 8.17 |
| *kron* | 0.75 | **0.56** | 0.82 | 3.98 | 4.15 | 4.43 | 0.99 | 5.17 | **4.17** | 7.80 | 34.28 | 34.94 | 33.38 | 35.63 |
| *pld* | 0.27 | **0.24** | 0.26 | 0.66 | 0.62 | 0.68 | 0.55 | 2.32 | **2.29** | 3.01 | 5.08 | 4.87 | 5.21 | 2.79 |
| *live* | 0.06 | **0.04** | 0.06 | 0.10 | 0.12 | 0.12 | 0.05 | 0.35 | **0.26** | 0.47 | 0.66 | 0.73 | 0.82 | 0.41 |
| *wiki* | 0.20 | 0.12 | 0.22 | 0.23 | 0.23 | 0.23 | **0.09** | 1.33 | 0.76 | 1.38 | 1.32 | 1.40 | 1.39 | **0.63** |
| *twitter* | 0.76 | **0.45** | 0.95 | 1.40 | 1.64 | 1.81 | 0.72 | 7.81 | **3.58** | 10.70 | 13.36 | 16.88 | 18.89 | 7.42 |
| *mpi* | 1.74 | 1.09 | 1.66 | 1.70 | 1.82 | 2.05 | **0.97** | 16.47 | 7.95 | 13.40 | 11.30 | 13.16 | 13.75 | **6.34** |
| | Connected Component | | | | | | | Single Source Shortest Path | | | | | | |
| *urand* | 3.54 | 3.34 | 3.55 | 3.17 | 3.10 | 3.00 | **3.00** | 8.13 | 7.58 | 8.00 | 8.12 | 8.75 | **7.34** | 8.07 |
| *kron* | 2.09 | **1.14** | 2.21 | 6.87 | 7.94 | 7.45 | 2.34 | 1.81 | **1.53** | 3.94 | 8.08 | 9.65 | 9.72 | 2.25 |
| *pld* | 0.91 | **0.79** | 1.11 | 1.55 | 1.45 | 1.66 | 0.84 | 1.23 | **1.11** | 1.67 | 2.23 | 2.15 | 2.63 | 1.27 |
| *live* | 0.48 | **0.29** | 0.30 | 0.28 | 0.29 | 0.36 | 0.38 | 0.53 | **0.49** | 0.97 | 0.98 | 1.10 | 1.19 | 0.68 |
| *wiki* | 0.97 | **0.60** | 1.08 | 0.60 | 0.65 | 0.69 | 0.57 | 0.96 | **0.75** | 1.75 | 1.45 | 1.54 | 1.48 | 0.76 |
| *twitter* | 1.81 | **1.36** | 3.31 | 3.31 | 3.95 | 4.19 | 1.89 | 2.52 | **1.54** | 5.89 | 6.02 | 6.29 | 6.73 | 2.35 |
| *mpi* | 4.87 | **1.88** | 4.18 | 3.75 | 3.85 | 4.00 | 2.18 | 6.16 | 3.54 | 9.79 | 5.42 | 5.77 | 6.29 | **3.14** |
| | Breadth-First Search | | | | | | | Parallel Nibble | | | | | | |
| *urand* | 2.23 | 2.44 | 2.25 | 2.19 | 2.23 | 2.20 | **2.11** | 2.88 | 2.92 | 2.91 | 2.80 | 2.74 | 2.47 | **2.39** |
| *kron* | 1.35 | **1.26** | 1.08 | 3.25 | 3.71 | 3.55 | 2.63 | 0.76 | **0.68** | 0.69 | 1.46 | 1.43 | 1.40 | 0.77 |
| *pld* | 0.84 | **0.72** | 0.67 | 1.38 | 1.29 | 1.48 | 1.04 | 6.55 | **6.25** | 3.07 | 7.38 | 7.44 | 6.54 | 6.57 |
| *live* | 0.21 | **0.19** | 0.28 | 0.35 | 0.38 | 0.49 | 0.52 | 8.17 | **5.16** | 12.09 | 13.27 | 14.87 | 15.21 | 10.19 |
| *wiki* | 0.40 | **0.29** | 0.43 | 0.48 | 0.52 | 0.55 | 0.36 | 5.52 | **3.38** | 5.81 | 6.22 | 5.75 | 5.64 | 2.91 |
| *twitter* | 1.29 | **1.09** | 1.49 | 2.43 | 2.56 | 3.51 | 1.42 | 3.12 | **2.59** | 2.97 | 3.51 | 3.30 | 3.54 | 3.22 |
| *mpi* | 2.20 | **1.70** | 2.07 | 2.75 | 3.14 | 3.22 | 2.19 | 3.50 | **2.21** | 3.58 | 2.81 | 2.43 | 2.84 | 2.93 |

*The original processing time (Orig.) without reordering is provided as the baseline for comparison.*

improvement, while speedup $< 1$ (or slowdown $> 1$) means performance deterioration.

Corder outperforms state-of-the-art reordering methods in two aspects. First, it achieves the *highest* acceleration on the skewed graphs, which is up to $2.59\times$ and on average $1.45\times$. Second, Corder consistently boosts the performance of *every* graph application on *every* skewed graph. Slight slowdown occurs on graph *urand*, as it is an artificial graph with completely uniform distribution. Moreover, by analyzing the characteristics of graph datasets and graph applications, we acquire following findings:

*Finding 1.* The deployment of contemporary lightweight reordering techniques, including HC, FBC, DBG and Sort, often leads to considerable slowdowns. The average slowdowns of them are $1.09\times$, $1.37\times$, $1.43\times$ and $1.54\times$ respectively. The observation contradicts with the statement reported by [12], [15], [17] that speedup is expected. This contradiction ascribes to the "processing paradigm".

The *vertex-centric* paradigm (e.g., Ligra), as originally implemented underneath HC, FBC and DBG, addresses the graph at the granularity of vertex per thread. As a result, the issue of workload imbalance is relatively trivial, even though the degrees and corresponding workloads of the vertices could vary from 0 to millions as listed in Table 4.

The *partition-centric* paradigm adopted in our work, however, treats the graph at a much coarser granularity. A partition containing over hundred thousand vertices (e.g., 262,144 vertices in our experiment) acts as the basic unit for single thread to process. Hence, with the use of HC, FBC, DBG and Sort, hot vertices are concentrated into a small fraction of subgraphs. As listed in Table 6, the L-$S_{20\%}$ of graphs are significantly enlarged by those reordering methods. The gathering of hot vertices heavily exacerbates the imbalance issue and consequently becomes the primary reason for performance degradation. Contrariwise, Corder and RND lower the L-S of graphs and effectively alleviate the problem of workload imbalance. Thus, the execution time is shortened (though not always for RND).

*Finding 2.* Random reordering is occasionally beneficial to the performance. Sometimes it provides performance improvement even higher than Corder, such as PR on graph *wiki*. From the perspective of statistics, random reordering can be interpreted as uniformly distributing hot vertices across all subgraphs. After applying RND, subgraphs not only accommodate the same number of hot vertices, but also incur almost identical workloads. As presented the RND column in Table 6, RND achieves the smallest L-$S_{20\%}$

TABLE 6
Locality-Skew ($\lambda = 20\%$) of Graphs After Graph Reordering

| | Orig. | Corder | HC | FBC | DBG | Sort | RND |
|---|---|---|---|---|---|---|---|
| *urand* | **1.00** | 1.00 | 1.34 | 1.47 | 1.33 | 1.632 | 1.00 |
| *kron* | **1.10** | 1.11 | 57.71 | 57.71 | 95.01 | $\infty$ | 1.11 |
| *pld* | 1.61 | 1.48 | 32.30 | 32.30 | 52.48 | $\infty$ | **1.22** |
| *live* | 13.19 | 2.40 | 25.07 | 26.50 | 30.94 | 84.81 | **1.01** |
| *wiki* | 12.04 | 2.37 | 70.90 | 70.90 | 156.45 | $\infty$ | **1.02** |
| *twitter* | 15.56 | 7.81 | 21.27 | 21.28 | 28.30 | 101.47 | **1.42** |
| *mpi* | 22.69 | 4.43 | 42.55 | 42.55 | 98.01 | 746.39 | **1.05** |

*$\infty$ indicates that the sum of degrees over ending subgraphs, i.e., $\sum_{s \in end_\lambda} L(s)$ in Eq. (1), is zero.*

TABLE 7
L2 Cache Misses (in Billions) During 20 Iterations of Pagerank After Applying Different Reordering Methods

| | Orig. | Corder | HC | DBG | FBC | SRT | RND |
|---|---|---|---|---|---|---|---|
| *urand* | 41.95 | **40.78** | 41.90 | 42.56 | 42.70 | 42.75 | 44.41 |
| *kron* | 15.36 | **11.75** | 15.22 | 21.14 | 19.82 | 20.86 | 29.86 |
| *pld* | 6.28 | **5.12** | 6.90 | 6.72 | 7.72 | 7.42 | 9.87 |
| *live* | **0.49** | 0.56 | 0.52 | 0.61 | 0.67 | 0.70 | 1.05 |
| *wiki* | 1.11 | **1.08** | 1.18 | 1.35 | 1.48 | 1.46 | 2.62 |
| *twitter* | **9.61** | 9.82 | 9.96 | 13.72 | 13.05 | 16.08 | 23.72 |
| *mpi* | **10.97** | 11.88 | 13.22 | 16.36 | 17.16 | 18.76 | 32.51 |

in comparison with other reordering techniques. The problem of workload imbalance is solved by randomization.

However, the random placement of vertices completely destroys natural graph structure and eliminates graph locality. As the graph locality vanishes, cache misses are dramatically increased. For RND, there is an inevitable conflict between workload balance and graph locality. The overall performance is undermined when the loss from graph locality outweighs the gain from workload balance, and vice versa.

As shown in Table 6, graph *pld* is fairly balanced as its L-$S_{20\%}$ in the original order is low: 1.61. Applying RND on the graph, the L-$S_{20\%}$ is further reduced to 1.22, whereas the cache misses (e.g., for PR) are increased from 6.28 Billions to 9.87 Billions as in Table 7. In such scenario, the cache miss dominates the workload balance. Consequently, the execution time of PR per iteration on graph *pld* is extended from 0.27s to 0.55s due to the increased cache misses by RND. On the contrary, in the case of graph *mpi* that suffers from severe imbalance, RND effectively reduces the inequality of workloads, lowering L-$S_{20\%}$ from 22.64 to 1.05. Cache misses, though increased from 10.97 Billions to 32.51 Billions, are compensated by balanced workloads. As a result, the overall processing time of PR is shortened from 1.74s to 0.97s by the use of RND.

*Finding 3.* The higher Locality-Skew a graph is associated with, the better result Corder delivers. This meets our expectation proposed in the end of Section 3. Fig. 6 visualizes the correlation between the L-$S_{20\%}$ of graphs in original order and the speedups offered by Corder. Graph applications are increasingly accelerated as the Locality-Skew of graphs rises. For instance, graph *mpi* with L-$S_{20\%} = 22.69$ achieves $1.81\times$ speedup on average of all applications. By contrast, the average gain of graph *pld* with L-$S_{20\%} = 1.61$ is significantly lower, which is $1.10\times$ speedup. The ascending

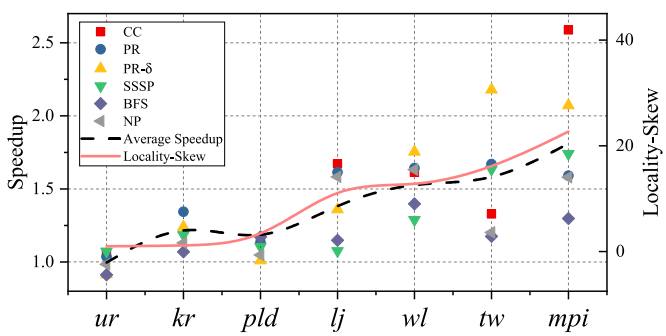



Fig. 6. The correlation between speedup and Locality-Skew ($\lambda = 20\%$).

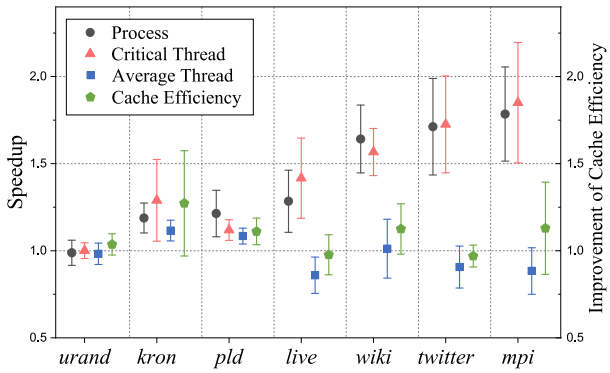



Fig. 7. The impact of Corder on thread behaviors and cache efficiency. The speedups (left) of graph processing, critical thread, and average thread are plotted with the improvement of cache efficiency (right). The results are calculated by the mean of different graph applications with 90 percent confidence interval.

trend of speedup towards the growing L-$S_{20\%}$ validates the effectiveness of Corder in balancing workloads.

## 4.4 Multicore Activities With Corder

### 4.4.1 Thread Behavior

In a multicore system where synchronization is required, the processing time of a parallel region mainly depends on the thread with the longest execution time. We define such thread as the *critical thread*, which is positioned in the most imbalanced point in a multi-threaded task. Meanwhile, we define the *average thread* as the representative of all threads. The performance of the average thread is nothing but the arithmetic mean of all threads within the parallel region. Fig. 7 illustrates the speedups of graph processing, the belonging critical threads and the average threads.

It can be observed that the performance of the process is tightly related with the critical thread. As the Locality-Skew grows from graph *urand* to *mpi*, the process and the critical thread are simultaneously accelerated. Corder addresses the imbalance issue by shortening the critical thread, and consequently accelerates the overall graph processing. Moreover, the correlation between the critical thread and the overall performance indicates that the performance gain mainly results from the balanced workloads (i.e., shortened critical threads). In Section 4.5, we will present detailed verification.

Although the overall process is boosted by Corder, the average thread completion speed may be actually slowed down. For instance, for PR on graph *live*, the overall processing time reduces from 63 ms to 39 ms per iteration (i.e., $1.61\times$ speedup), as its critical thread time decreases from 44 ms to 31 ms (i.e., $1.42\times$ speedup). Meanwhile, its average thread time of increases from 18 ms to 21 ms (i.e., $1.16\times$ slowdown). As Fig. 7 depicts, the behavior of the average thread is closely related with the cache efficiency, which is to be detailed in the next section.

### 4.4.2 Cache Utility

Since our optimization on the cache utility is focused on the L2 cache, we collect the L2 cache misses for evaluation. In particular, we characterize the *improvement of cache efficiency* (ICE) by the cache misses prior and subsequent to the deployment of graph reordering: ICE = {original cache

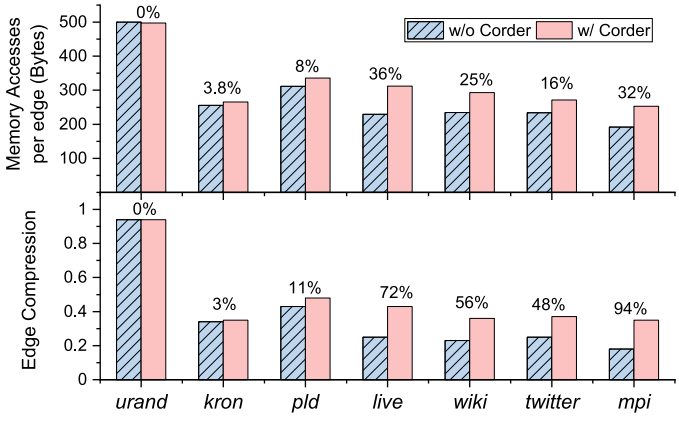



Fig. 8. Upper: Sustained memory bandwidth. Lower: No. of compressed edges compared to no. edges in the original graphs. Corder inflates graph inter-edges and thus raises memory traffic. Every pair of results (w/o versus w/ Corder) is denoted with the increased percentage.

misses}/{reordered cache misses}. When ICE $> 1$, cache misses are reduced and therefore the cache is better utilized, and vice versa. Due to space limit, only a subset of concrete data regarding the cache misses are presented in this paper, such as Table 7.

As depicted in Fig. 7, the ICE delivered by Corder shows that the behavior of the average thread corresponds with the trend of cache efficiency. Sometimes, the cache efficiency deteriorates due to the increased graph edges (see explanations in Section 4.4.3). Accordingly, the average speed of threads declines. However, the slowdown of the average threads cannot prevents the the overall processing from being speeded up.

For instance, the cache misses of PR on *live* grow from 0.49 Billions to 0.56 Billions (i.e., ICE = 0.86). At the same time, its average thread time extends from 18ms to 21ms (i.e., 1.16× slowdown). This phenomenon follows our common sense: poor cache utilization decelerates the execution of (a majority of) threads. Nevertheless, as discussed in last section, the critical thread is accelerated, which plays a crucial role in deciding the processing time of a parallel region. The reduction in execution time of critical thread compensates the increased computing cost of other threads. For this reason, the graph processing is accelerated.

To conclude, cache efficiency exhibits less importance compared to the issue of workload balance. Its behavior synchronizes with the average thread. However, its impact on the overall performance of graph processing is shadowed by the critical thread, specially for the graphs with high L-S, such as *wiki*, *twitter* and *mpi*.

#### 4.4.3 Memory Dynamics

Corder leads to an upsurge of inter-edges, which acts as a side effect to the partition-centric paradigm. The lower part of Fig. 8 shows the ratio of compressed edges (i.e., inter-edges) compared to original vertex-to-vertex edges. After employing Corder, hot vertices are evenly distributed among subgraphs, incurring frequent message exchanges between them inside different processing cores. Therefore, the inter-edges of graphs, representing the core-to-core communication, are increased.

The inflation of graph inter-edges results in heavier computation loads, thus more cache misses (discussed in Section 4.4.2) and memory traffic. Fig. 8 presents the growth of memory accesses (upper) due to the addition of edges (lower). Compared to graphs with low L-S (e.g., *kr*), skewed graphs with high L-S (e.g., *mpi*) tend to invoke less memory accesses per edge and thus consume lower memory bandwidth . Also, the impact of Corder on the skewed graphs are significantly larger too.

Corder burdens the efficiency of memory (i.e., $r(M)$ in Eq. (2)) because of the raised memory accesses incurred by additional inter-edges. The accesses to memory are typically slower than those to caches by orders of magnitude. Hence, the extra memory accesses might potentially decelerate the graph processing, though its negative impact is compensated by the improvement from other factors, such as the shortened critical thread. Comparing Fig. 7 with Fig. 8, it shows an inverse correlation between the speedup of graph processing and the memory efficiency; that is, the higher the speedup, the more the memory accesses, and the poorer the memory efficiency. Therefore, we expect a negative coefficient for $r(M)$.

### 4.5 The Derivation of Performance Model

Based on previous discussions, we obtain a preliminary observation about the influential factors for the speedup of partition-centric graph processing in the multicore system: the *workload imbalance* among threads dominates the *cache efficiency* and the *memory efficiency*, and moreover, the latter behaves oppositely to the trend of speedup.

For concreteness, we perform a multivariate regression on the observed data. It characterizes the association among the input variables and explains the dependence of the outcome on the inputs [48]. Specifically, we feed the model in Eq. (2) with the critical thread execution time, L2 cache misses and memory accesses. First, for simplicity, we only use the experimental results from PageRank, during which all vertices stay active. The fitted model is given as follows:

$$speedup = 0.87 \cdot r(T) + 0.27 \cdot r(C) - 0.16 \cdot r(M). \quad (3)$$

Eq. (3) validates our observation. The importance of the critical thread outweighs other factors. Cache efficiency is improved, though its contribution is relatively minor. Also, the memory access is associated with a negative coefficient, indicating it is itself a (minor) cost for the speedup. We visualize the fitted model in Fig. 9a. As described in Section 4.1, this model employs a mix of Case 1 and Case 3. Then, we use all data collected from all graph applications for regression. The resulting model is depicted in Fig. 9b and formulated as follow:

$$speedup = 0.92 \cdot r(T) + 0.05 \cdot r(C) + 0.08 \cdot r(M). \quad (4)$$

As Eq. (4) shows, the importance of the critical thread is further elevated, while the other two are drastically decreased. Such adjustment originates from the active vertex set – a new factor introduced by all graph algorithms except for PR. In every iteration, the active vertices dynamically change according to the internal structure of the graph, which considerably complicates the performance model (and goes beyond the scope of this paper). As a

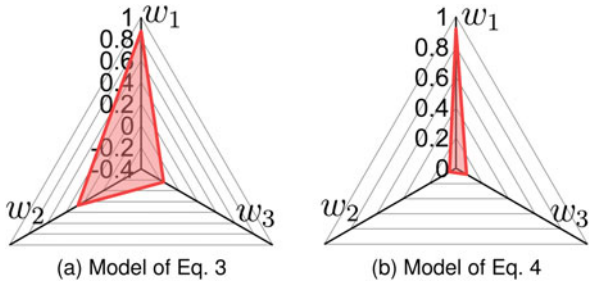



Fig. 9. The coefficients of performance models fitting the experimental data of PageRank (a) and all graph applications (b).

consequence, the weight of imbalanced threads is enlarged, falling into the category of Case 1.

The models proposed for Corder also explain the behaviors of other reordering methods. For example, as discussed in Findings 2, RND substantially boosts the performance of PageRank on the highly skewed graph *mpi*. After deploying RND, though the cache misses are increased by $3\times$ as in Table 7, the critical thread execution time is accelerated by $8\times$. Since the importance of critical thread (i.e., workload imbalance) outweighs the cache efficiency, the overall execution time is consequently shortened.

Lastly, it is worth noting that the performance models in Eqs. (3) and (4) are derived to rank the relative importance of the thread performance, cache utilization and memory traffic from a statistical point of view. The models simplifies the interaction among the counted factors and neglects many other issues, such as the dynamic active vertices and the choice of root vertex. Hence, we do not expect to solely rely on these models to precisely predict the performance of a given graph application.

### 4.6 Sensitivity Analysis of Partition Size

The partition size of Corder represents a trade-off. Small subgraphs facilitate fine-granularity workload balance; whereas large subgraphs promote high locality and preserve better graph structure. At the extreme, if each subgraph is so small that it contains only a single vertex, Corder becomes the RND reordering. On the other side, if the subgraph is large enough to feature the same size as original graph, there exist only one subgraph including all vertices. In such sense, Corder eventuates to be HC. However, as Table 5 shows, neither of them promises performance boost.

In order to evaluate the impact of Corder's partition size on the execution time of graph processing, we examine the performance of PR on various graphs. The PR is selected as the benchmark application for demonstration due to its high sensitivity to vertex order. Fig. 10 presents the execution time with partition size of Corder varying from 128 KB (32K vertices) to 16 MB (4M vertices). Meanwhile, the underlying GPOP is fixed with 1 MB partition size to eliminate extraneous variables.

As in prior experiments, graph *urand* remains almost unaffected by partition size due to its low graph locality and uniform distribution of vertex degree. For skewed graphs, there is a distinguish threshold at 1M bytes, which equals the size of L2 cache. As long as the partition size varies within 1M, Corder achieves the same level of

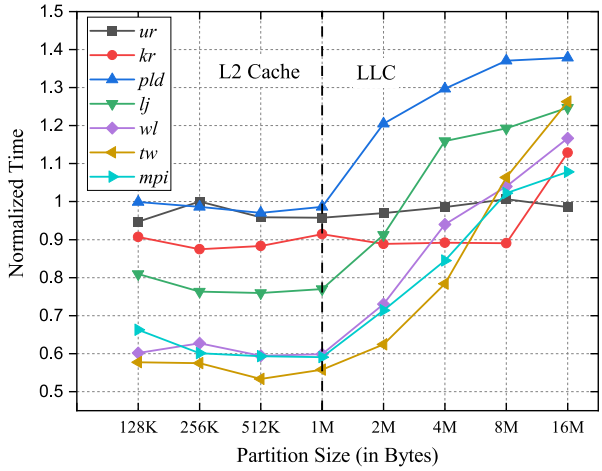



Fig. 10. The impact of partition size on the execution time. The time is normalized by the result of unordered graph.

acceleration. In this range, 1M is preferable to other options because it results in less disruption to graph structure.

Nevertheless, once the partitioning size exceeds the capacity of L2 cache, the vertices are spilled over from L2 into LLC. The spillover incurs drastic increase of cache misses as well as memory traffic. Hence, the computation is decelerated. As an exception, *kron* is not heavily influenced by the partition size ranging from 128K to 8M due to its well balanced workload (e.g., low Locality-Skew). The partition size of 16M, however, diminishes its internal balance, and thus slows the execution down.

### 4.7 Scalability Onto Vertex-Centric Paradigm

With the awareness of cache hierarchy, Corder is developed based on the recently proposed partition-centric graph processing paradigm. Nevertheless, the conventional vertex-centric paradigm is popularly adopted in numerous graph frameworks over the past decades [6], [33], [35], [49]. The reordering methods proposed for the vertex-centric framework (e.g., Ligra), such as HC, FBC and DBG, often deteriorate the performance of graph analytics when deployed in the partition-centric GPOP. Hence, in reverse, it is necessary to investigate the scalability of Corder onto the vertex-centric platforms.

Fig. 11 compares the speedups of four different frameworks. Three of the four are programmed in vertex-centric paradigm, including Ligra, Polymer and the Non-Uniform Memory Access scheduled Ligra (NUMA-Ligra). The partition-centric one, PCPM [30], is designed for efficient Pagerank algorithm. Again, we use PR as the benchmark application because of its sensitivity to vertex order. Overall, Corder demonstrates higher scalability than any other reordering methods. It consistently achieves speedups (at least no slowdown) for skewed graphs across all frameworks, except for *live* in NUMA-ligra.

*PCPM.* The partition-centric PCPM behaves similarly to GPOP. The aggravation of hot vertices promoted by HC, FBC, DBG and Sort intensified the issue of workload imbalance. Therefore, the speed of graph processing is degraded. RND leads to performance boost, when graphs exhibit inherent high imbalance (i.e., high Locality-Skew), such as *mpi* and *twitter*. It results in a slowdown, when the vertices are already well distributed in the graphs (e.g., *kron* and *pld*). Corder,

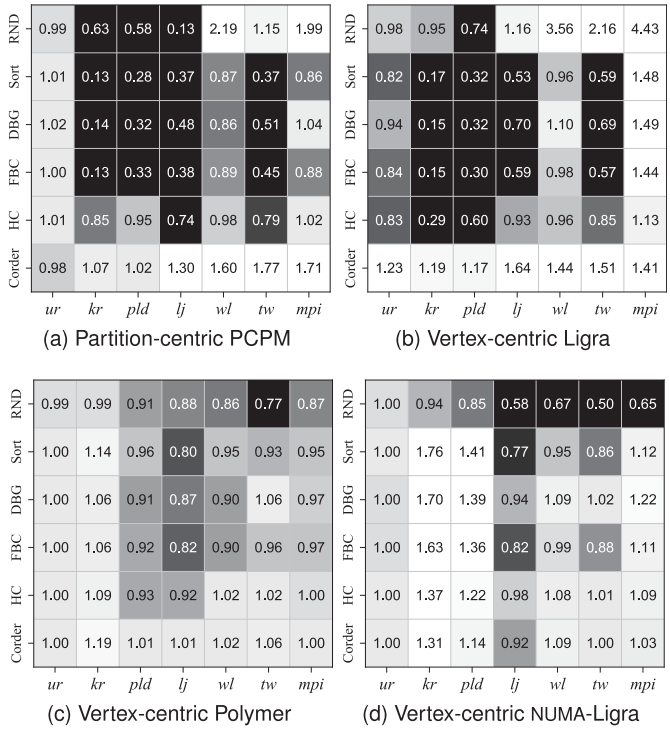



Fig. 11. Speedups in different graph processing frameworks. The lighter the color, the higher the speedup. Corder delivers consistent performance boost across all frameworks.

with the optimization of workload balance as well as graph locality, consistently achieves speedups for the skewed graphs, which is up to $1.77\times$ and on average $1.41\times$.

*Ligra*. Unexpectedly, the performance of vertex-centric Ligra oftentimes deteriorates when HC, FBC and DBG are deployed. Corder is the only reordering method that accelerates the execution of all graphs with speedup of up to $1.64\times$ (on average $1.37\times$).

A key feature of Ligra-based PageRank is the atomic update of data [6]. Atomic operations are used to ensure the correctness of multiple threads accessing a common vertex. It is an expensive synchronizing operation that stalls the execution of other threads.

As hot vertices are concentrated via graph reordering (e.g., DBG), collisions between threads are more prone to occur. Therefore, the parallel execution of multi-threads is frequently stalled by atomics, shrinking to single-thread processing. In other words, the problem of thread blocking is exacerbated by the concentration of hot vertices, which outweighs the benefit from enhanced graph locality. An abnormality is graph *mpi* with L-S = 432.43 ($\lambda = 1\%$). Since *mpi* already suffers from severe thread blocking due to its high skew, the side effect of graph reordering is relatively minor on it. Consequently, enjoying the enhancement of graph locality, *mpi* achieves substantial speedup.

*Polymer*. Polymer is implemented based on the code of Ligra and follows the vertex-centric paradigm. However, the effect of reordering techniques on Polymer is insignificant. With the awareness of NUMA, Polymer evenly partitions and allocates graph data in the local memory of NUMA nodes. This minimizes the effectiveness of graph reordering. Hence, compared with other frameworks, the performance variation of Polymer across the graphs and reordering techniques is trivial. Corder performs as the only reordering method that does not cause a slowdown.

Nevertheless, the advancement of Polymer is achieved at the price of sophisticated optimizing techniques. To utilize the NUMA feature, it redesigns the data layout, and requires graph partitioning as well as thread binding. Together, these implementations contribute to thousands of lines of codes. Moreover, such complexity produces staggering overhead. For instance, the preprocessing overhead of Polymer ranges from 7.61s of *live* to 346s of *urand*. On the other hand, Corder caters to NUMA without manipulation of underlying framework. It involves less coding effort, e.g., about hundred lines of codes. Also, it introduces minimum overhead, which is to be discussed in following section.

*NUMA-Ligra*. Though NUMA-Ligra facilitates graph processing with the support of NUMA too, it differs from Polymer. NUMA-Ligra is enabled by the command line utility on Linux-based NUMA machines. The NUMA scheduling and memory placement are automated by the NUMA policy kernel [50]. Simply with one line of command (e.g., `numactl [-options] program`), users can execute a program with specific NUMA policy.

The interleave memory allocation of NUMA-Ligra alleviates the workload imbalance incurred by graph reordering. Moreover, compared with Polymer, it ensures the graph locality enhanced by those methods. Therefore, graphs with low locality, e.g., *kron* and *pld*, acquires substantial performance boost due to the enhancement of graph locality using HC, FBC and DBG. Other graphs, which are already associated with high locality, achieve limited improvement (e.g., *wiki*, *twitter* and *mpi*) or even degradation (e.g., *live*).

TABLE 8
Reordering Overhead in Seconds

| | Mapping | | | | | Composing |
|---|---|---|---|---|---|---|
| | Corder | HC | FBC | DBG | Gorder | |
| *urand* | 0.393 | 0.912 | 1.008 | **0.091** | $>1$ h | $3.081 \pm 0.271$ |
| *kron* | 0.222 | 0.192 | 0.700 | **0.104** | $>1$ h | $2.151 \pm 0.125$ |
| *pld* | 0.154 | **0.136** | 0.435 | 0.164 | $>1$ h | $0.825 \pm 0.035$ |
| *live* | 0.017 | 0.029 | 0.058 | **0.007** | 66.8 | $0.073 \pm 0.000$ |
| *wiki* | 0.060 | 0.115 | 0.214 | **0.027** | 220.6 | $0.250 \pm 0.007$ |
| *twitter* | 0.113 | 0.114 | 0.370 | **0.049** | $>1$ h | $0.901 \pm 0.042$ |
| *mpi* | **0.153** | 0.185 | 0.574 | 0.169 | $>1$ h | $1.520 \pm 0.065$ |
| PR Iter. | 0.284 | 0.398 | 0.879 | 0.175 | $>1000$ | 1.840 |

*The overhead consists of the mapping time and composing time. For Gorder, the reordering process is terminated once it exceeds 1 hour. The last row lists the average number of iterations of PR on original graphs needed to amortize the time cost.*

## 4.8 Reordering Overhead

The practicality of a reordering method is heavily influenced by the overhead it imposes. For instance, as listed in Table 8, it costs Gorder 66.8 seconds to finish the computation on the smallest graph *live*. Nonetheless, the downstream graph applications typically ends in 10 seconds even without reordering preprocessing. Considering such difference, the overhead of Gorder is prohibitively expensive.

The reordering overhead consists of two phases: *mapping* and *composing*. In the mapping phase, a new vertex ID is assigned to every vertex in the original graph by a reordering method. Thereby, a mapping pair of vertex IDs prior and after

the reordering assignment is created, which point to the same vertex. For instance, $pair(old, new) = (v1, v100)$. In the composing phase, a new graph is constructed based on the new IDs of vertices. The composition of a graph (e.g., CSR, edge list or adjacency matrix) is decided by the data layout implemented in underlying graph frameworks. Adding up the mapping time and the composing time, we acquire the result of the reordering overhead.

To compare the overheads, we deploy the reordering algorithms proposed by prior works, including HC, FBC, DBG and Gorder, using the open-sourced code provided by the authors. The source code of HC, FBC, DBG is developed in the framework of Ligra. Hence, to provide a fair comparison, we re-implement Corder and evaluate its overhead in Ligra too.

Table 8 presents the reordering overheads in terms of the mapping time and the composing time. The mapping time reflects the efficiency of a reordering algorithm. Corder offers the fastest mapping speed for graph *mpi*. Overall, it is the second fastest reordering algorithm only behind DBG. On average, the mapping time of Corder can be amortized by 0.284 iteration of PR on the original graph. Featuring a lightweight design of parallelism, Corder achieves high efficiency.

The composing time is independent of the reordering methods, but subject to the size of a graph. Compared with the mapping phase, the composing phase charges time cost by an order of magnitude, which is 1.84 iterations on average. Hence, the overall reordering overhead, especially for applications such as BFS that finishes within 3 seconds and can be roughly amortized by 2 or 3 iterations of PR, is a non-trivial consideration.

## 5 Conclusion

Graph reordering functions as a critical prerequisite to facilitate the processing of large-scale graphs. In this work, we demonstrate that workload imbalance severely deteriorates the performance of parallel graph processing. A novel measurement approach, i.e., Locality-Skew, is proposed to estimate the imbalance in a graph. Inspired by it, we present Corder, a reordering method designed for multicore systems with awareness of the cache hierarchy. First, Corder partitions a graph into subgraphs, of which the vertex subsets fit into L2 cache. Then, Corder adopts two optimization strategies to facilitate graph processing. Within the shared memory, hot vertices are evenly distributed across subgraphs. Inside the local cache, hot vertices are concentrated to improve graph locality.

Our evaluation shows the effectiveness of Corder not only in a variety of graph datasets and graph applications, but also in diverse graph analytics frameworks. Moreover, we analyze and rank the influential factors correlating with Corder in the multicore systems, including thread behavior, cache and memory efficiencies. Amongst them, the balancing of thread execution time plays the most weighty role in the boost of graph processing.

The future work includes the integration of cache-aware reordering methodology with NUMA. Besides, we plan to investigate the difference in behaviors for directed graphs when hot vertices are classified by in-degrees instead of out-degrees.

## Acknowledgments

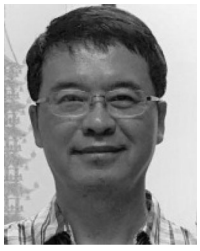
This work was supported by the National Key Research and Development Program of China under Grant 2018YFB1003505.

## References

[1] N. Kannan and S. Vishveshwara, "Identification of side-chain clusters in protein structures by a graph spectral method," *J. Mol. Biol.*, vol. 292, no. 2, pp. 441–464, 1999.
[2] P. J. Carrington, J. Scott, and S. Wasserman, *Models and Methods in Social Network Analysis*, vol. 28. Cambridge, U.K.: Cambridge Univ. Press, 2005.
[3] C. C. Noble and D. J. Cook, "Graph-based anomaly detection," in *Proc. 9th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2003, pp. 631–636.
[4] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 375–386.
[5] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 31–46.
[6] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 135–146.
[7] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna, "GPOP: A scalable cache-and memory-efficient framework for graph processing over parts," *ACM Trans. Parallel Comput.*, vol. 7, no. 1, pp. 1–24, 2020.
[8] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 599–613.
[9] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 301–316.
[10] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "MOSAIC: Processing a trillion-edge graph on a single machine," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 527–543.
[11] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Proc. IEEE Int. Symp. Workload Characterization*, 2015, pp. 56–65.
[12] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1813–1828.
[13] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 251–262, 1999.
[14] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.
[15] V. Balaji and B. Lucia, "When is graph reordering an optimization? Studying the effect of lightweight graph reordering across applications and input graphs," in *Proc. IEEE Int. Symp. Workload Characterization*, 2018, pp. 203–214.
[16] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 22–31.
[17] P. Faldu, J. Diamond, and B. Grot, "A closer look at lightweight graph reordering," in *Proc. IEEE Int. Symp. Workload Characterization*, 2019, pp. 1–13.
[18] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 293–302.
[19] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2015, *arXiv:1508.03619*.
[20] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "The graph structure in the web–analyzed on different aggregation levels," *The J. Web Sci.*, vol. 1, no. 1, pp. 33–47, Jul. 2015.
[21] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection," 2014. Accessed: Aug., 2021. [Online]. Available: http://snap.stanford.edu/data/index.html
[22] J. Kunegis, "KONECT: The koblenz network collection," in *Proc. 22nd Int. Conf. World Wide Web*, 2013, pp. 1343–1350.
[23] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 591–600.

[24] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," in *Proc. 4th Int. AAAI Conf. Weblogs Soc. Media*, 2010, pp. 10–17.
[25] Y. Lim, U. Kang, and C. Faloutsos, "SlashBurn: Graph compression and mining beyond caveman communities," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 12, pp. 3077–3089, Dec. 2014.
[26] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proc. 20th Int. Conf. World Wide Web*, 2011, pp. 587–596.
[27] D. Lasalle and G. Karypis, "Multi-threaded graph partitioning," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 225–236.
[28] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 631–643.
[29] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *Proc. IEEE Int. Symp. High Perform. Comput. Architect.*, 2020, pp. 234–248.
[30] K. Lakhotia, R. Kannan, and V. Prasanna, "Accelerating PageRank using partition-centric processing," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 427–440.
[31] P. Kumar and H. H. Huang, "G-Store: High-performance graph store for trillion-edge processing," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 830–841.
[32] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 456–471.
[33] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *Proc. 20th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2015, pp. 183–193.
[34] N. Sundaram *et al.*, "GraphMat: High performance graph analytics made productive," in *Proc. VLDB Endowment*, vol. 8, no. 11, 2015, pp. 1214–1225.
[35] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "GraphGrind: Addressing load imbalance of graph partitioning," in *Proc. Int. Conf. Supercomputing*, 2017, pp. 1–10.
[36] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 246–260, 2018.
[37] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 472–488.
[38] S. Zhou *et al.*, "Design and implementation of parallel PageRank on multicore platforms," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2017, pp. 1–6.
[39] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Stanford, CA, Tech. Rep. 1999–66, 1999.
[40] T. N. Bui and C. Jones, "Finding good approximate vertex and edge partitions is NP-hard," *Inf. Process. Lett.*, vol. 42, no. 3, pp. 153–159, 1992.
[41] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering*. Berlin, Germany: Springer, 2016, pp. 117–158.
[42] C. Jiang, F. Coenen, and M. Zito, "A survey of frequent subgraph mining algorithms," *Knowl. Eng. Rev.*, vol. 28, no. 1, pp. 75–105, 2013.
[43] S. Harenberg *et al.*, "Community detection in large-scale networks: A survey and empirical evaluation," *Wiley Interdisciplinary Rev.: Comput. Statist.*, vol. 6, no. 6, pp. 426–439, 2014.
[44] V. M. Weaver, "Linux perf_event features and overhead," in *Proc. 2nd Int. Workshop Perform. Anal. Workload Optimized Syst. FastPath*, 2013, vol. 13, Art. no. 5.
[45] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proc. 39th Int. Conf. Parallel Process. Workshops*, 2010, pp. 207–216.
[46] D. A. Spielman and S.-H. Teng, "A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning," *SIAM J. Comput.*, vol. 42, no. 1, pp. 1–26, 2013.
[47] J. Shun, F. Roosta-Khorasani, K. Fountoulakis, and M. W. Mahoney, "Parallel local graph clustering," *Proc. VLDB Endowment*, vol. 9, no. 12, 2016, pp. 1041–1052.
[48] K.-Y. Liang, S. L. Zeger, and B. Qaqish, "Multivariate regression analyses for categorical data," *J. Roy. Statist. Soc.: B (Methodol.)*, vol. 54, no. 1, pp. 3–24, 1992.
[49] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[50] A. Kleen, "A numa API for linux," *Tech. Whitepaper SUSE Labs*, 2004. Accessed: Aug., 2021. [Online]. Available: https://halobates.de/numaapi3.pdf

**YuAng Chen** received the BS degree in electronic science and technology from Huazhong University of Science and Technology, China, in 2015, and the dual MS degrees in embedded system from Eindhoven University of Technology, Netherlands and Technische Universität Berlin, Germany, in 2018. Currently, he is working toward the PhD degree in computer science at The Chinese University of Hong Kong, Shenzhen, China. His research interests include computer system and architecture, high-performance computing and graph analytics.



**Yeh-Ching Chung** received the BS degree in computer science from Chung Yuan Christian University, Taiwan, in 1983, and the MS and PhD degrees in computer and information science from Syracuse University, Syracuse, New York, in 1988 and 1992, respectively. He is currently a professor of The Chinese University of Hong Kong, Shenzhen, China. His research interests include parallel and distributed processing, cloud computing, big data, and embedded system.