When is Graph Reordering an Optimization?

Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs

Vignesh Balaji Brandon Lucia
Carnegie Mellon University
{*vigneshb,blucia*}@*andrew.cmu.edu*
`http://abstract.ece.cmu.edu`

***Abstract*—Graph processing applications are notorious for exhibiting poor cache locality due to an irregular memory access pattern. However, prior work on graph reordering has observed that the structural properties of real-world input graphs can be exploited to improve locality of graph applications. While sophisticated graph reordering techniques are effective at reducing the graph application runtime, the reordering step imposes significant overheads leading to a net increase in end-to-end execution time. The high overhead of sophisticated reordering techniques renders them inapplicable in many important use cases wherein the input graph is processed only a few times and, hence, cannot amortize the overhead of reordering.**

In this work, we identify *lightweight* reordering techniques that improve performance even after accounting for the overhead of reordering. We first conduct a detailed performance evaluation of these lightweight reordering techniques across a range of applications to identify the characteristics of applications that benefit the most from lightweight reordering. Next, we address a major impediment to the general adoption of these reordering techniques – input-dependent speedups – by linking the speedup from lightweight reordering to structural properties of the input graph. We leverage the structure dependence of speedup to propose a low-overhead mechanism to determine whether a given input graph would benefit from reordering. Using our *selective* lightweight reordering, we show maximum end-to-end speedup of up to 1.75x and never cause a slowdown beyond 0.1%.

I. Introduction

Graph processing applications form an important workload with diverse applications such as path planning, social network analysis, semi-supervised learning, data mining, and threat detection in networks. The ability to fit large graphs in the increasing main memory capacities of server-class processors has sparked an interest in single-node, shared-memory graph processing frameworks[1]. While shared-memory frameworks outperform distributed graph processing systems for graphs that fit in main memory [2], graph processing on shared-memory single-node systems is far from optimal.

The primary source of inefficiency in single-node graph processing is the poor locality in the processor cache hierarchy. Graph processing applications exhibit an irregular memory access pattern which leads to poor spatial and temporal locality in the memory access stream. Consequently, graph applications make sub-optimal use of the cache hierarchy and incur many long-latency main memory accesses. Prior work showed that the poor locality can result in graph applications spending up to 80% of the execution time stalled on main memory accesses [3].

While the irregular memory access pattern complicates efficient use of caches, the structural properties of many real-world graphs present opportunities to improve locality. The locality of graph applications can be improved with better graph data layout. The layout of graph data is influenced by the IDs assigned to vertices of the graph. Graph reordering techniques relabel vertices with different IDs by exploiting the structural properties of graphs to improve locality. Sophisticated reordering techniques provide significant performance improvements but incur extremely high overhead. For instance, we ran Gorder [4], a state of the art reordering algorithm, and observed that it took around *15 hours* to reorder a graph which reduced the run time of the Page Rank graph application from *12 seconds* to *5 seconds*. Such high overheads of graph reordering can only be justified if the same graph is expected to be processed multiple times. For example, to amortize the overhead of Gorder, the Page Rank application would have to be run nearly *7,000 times* on the reordered graph.

The main assumption of sophisticated graph reordering techniques – amortizing the high overhead of reordering over multiple executions on the reordered graphs – does not hold true in many important application scenarios. Prior work [5] noted that graph analysis might need to be performed on snapshots of a dynamically evolving graph at different instants of time (referred to as *temporal graph mining*). Examples of such temporal analyses include computing Page Ranks in dynamically changing social networks [6] or tracking changes in the diameter of an evolving graph [7]. In such application scenarios, an input graph is often processed only once which renders sophisticated graph reordering techniques to be completely ineffective. To improve locality for such application scenarios, *lightweight* graph reordering techniques are required that can provide a net performance improvement even after including the overhead of reordering.

In this work, we study lightweight graph reordering (LWR) techniques that impose low reordering overhead and can provide *end-to-end* performance improvements. However, a key limitation of lightweight reordering techniques is that the performance improvements from these techniques only materialize for a select category of applications and input graphs. Identifying the applications and input graphs that benefit from lightweight reordering and understanding their execution

[1]As observed in prior work [1], the uncompressed Facebook friend graph requires about 1.5TB memory which is within the capacity of modern servers

characteristics is the central goal of this work.

To identify the application characteristics that benefit the most from lightweight reordering, we perform a detailed study of performance improvements from three lightweight reordering techniques (with varying levels of complexity and overhead) for 11 applications across two graph benchmark suites running on 8 large input graphs. Among the applications that benefit the most from lightweight reordering, we explain the variation in performance improvement across input graphs by studying differences in the structural properties and original orderings of the input graphs. Based on our analysis, we propose a computationally-inexpensive metric to identify the input graphs that are likely to benefit from lightweight reordering and *selectively* reorder only such graphs. The selective application of lightweight reordering enabled achieving end-to-end speedups of up to 1.75x while never causing a slowdown of more than 0.1% across a set of 15 input graphs.

To summarize, we make the following contributions in this work:

- We show that lightweight reordering can effectively improve locality and provide end-to-end speedups of up to 1.75x.
- We perform a detailed characterization of the effectiveness of 3 lightweight reordering techniques across a diverse set of 11 graph applications. We expect the characterization results to be useful to both practitioners (for selecting appropriate lightweight reordering techniques for their applications) and researchers (for designing reordering algorithms tailored to an application's execution characteristics).
- We observe that the speedup from lightweight reordering depends on the structure and original ordering of input graphs. Based on this observation, we propose a low-overhead metric to predict whether an input graph is likely to receive end-to-end speedups from lightweight reordering, thereby, enabling selective reordering.

II. Background and Motivation

Graph applications have an irregular memory access pattern leading to sub-optimal performance due to poor use of on-chip caches. Many graph reordering techniques have been proposed to improve the performance of graph applications. While effective in improving locality, graph reordering techniques typically incur high overhead for reordering the graph.

A. Graph processing overview

Graph processing systems rely on a few common data structure to represent graphs, a few common access patterns for traversing graphs, and a few popular implementation strategies for optimizing graph processing implementations.

Compressed Sparse Row (CSR) Representation: Graphs are often represented in the CSR format due to the memory efficiency of the format. Figure 1 illustrates a graph as a CSR, comprising two arrays that efficiently represent a graph's edges (sorted by edge source ID). The Coordinates Array (CA) contiguously stores the neighbor IDs of each vertex in the graph. The Offsets Array (OA) stores each vertex's starting offset into the Coordinates Array. To access the neighbors of vertex i, a program accesses the i^{th} entry in OA to find vertex i's first neighbor in the CA. The number of neighbors of vertex i is equal to the difference of entries $i+1$ and i in OA. A directed graph (e.g., Figure 1), has two CSRs, one for outgoing neighbors and another for incoming neighbors.

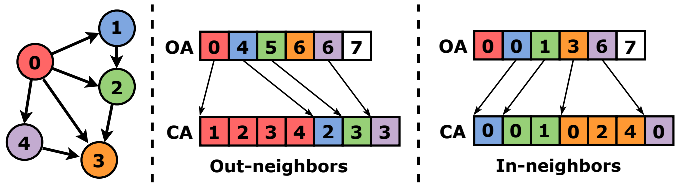


Fig. 1: **CSR representation of a directed graph**

Algorithm 1 Typical graph processing kernel

```
1: for $v$ in $G$ do
2:     for $u$ in $Neigh(v)$ do
3:         process(..., vData [$u$], ...)
```

Processing a Graph: Algorithm 1 shows a typical graph processing execution kernel. The outer loop (line 1) visits all the vertices in a graph and the inner loop (line 2) iterates over the neighbors of each vertex. The algorithm then makes irregular accesses to `vData`, which is an application-specific data structure that is indexed by the neighbor's ID, and processes its values.

Typical graph processing applications execute the above kernel iteratively, with each iteration processing a subset of vertices called the frontier (line 1). The fraction of vertices in the frontier is dependent on the application with some applications processing all the vertices every iteration (eg. Page Rank) while others process only a fraction of the vertices every iteration (eg. BFS, SSSP).

Push vs. Pull Implementations: A graph application may use *push* and/or *pull* implementations [8], [9], [10]. In a pull implementation, processing a vertex involves reading updates from the vertex's in-neighbors (i.e. traversing the incoming neighbors in line 2 of Algorithm 1), while in a push implementation, processing a vertex involves propagating updates to its out-neighbors (i.e. traversing the outgoing neighbors in line 2 of Algorithm 1). The efficiency of these implementations varies by algorithm [8] and some applications switch between the two modes in different iterations of the computation [9], [10].

B. Graph Processing Has Poor Locality

Graph processing applications have poor cache locality, which limits performance. Prior work have observed that graph applications spend a majority of the execution time stalled on long-latency memory accesses, which can contribute up to 80% of execution cycles [11]. The high number of main memory accesses are caused by the irregular access to the `vData` array shown in Algorithm 1. Figure 2 shows that the contents of the Coordinates Array (CA) defines the pattern of accesses to vData. Accesses to vData will lack spatial locality if a vertex's neighbors do not have consecutive IDs. Accesses to vData will have poor temporal locality if there is little overlap between the neighbors of vertices with consecutive IDs. While the vertex ID assignment in graphs has a significant impact in determining the locality of graph applications, vertex IDs in publicly available graph datasets are often arbitrarily assigned.

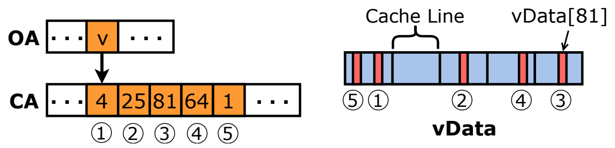


Fig. 2: **Irregular accesses to the vData array based on the contents of CA:** *The circled numbers represent the order of accessing vData elements.*

C. Reordering Graphs to Improve Locality

While the irregular memory accesses to vData lead to poor cache locality, structural properties of real-world graphs provide opportunities to optimize locality. Real-world graphs often have a power-law degree distribution [12], which causes a disproportionately large number of vData accesses to be associated with only a few highly connected vertices (i.e. "hubs"). Consequently, accesses to vData would benefit from high spatial and temporal locality if the data corresponding to hub vertices were allocated contiguously. Another common property of real-world graphs is that they exhibit a "community structure" which causes the graphs to be composed of islands of densely connected subgraphs (communities) with few connections across subgraphs [13]. Storing data for vertices in the same community contiguously improves locality of vData accesses because vertices in the same community are likely to be accessed in tandem.

Graph reordering techniques exploit the structural properties mentioned above to reassign IDs to vertices (i.e. "reorder"), thereby changing the layout of vData and other graph data structures. The power-law distribution and the community structure allow the new layout to provide improved access locality for vData. Note that relabeling the vertices does not change the structure of the graph and only affects the layout of graph data structures and locality of vData accesses.

Several reordering techniques exist in prior work [14], [15], [4], [11], [16], [17], [18], [19]. We focus on four representative reordering techniques that leverage the structural properties of graphs to maximize locality for irregular vData accesses

Gorder: Gorder [4] is a sophisticated graph reordering algorithm that relabels vertices to maximize the overlap between neighbors of vertices with consecutive IDs. Assigning consecutive IDs to vertices with many common neighbors improves reuse while accessing neighbors data across consecutively processed vertices. Finding an optimal ordering with the maximum overlapping neighbors across consecutive vertices is NP-hard [4], and we use Gorder as an exemplar for a class of heavyweight input graph reordering techniques with high computational complexity.

Rabbit: Rabbit Ordering [20] is a recently proposed *lightweight* graph reordering technique that exploits the community structure of graphs. The key idea of Rabbit Ordering is to map the hierarchically dense communities in graphs to different levels of the cache hierarchy; with the smaller, denser communities being mapped to caches closer to the processor. The authors of Rabbit Ordering use a scalable algorithm for rapid community detection allowing Rabbit Ordering to provide *end-to-end* performance improvements compared to other commonly used graph reordering techniques [20].

Hub Sorting: Frequency based clustering [11] (or "hub sorting") is another lightweight reordering technique. Hub Sorting relabels the hub vertices (defined as vertices with degree greater than average degree) in descending order of degrees, while retaining the vertex ID assignment for most non-hub vertices. Hub Sorting improves spatial and temporal locality of vData accesses for power-law graphs. Spatial locality of vData accesses is improved since assigning vertices in descending order of degree places the most highly accessed elements of vData (i.e. hub vertices) in the same cacheline. Temporal locality of vData accesses is improved since assigning the highly-accessed hub vertices a contiguous range of IDs increases the likelihood of serving requests to the high-reuse portion of vData from on-chip caches.

Hub Clustering: Hub Clustering is our variation of Hub Sorting that ensures hub vertices are assigned a contiguous range of IDs, but does not guarantee that the vertex IDs are assigned in descending order of degree. Hub Clustering improves temporal locality of vData accesses by ensuring tight packing of high-reuse hub vertex data. Hub Clustering incurs lower reordering overhead compared to Hub Sorting since it does not sort the hub vertices in descending order of degree. However, Hub Clustering provides reduced speedup compared to Hub Sorting since it misses the opportunity to improve spatial locality by placing the most frequently accessed vertices in the same cacheline. We included Hub Clustering in our evaluation to understand the tradeoff between reordering overhead and the effectiveness of the reordering technique used.

For Hub Sorting and Hub Clustering, the vertices are sorted by out-degrees for pull implementations (or pull-phase dominated implementations) and in-degrees for push implementations. The rationale for this decision is that vertices with high out-degree will be in-neighbors of many vertices, occurring frequently in the in-neighbor CA of the graph's CSR (for example, vertex 0 in Figure 1 is a hub and it constitutes majority of the graph's in-neighbor CA). Since accesses to vData are determined by the composition of the CA of a graph's CSR (Line 3 in Algorithm 1), a pull-based implementation will make a majority of the accesses to high out-degree vertices since a pull-implementation iterates over in-neighbors (line 2). Therefore, sorting vertices by out-degrees for pull implementations will increase the likelihood of frequently-accessed vertices being cached. Symmetrically, a push-based algorithm is likely to make majority of its accesses to vertices with high in-degree because the algorithm iterates over its out-neighbors (line 2). Therefore, push-implementations would benefit from ordering vertices in descending order of in-degrees.

Figure 3 shows vertex ID reassignment produced by Degree Sorting, Hub Sorting, and Hub Clustering. We omit Gorder and Rabbit Ordering because they are difficult to visualize.

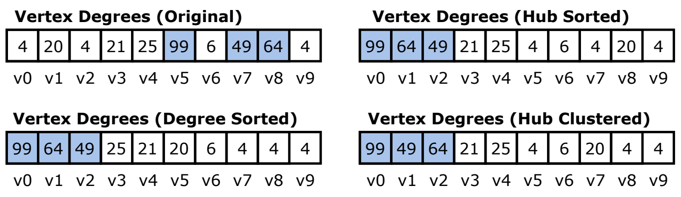

Vertex Degrees (Original)
4 20 4 21 25 99 6 49 64 4
v0 v1 v2 v3 v4 v5 v6 v7 v8 v9
Vertex Degrees (Hub Sorted)
99 64 49 21 25 4 6 4 20 4
v0 v1 v2 v3 v4 v5 v6 v7 v8 v9
Vertex Degrees (Degree Sorted)
99 64 49 25 21 20 6 4 4 4
v0 v1 v2 v3 v4 v5 v6 v7 v8 v9
Vertex Degrees (Hub Clustered)
99 49 64 21 25 4 6 20 4 4
v0 v1 v2 v3 v4 v5 v6 v7 v8 v9

Fig. 3: **Vertex ID assignments generated by different reordering techniques:** *Vertex IDs are shown below the degree of the vertex. Highly connected (hub) vertices are highlighted. Degree Sorting is shown for instructive purposes*

D. Sophisticated Reordering Techniques have High Overhead

We take Gorder [4] as a representative of the class of sophisticated graph reordering techniques that incur a high runtime overhead. We show that such techniques are only applicable in application scenarios where the reordered graph is processed multiple times to amortize the overhead of reordering. Table I shows the run times for the GAP implementation of Page Rank on five different graphs with 56 threads. "Baseline" runs Page Rank on the original graph, "Gorder" runs Page Rank after reordering with Gorder. The Gorder overhead is the time to run the authors' original Gorder implementation (which is single-threaded). Gorder consistently improves Page Rank's performance across all input graphs with a run time reduction of 35% on average and a maximum reduction of 61%.

	gplus	web	pld-arc	twitter	kron26
Run Time (baseline)	6.40s	7.84s	12.40s	21.3s	12.88s
Run Time (Gorder)	4.48s	7.77s	6.54s	13.09s	5.01s
Overhead (Gorder)	1685.9s	459.8s	7255s	25200s	53234s
#Runs to amortize ovhd	873	6477	1237	3072	6771

TABLE I: **Gorder data for Page Rank:** *Gorder improves performance but with extreme overhead.*

While Gorder is effective at reducing application run time, the overhead is extremely high. Gorder's worst case *overhead* adds a time cost equal to 1200× the original run time of the algorithm. Even if Gorder was perfectly parallelizable (our initial investigations suggest it is not), its run time on our system would be 21× the run time of the algorithm. The table also shows the minimum number of executions of Page Rank on the reordered graph required to amortize the overhead of Gorder. Across input graphs, a large number of runs are required to justify the overhead of reordering the graph using Gorder. For application scenarios where the reordered graph will be processed multiple times, reordering techniques such as Gorder might be a viable approach. However, for application scenarios such as temporal graph mining (e.g. analyzing the variation in the diameter of a dynamic graph over time) or an one-shot execution on a graph, performing sophisticated reordering (such as Gorder) is impractical.

The results highlight the need for lightweight reordering techniques that improve the performance of graph applications without imposing prohibitively high overheads. Section IV presents our study of such lightweight reordering techniques.

III. Experimental Setup

Before presenting our evaluation results, we describe the evaluation methodology used for our quantitative studies. We studied the performance improvement from lightweight reordering on a diverse set of applications spanning two graph benchmark suites using large real-world input graphs that stressed the limits of memory available in our server.

A. Evaluation Platform and Methodology

We performed all our experiments on a dual-socket server machine with two Intel Xeon E5-2660v4 processors. Each processor has 14 cores, with two hardware threads each, amounting to a total of 56 hardware execution contexts. Each processor has a 35MB Last Level Cache (LLC) and the server has 64GB of DRAM provided by eight DIMMs. All experiments were run using 56 threads and we pinned the software thread to hardware threads to avoid performance variations due to OS thread scheduling. To further reduce sources of performance variation, we also disabled the "turbo boost" DVFS features and ran all cores at the nominal frequency of 2GHz.

We ran 17 trials for each application-input pair and report the geometric mean of the 16 trials. We exclude the timing of the first trial to allow the caches to warm up. For source-dependent traversal applications (e.g. BFS, SSSP, BC, etc.), we select a source vertex belonging to the largest connected component to ensure that a significant fraction of the graph is traversed. To identify such a source, we ran 100 trials of these applications with different sources and selected the source that traversed the maximum number of edges in the graph. We also maintain a mapping between the vertex ID assignments before and after reordering to ensure that traversal applications running on the reordered graphs use the same source as the baseline execution running on the original graph [21].

B. Applications

We used 11 applications from the GAP [1] and Ligra [10] benchmark suites. All the applications were compiled using g++-6.3 with -O3 optimization level and OpenMP [22] for parallelization. We evaluated all applications in the two benchmark suites with the only exception of Triangle Counting. We exclude Triangle Counting from our evaluation because the GAP implementation already applies a common optimization of reordering vertices in decreasing order of degree.

We provide a brief description of the execution characteristics of each application and refer the reader to the original references for additional information [10], [1].

Page Rank (PR-G and PR-L): Page Rank [23] is a popular graph benchmark that iteratively refines per-vertex ranks until the sum of all ranks drops below a convergence threshold. The implementation performs pull-style accesses every iteration and processes *all* the vertices each iteration, causing many random reads to `vData`. The GAP and Ligra implementations are similar with the only significant difference being that Ligra uses a lower default convergence threshold ($1e^{-7}$ vs $1e^{-4}$) leading to longer application runtime.

Radii Estimation (Radii-L): Graph Radii estimation approximates the diameter of a graph (longest shortest path) by simultaneously performing multiple BFS traversals from different random sources. As a consequence of performing multiple BFS traversals, the application processes a large fraction of the total number of edges; visiting each vertex multiple times that leads to reuse of `vData` accesses. To avoid the cost of synchronization, the implementation processes large frontiers using pull-style accesses.

Collaborative Filtering (CF-L): Collaborative filtering is commonly used in recommender systems and has execution characteristics similar to Page Rank (processing all the vertices each iteration and performing pull-style accesses every iteration). However, CF has two distinguishing features. First, the application operates only on weighted symmetric bipartite graphs causing CF to have a unique access pattern to `vData` (discussed in Section IV-A). Second, CF has a significantly larger per-element size of `vData` compared to other applications (160B versus 4/8B) leading to a significantly larger `vData` working set size.

Components (Comp-G and Comp-L): Connected components is used to find disconnected subgraphs in a graph. Components iteratively refines the labels of each vertex until all the vertices in a connected component share the same label. The algorithm causes the application to process a large fraction of total edges during the initial iterations of the computation. The main distinction between the GAP and Ligra implementations is that GAP supports directed graphs while the Ligra implementation only processes undirected (symmetric) graphs.

Maximal Independent Set (MIS-L): MIS iteratively refines per-vertex labels to find largest independent set (set of vertices wherein no two vertices are connected) in a graph. The application has execution characteristics similar to the Ligra implementation of Components. Both applications operate on undirected graphs and perform pull-style accesses during the initial iterations when the frontier sizes are large.

Page Rank-Delta (PR-Delta-L): Page Rank Delta is a variant of Page Rank that only processes a subset of vertices for which the rank value changed beyond a δ amount. While Page Rank Delta does not process all the vertices every iteration like Page Rank, the application processes large frontiers during the initial iterations of the computation. In contrast to most other Ligra applications, the implementation does not switch between push and pull style accesses based on frontier sizes and performs push-style accesses every iteration.

SSSP-Bellman Ford (SSSP-L): The Ligra implementation of SSSP uses the Bellman Ford algorithm. Due to the work inefficient nature of the Bellman Ford algorithm, the application processes a significant fraction of total edges in the initial iterations and, hence, offer reuse in `vData` accesses. Similar to Page Rank Delta, the SSSP implementation does not switch between push and pull style accesses and always performs push-style accesses.

Betweenness Centrality (BC-G and BC-L): Betweenness Centrality finds the most central vertices in a graph by using a BFS kernel to count the number of shortest paths passing through each vertex from a source. Since the application traverses over a BFS tree of a graph, the application processes a limited fraction of total edges for most iterations.

SSSP-Delta Stepping (SSSP-G): The GAP implementation of Single Source Shortest Path problem uses the delta stepping algorithm [24] which strikes a balance between work-efficiency and parallelism. The cost of updating thread-local containers used for work-efficient scheduling of vertices reduces the fraction of the application runtime spent executing the irregular access kernel (Algorithm 1). The implementation performs push-style accesses to process vertices each iteration.

Breadth First Search (BFS-G and BFS-L): The GAP and Ligra implementations of BFS use the push-pull direction-switching optimization proposed in prior work [9] to reduce the total number of edges processed relative to a traditional implementation. Consequently, BFS processes the fewest edges among all applications; offering limited room for performance improvement from locality optimization. Additionally, the short runtime of the BFS application offers limited room for amortizing the overhead of graph reordering.

K-core Decomposition (KCore-L): KCore is an application that finds sets of vertices (called cores) with degree greater than K for different values of K. The application takes many iterations ($\approx$1000) to converge and, hence, has a long runtime. Additionally, the algorithm used for K-core computation causes the executions to spend only a small fraction ($\approx$10%) of the total run time performing irregular accesses (Algorithm 1).

	DBP	**GPL**	**PLD**	**KRON**	**TWIT**	**MPI**	**WEB**	**SD1**
Reference	[25]	[26]	[27]	[21]	[6]	[25]	[28]	[27]
$\|V\|$ **(in M)**	18.27	28.94	42.89	33.55	61.58	52.58	50.64	94.95
$\|E\|$ **(in B)**	0.172	0.462	0.623	1.047	1.468	1.963	1.93	1.937
`vData` **Sz (MB)**	146.16	231.52	343.12	268.4	498.64	420.64	405.12	759.6
CSR Sz (GB)	1.41	3.66	4.96	8.05	11.34	15.02	14.75	15.13

TABLE II: **Statistics for the evaluated input graphs:** *The size of* `vData` *for all the graphs exceeds the LLC capacity.*

C. Input graphs

We use large, real-world input graphs with power-law degree distribution that have been collected from a variety of datasets for evaluating the performance benefits from lightweight reordering. Table II lists the number of vertices, edges, the size of the `vData` array (assuming 8B element size), and the size of a CSR representation for the graph that are used for majority of our evaluation. We use the graph converters available in the GAP and Ligra benchmarks to create undirected and/or weighted versions of these graphs based on the application requirements. For Collaborative Filtering, we use the 8 largest bipartite graphs available in the Konect dataset [25]. The data show that the size of the `vData` array (which is accessed irregularly) far exceeds our system's aggregate LLC of 70 MB, making locality optimization critical to reduce long-latency DRAM accesses.

IV. Performance Improvement from Lightweight Reordering

Lightweight reordering (LWR) techniques can improve graph processing performance with low overhead. However, speedup from LWR depends on the LWR technique used, application characteristics, and properties of the input graph. This section identifies the characteristics of applications that receive end-to-end performance benefits from LWR, by studying three techniques (of varied sophistication and overhead) – Rabbit Ordering, Hub Sorting, and Hub Clustering – across the applications and graphs presented in Section III. We discuss the input-dependence of speedup from LWR in Section V.

A. Main Findings

Figure 4 plots LWR performance improvements for each application and several input graphs. For each execution (application + input graph + LWR technique), we show speedup without reordering overhead (total bar height) and end-to-end speedup accounting for the overheads (solid bar). The baseline is an execution on the input graph as originally ordered by the publishers of the graph datasets [25], [29], [27], [28]. We omit data for Rabbit Ordering on MPI, WEB, and SD1 because Rabbit Ordering exhausts our machine's 64GB of memory for these graphs. We also omit data for COMP-L, MIS-L, and KCore-L for the undirected versions of the same graphs because the applications run out of memory.

To understand variation in performance across applications, we measured the average fraction of edges processed in an iteration across applications from Ligra (shown in Table III). We weight the fraction of edges processed in an iteration by the fraction of total execution time spent in that iteration to focus on iterations that dominate runtime. The data in Table III help explain the benefit due to LWR, which we present next.

	DBP	GPL	PLD	KRON	TWIT	MPI	WEB	SD1	AVG
PR	100	100	100	100	100	100	100	100	100
Radii	55.59	71.79	69.82	87.97	75.07	72.35	41.39	43.19	64.65
CF	100	100	100	100	100	100	100	100	100
BFS	1.31	1.62	1.78	0.74	0.78	0.9	0.22	0.56	0.99
BC	22.38	22.63	28.56	43.60	28.98	25.72	9.78	15.72	24.67
SSSP	47.72	70.1	59.1	82.31	76.27	67.28	31.97	58.68	61.67
PR-δ	80.19	84.45	76.32	90.70	83.31	83.76	76.97	72.36	81.00
KCore	0.17	0.03	0.05	1.02	0.02	-	-	-	0.25
COMP	98.69	98.36	83.22	84.03	98.12	-	-	-	92.48
MIS	71.48	56.54	76.68	79.24	54.32	-	-	-	67.65

TABLE III: **Average percentage of edges processed by Ligra applications:** *A higher average percentage of edges processed corresponds to greater reuse in* `vData` *accesses. The AVG field for each application represents the average value of the metric across 8 input graphs.*

Finding 1: Lightweight reordering can provide end-to-end speedups. Figure 4 shows that the Page Rank (GAP and Ligra), Radii, Collaborative Filtering, Components, and MIS see a net speedup including LWR overheads in some cases. Table III shows that these applications all process a significant fraction of edges in each iteration. The high average percentage of edges processed leads to significant reuse in `vData` accesses and offers a higher room for locality improvement from LWR.

Finding 2: Hub Sorting is a good balance of effectiveness and reordering overhead. The data for Page Rank and Radii reveal a tension between LWR effectiveness and the overhead of graph reordering. Compared to Hub Clustering, Hub Sorting yields higher speedup (excluding overhead) than Hub Sorting for the PLD, TWIT, KRON, and SD1 graphs. The higher speedup is due to reordering frequently accessed vertices in decreasing degree order, improving locality by placing the most frequently accessed `vData` elements in the same cache line. Hub Clustering misses this opportunity for spatial locality because it does not store vertices in decreasing degree order. In contrast, Hub Sorting incurs a higher overhead than Hub Clustering (i.e., the shaded portion in each bar) reducing the difference in the net speedup between the two techniques, especially for the applications with short runtimes (Radii and GAP's Page Rank).

The data for Rabbit Ordering reveal a surprising trend: the less sophisticated Hub Sorting algorithm has higher speedup than the more sophisticated Rabbit Ordering algorithm. Ignoring overhead, Rabbit Ordering does not consistently outperform Hub Sorting because the authors of Rabbit Ordering use heuristics to parallelize the reordering algorithm [20]. After accounting for reordering overhead, Hub Sorting consistently outperforms Rabbit Ordering. The data suggest that for Page Rank and Radii, Hub Sorting is an effective middle ground, improving performance with low overhead.

Finding 3: Hub Sorting is a poor fit for symmetric bipartite graphs. Figure 4 shows that Collaborative Filtering has different performance characteristics than Page Rank despite having similar execution characteristics. While Rabbit Ordering consistently improves performance, Hub Clustering has the best net speedup after accounting for overhead. Surprisingly, Hub Sorting causes *slowdown* for the EDIT, LIVEJ, and TRACK graphs, even after ignoring reordering overhead.

Collaborative Filtering is different because its input graphs are symmetric and bipartite (i.e., vertices fall into two parts A or B and no pair of nodes in the same part are connected). Neighbors of a vertex $u \in A$ are in part B and vice versa. Contiguously ordering vertices in a part offers temporal locality because irregular `vData` accesses are restricted to one part at a time. The base ordering of our bipartite graphs had their parts originally laid out contiguously. Naively Hub Sorting mixes vertices from different parts, leading to the slowdown.

We studied the part-wise ordering effect by visualizing the part number (A or B) for each vertex in the base ordering and in the ordering produced by Hub Sorting and Hub Clustering (Figure 5) for two symmetric bipartite graphs. The data show that Hub Clustering is better at preserving part-wise locality compared to Hub Sorting and, hence, provides better performance.

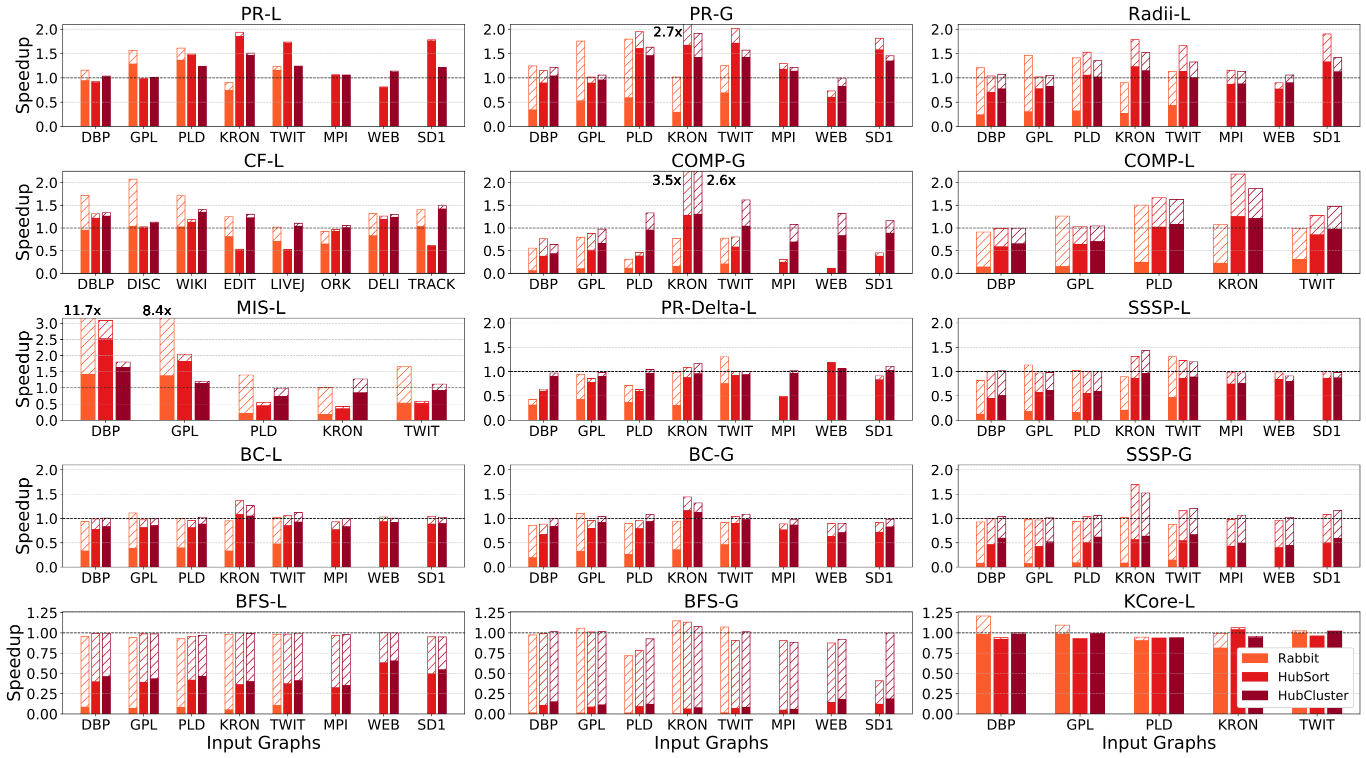
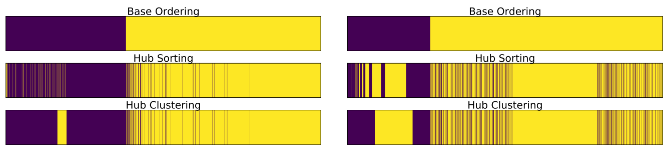


Fig. 4: **Speedup after lightweight reordering:** *Data are normalized to run time with the original graph labeling. The total bar height is speedup* ***without*** *accounting for the overhead of lightweight reordering. The upper, hashed part of the bar represents* ***the overhead*** *imposed by lightweight reordering. The filled, lower bar segment is the* ***net performance improvement*** *accounting for overhead. The benchmark suites are differentiated using a suffix (G/L).*



(a) LIVEJ (b) DBLP

Fig. 5: **Reorderings of a symmetric bipartite graph by Hub Sorting and Hub Clustering:** *The two colors represent the parts of the graph. Hub Sorting produces a vertex order wherein vertices from different parts are assigned consecutive vertex IDs whereas Hub Clustering produces an ordering where vertices belonging to the same part are often assigned consecutive IDs.*

		DBP	**GPL**	**PLD**	**KRON**	**TWIT**
Comp-G	**Rabbit**	2.39x	2.0x	5.23x	1.33x	1.47x
	HubSort	1.36x	1.0x	2.09x	0.67x	0.9x
	HubCluster	1.81x	1.0x	1.05x	0.67x	0.88x
Comp-L	**Rabbit**	1.5x	1.25x	1.27x	1.0x	0.99x
	HubSort	1.25x	1.0x	1.0x	0.67x	0.93x
	HubCluster	1.25x	1.0x	1.0x	0.83x	0.94x
MIS-L	**Rabbit**	0.3x	0.56x	0.56x	0.96x	0.52x
	HubSort	0.69x	0.56x	0.79x	2.27x	1.01x
	HubCluster	0.85x	0.85x	0.98x	1.19x	1.02x

TABLE IV: **Impact of LWR on iterations until convergence:** *Values greater than 1 indicate delayed convergence compared to baseline execution on the original graph.*

Finding 4: LWR can affect convergence rate. For Components (GAP and Ligra) and MIS, performance with LWR varies due to a change in the number of iterations to convergence. Convergence varies because in these algorithms the total amount of work performed per iteration depends on the *vertex ID assignment*. Note that reordering does not affect correctness of these applications. Table IV shows the increase in iterations to convergence for each LWR technique. Speedups in Figure 4 track the variation in iterations to convergence in most cases. However, the increase in iterations to convergence does not vary consistently with application, input graph, or LWR technique used.

Finding 5: Push-mode applications benefit less from LWR. SSSP-Bellman Ford (Ligra) and Page Rank Delta applications do not speed up with LWR despite processing a large fraction of edges per iteration (Table III). The distinguishing feature of these two Ligra applications is that they do not use the push-pull direction optimization, instead using push-mode accesses only. While laying out the most frequently accessed vertices together improves performance for push-mode applications, doing so may also increases the likelihood of false sharing, which degrades performance. False sharing affects Page Rank Delta on the DBP, GPL, and MPI graphs. Consequently, Rabbit Ordering and Hub Sorting cause slowdowns even *without* the reordering overhead. SSSP-Bellman Ford has better performance than Page Rank Delta because it is optimized to use Test&Test&Set [30] operations, which reduces false sharing.

To help understand the loss due to false sharing from LWR in these push-style applications, we evaluated speedup from LWR after modifying the applications to use the push-pull optimization. Table V shows speedups (without overhead) from LWR for these push-pull versions of Page Rank Delta and SSSP-Bellman Ford. The results shows that all three LWR techniques provide greater speedup when the two applications use the push-pull optimization compared to when the applications perform push-style accesses throughout the execution (Figure 4). Although the push-pull implementations of Page Rank Delta and SSSP-Bellman Ford are slower than the push-style implementations, the results of Table V illustrate that push-style accesses reduce the performance benefits of LWR, even in applications that process a large fraction of edges per iteration (Table III).

		DBP	GPL	PLD	KRON	TWIT
PR-δ-L	**Rabbit**	1.11x	1.53x	1.53x	0.92x	1.26x
	HubSort	0.94x	0.99x	1.43x	1.77x	1.77x
	HubCluster	1.06x	1.01x	1.24x	1.46x	1.27x
SSSP-L	**Rabbit**	0.87x	1.36x	1.2x	0.95x	0.97x
	HubSort	1.02x	1.14x	1.58x	2.0x	1.4x
	HubCluster	1.14x	1.07x	1.47x	1.58x	1.4x

TABLE V: **Speedups from LWR for push-pull implementations:** *LWR techniques provide greater performance improvements for applications that perform pull-style accesses while processing large frontiers.*

Finding 6: Applications that process few edges per iteration do not benefit from LWR. Applications that process a small fraction of edges per iteration (BC, BFS, and KCore) see little benefit from LWR, even after ignoring reordering overhead. Figure 4 shows that these applications consistently see no speedup from LWR even without accounting for the reordering overhead. The KRON graph is a notable exception, seeing appreciable benefit due to its *flat* graph structure[2] offering reuse of `vData` accesses. However, the real-world graphs in our dataset do not share KRON's flat structure and, hence, do not benefit from LWR.

The data for BFS and KCore further highlight the lack of benefit with few edges processed per iteration. Table III shows that the BFS and KCore application process the fewest edges per iteration of all the applications we evaluated. The small fraction of edges processed per iteration leads to limited reuse in `vData` access and offers little room for improvement from LWR.

B. When is LWR a suitable optimization?

We summarize our analysis across LWR techniques, applications, and input graphs by listing recommendations for the lightweight reordering technique suitable for different categories of applications.

Takeaway 1: Applications like Page Rank and Radii that process a large fraction of edges in each iteration in the pull-mode are most amenable to lightweight reordering techniques.

[2] Flat means the BFS tree of KRON is shallow, with a majority of vertices in a few levels of the tree

Takeaway 2: Existing lightweight reordering techniques are inappropriate for symmetric bipartite graphs (as in CF) unless modified to store vertices in each part contiguously in memory.

Takeaway 3: In some cases (e.g., Components and MIS) lightweight reordering changes the number of iterations to convergence, revealing an opportunity for future techniques leveraging vertex ordering to speed convergence.

Takeaway 4: Applications that are push-style (e.g., Page Rank Delta and SSSP-Bellman Ford) or process a few edges per iteration (e.g., BC, SSSP-Delta Stepping, BFS, and KCore) do not benefit from lightweight reordering because of false-sharing and limited reuse in `vData` accesses respectively. The ineffectiveness of lightweight reordering in these applications is *not* due to the overhead of reordering.

Takeaway 5: When Hub Sorting is effective (e.g., Page Rank and Radii), its benefit is input graph-dependent (Figure 4), sometimes providing no speedup (e.g., DBP, GPL, MPI, WEB) and instead causing a net slowdown due to its overhead. The next section studies the characteristics of graphs for which lightweight reordering provides end-to-end speedups.

V. Selective Lightweight Graph Reordering

Hub Sorting is an effective lightweight reordering technique that provide end-to-end performance improvement for applications like Page Rank and Radii. However, the speedup from lightweight reordering depends on the input graph. A system should not unconditionally reorder its input graph because reordering sometimes causes a slowdown. This section shows that the *graph structure* and the *original graph ordering* determine the speedup of reordering. Next, we propose a low-overhead metric to identify the properties of the input graph critical for achieving speedup from reordering and show that the metric enables *selective* application of Hub Sorting.

A. Input-dependent speedup from Hub Sorting

The variation in a graph's structure and its original vertex ID assignment explains the difference in speedups from Hub Sorting across input graphs. Assigning hub vertices a contiguous range of IDs ensures that the frequently accessed elements of `vData` (i.e., hubs) are packed closely, spanning a small number of cache lines. The hub vertices in a graph are connected to a significant fraction of the graph with the hub vertices accounting for 80% of total edges across the graphs shown in Table II. Consequently, accesses to cache lines containing hubs' elements in `vData` are frequent and lines containing tightly-packed hubs are likely to be frequently re-used. These tightly-packed hubs' cache lines are also likely to remain resident in the Last Level Cache (LLC), improving locality. Moreover, sorting vertices by decreasing vertex degree puts the most frequently accessed vertices in the same cache line improving spatial locality.

In order to benefit from Hub Sorting, an input graph must be skewed and its hubs must not already be tightly-packed before reordering. In a skewed input graph, a few vertices have a disproportionately higher degree than all other vertices. Skewed graphs allow Hub Sorting to pack the few hubs into even fewer cache lines, increasing the likelihood that `vData` accesses will hit in the LLC because the hubs' cache lines will remain cached. Additionally, to benefit from Hub Sorting, the original layout of hub vertices must also be sufficiently sparse in memory such that multiple hubs are unlikely to reside in the same cache line. If hub vertices are originally tightly-packed, Hub Sorting is ineffective because accesses to the hubs will already have good locality. In contrast, graphs that originally have sparsely distributed hub vertices suffer from poor temporal and spatial locality. For such graphs, Hub Sorting provides performance gains by reordering the hub vertices such that the highly accessed `vData` elements span fewer cache lines.

B. Packing Factor

To identify whether an input graph will benefit from Hub Sorting, we develop *Packing Factor*, a metric that quantifies graph skew and the sparsity of hub vertices[3]. Packing Factor directly computes the decrease in sparsity of hubs after Hub Sorting. To compute Packing Factor, we compute the original graph's *hub working set*, which is the number of distinct cache lines containing hub vertices. Packing Factor is the ratio of the original graph's hub working set to the minimum number of cache lines in which the graph's hubs can fit, based solely on cache line capacity. If the original graph's hub working set is much larger than the minimum number of lines required for the hub vertices then Packing Factor is high and Hub Sorting is likely to provide a large benefit by tightly packing the hubs. Algorithm 2 shows the steps involved in computing the hub working set of the original graph (Lines 4-11) and after performing Hub Sorting (Line 12)

Algorithm 2 Computing the Packing Factor of a graph

```
 1: procedure COMPUTEPACKINGFACTOR($G$)
 2:     $numHubs \leftarrow 0$
 3:     $hubWSet\_Original \leftarrow 0$
 4:     for $CacheLine$ in $vDataLines$ do
 5:         $containsHub \leftarrow False$
 6:         for $vtx$ in $CacheLine$ do
 7:             if ISHUB($vtx$) then
 8:                 $numHubs$ += 1
 9:                 $containsHub \leftarrow True$
10:         if $containsHub = True$ then
11:             $hubWSet\_Original$ += 1
12:     $hubWSet\_Sorted \leftarrow$ CEIL($numHubs / VtxPerLine$)
13:     $PackingFactor \leftarrow hubWSet\_Original / hubWSet\_Sorted$
    return $PackingFactor$
```

To understand the relationship between speedup from Hub Sorting and Packing Factor, we measured speedup of Page Rank (GAP and Ligra) and Radii on the 8 input graphs from the main evaluation (Figure 4) and 7 additional input graphs from Konect [25]. Figure 6 shows the Hub Sorting speedup

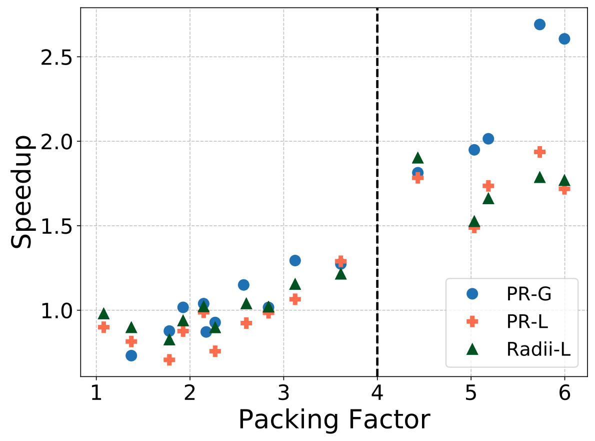


Fig. 6: **Relation between speedup from Hub Sorting and packing factor of input graph:** *Each point is a speedup of an application executing on Hub Sorted graph compared to the original graph. Different applications are indicated with different colors/markers. Hub Sorting provides significant speedup for executions on graphs with high Packing Factor.*

(excluding reordering overhead) and Packing Factor of the input graph for the three applications on 15 graphs. The data shows a strong correlation (r = 0.9) between speedup from Hub Sorting and Packing Factor of a graph.

The data in Figure 6 show that a graph's Packing Factor is a useful predictor of Hub Sorting's speedup. We empirically observe that graphs with a Packing Factor less than 4 do not experience a significant speedup from Hub Sorting (maximum speedup of 1.25x). For such graphs, speedup is likely to be negated by the overhead of Hub Sorting, leading to a net slowdown. Based on these data, we conclude that a system should *selectively* perform Hub Sorting on graphs with packing factor greater than the threshold value of 4 only. Selective Hub Sorting yields speedup for graphs with high Packing Factor and avoids degrading performance for other graphs. We evaluate such a system in the next subsection.

C. Selective Hub Sorting using Packing Factor

A graph's Packing Factor predicts whether Hub Sorting will yield a net speedup. Computing Packing Factor (Algorithm 2) imposes a low-overhead since it involves a highly-parallelizable scan of vertex degrees. Figure 7 shows the net speedup (including Hub Sorting overhead) for a system that unconditionally reorders a graph compared to a system that only selectively reorders a graph if the Packing Factor of the graph is a higher than our empirical threshold of 4. Selective reordering preserves the end-to-end speedup from unconditional reordering for graphs with high Packing Factor while avoiding slowdowns for graphs with low Packing Factor. Computing Packing Factor imposes negligible overhead [4], making selective reordering a practical alternative to unconditional Hub Sorting.

[3] We have open-sourced the code for Packing Factor computation and other reordering techniques at *https://github.com/CMUAbstract/Graph-Reordering-IISWC18*

[4] Across input graphs, computing the Packing Factor comprised at most 0.1% of the runtime on the original graph

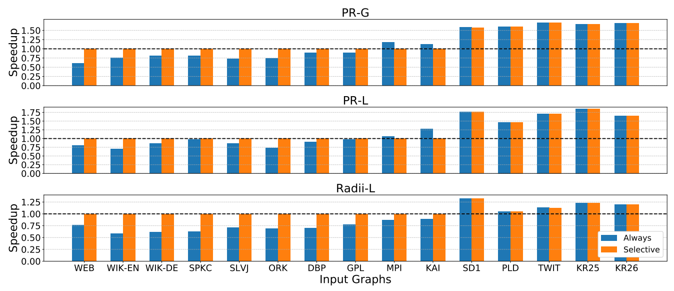


Fig. 7: **End-to-end speedup from selective Hub Sorting:** *Input graphs have been arranged in increasing order of Packing Factor. Selective application of Hub Sorting based on Packing Factor provides significant speedups on graphs with high Packing Factor while avoiding slowdowns on graphs with low Packing Factor.*

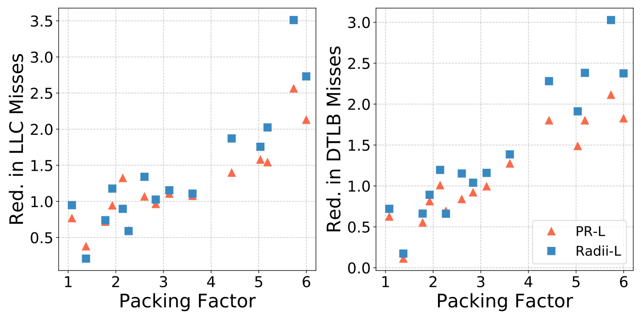


Fig. 8: **Reduction in LLC and DTLB misses due to Hub Sorting:** *Hub Sorting provides greater reduction in LLC misses and DTLB load misses for graphs with high Packing Factor.*

D. *Characterization of speedup from Hub Sorting*

We used native hardware performance counters [31] to verify that Hub Sorting improves performance by improving locality. We measured the reduction in Last Level Cache (LLC) load misses and the reduction in Data TLB load misses leading to a page walk. For these tests, we disabled hyperthreading and ran with only one thread per core (i.e., 28 threads) due to limitations of the performance counter infrastructure [32]. Figure 8 shows the reduction in LLC load misses and DTLB load misses compared to Packing Factor of input graphs for the two applications from Ligra. The data show that graphs with high Packing Factor get significant reduction in LLC misses and DTLB load misses from Hub Sorting. The linear relation between LLC and DTLB load miss reduction and Packing Factor is characteristic of the linear relation between speedup and Packing Factor in Figure 6. The data suggest that the speedup from lightweight reordering are due to reduction in LLC misses (fewer long-latency DRAM accesses) and a reduction in DTLB misses (fewer expensive page table walks).

VI. Related Work

We divide the prior research related to this work into five categories - graph reordering, cache blocking, vertex scheduling, graph partitioning, and in-memory graph processing.

Graph Reordering: There has been extensive research in developing graph reordering techniques of varying levels of effectiveness and sophistication. Sophisticated graph reordering techniques such as Gorder [4], ReCALL [33], Layered Label Propagation (LLP) [34], Nested Dissection [18], SlashBurn [35] provide significant speedups to the application but incur extremely high overheads. The high overhead of these techniques are justified only in the cases where the *same* input graph is expected to be processed multiple times. In contrast to such high-overhead reordering techniques, recent graph reordering proposals have focused on keeping the overhead of reordering low. Karantis et. al. [19] proposed a parallel implementations of common graph reordering techniques – Reverse Cuthill-McKee (RCM) and Sloan – to reduce reordering overheads. While parallelization improved the performance of reordering by more than 5x, the authors report an end-to-end speedup of 1.5x when performing 100 Sparse Matrix Vector (SpMV) iterations. Rabbit Ordering [20], which was studied in this work, was shown to provide better end-to-end performance improvements compared to parallel RCM. These research efforts support the need for effective lightweight reordering techniques to support application use cases where the assumption of amortizing high reordering overhead across multiple trials is not guaranteed.

Cache blocking: Cache blocking is an alternate technique to improve locality of graph processing applications. Zhang et. al. [3] recently proposed CSR segmenting – a technique to improve temporal locality of `vData` accesses by breaking the original graph into subgraphs that reduce the irregularity of `vData` accesses. The computation from each subgraph are buffered and later merged to produce the final result. Similar approaches were used in prior work aiming to exploit reuse at the LLC [36], [37]. Recent proposals [38], [39] have extended the idea of partitioning the graph and applied it to partitioning data transfers between vertices in Sparse Matrix multiplying Dense Vector (SpMV) application such as Page Rank. While blocking based techniques are effective in improving application performance, they require modifying the application unlike graph reordering techniques.

Vertex scheduling: Graph reordering techniques improves locality by optimizing the layout of graph data structures. The locality of graph applications can also be improved by changing the order of processing vertices. Prior work [40], [2], [11] have shown that traversing the edges of a graph along a Hilbert curve can create locality in the both the source vertex read from and the destination vertex written to. However, a key challenge with these techniques is that they can complicate parallelization [38], [3]. CGS [41] is an architectural proposal that improves locality of irregular accesses by scheduling vertices in a cache-friendly manner using a specialized cache engine at the LLC. Graph reordering is primarily a data layout optimization and, hence, is complementary to vertex scheduling. The low overhead lightweight reordering techniques studied in this work should be able to provide multiplicative performance improvements when combined with vertex scheduling.

Graph Partitioning: Graph partitioning is commonly applied in the context of distributed graph processing. The goal of graph partitioning is to reduce inter-node communication by creating graph partitions with minimal number of links between partitions [42], [43], [44], [45]. Graph partitioning has similarities to reordering since the partitioning problem can be viewed as trying to maximize locality within a node. Additionally, sophisticated graph partitioning techniques impose significant time and space overheads [14]. Prior work [46], [47] have proposed lightweight graph partitioning techniques that allow applying the benefits of graph partitioning to streaming distributed graphs. Research efforts in lightweight partitioning further highlight the importance and need for graph preprocessing steps to incur low overhead.

Graph partitioning has also been studied in the context of locality-optimization for single-node shared-memory systems. Sun et. al. [48] proposed using partitioning to improve temporal locality by assigning all the in-neighbors of every vertex into a separate partition. The proposed technique requires modifications to the algorithms and the data structures to handle a large number of partitions. GridGraph [49], X-stream [50], Graphchi [51], and Turbograph [52] use forms of partitioning to optimize the disk to memory boundary. While these systems allow scaling graph processing to larger graphs beyond the main memory capacity, prior work [3] has shown that for graph which fit in memory, applying the optimizations directed at reducing disk accesses cannot be applied to optimize random accesses to main memory. Grace [53] is a shared-memory graph management system that showed that partitioning the graph provided greater opportunity for reordering algorithms to optimize locality. While graph partitioning techniques share similarity to reordering, they often require changes to graph data structures and computations unlike graph reordering.

In-memory Graph processing: The ability to fit increasingly large graphs in the main memory of server class processors has spurred research in in-memory graph processing with many graph framework targeting single-node systems [21], [10], [54], [37], [55], [56], [57], [58]. Prior work [59], [1], [36], [60] surveyed the bottlenecks of popular in-memory graph processing frameworks and identified main memory accesses as the main bottleneck. The locality-optimizing lightweight reordering techniques studied in this work were motivated by this observation.

VII. Conclusions

We studied Lightweight Graph Reordering (lightweight reordering) techniques as a means to provide end-to-end speedups for graph processing applications. We address the important question of when to apply lightweight reordering by performing a detailed characterization of performance benefits from lightweight reordering across applications and input graphs. We expect the takeaways listed in Section IV to be useful for deciding whether lightweight reordering is suitable for a particular application. For applications where lightweight reordering is deemed suitable, the Packing Factor metric can enable selectively reordering the input graph. Our evaluations show that *selective* lightweight reordering techniques can offer end-to-end performance benefits while ensuring that applications never experience a slowdown.

Acknowledgements

Thanks to the anonymous reviewers for the insightful feedback, to Aamer Jaleel for feedback on the manuscript, and to Nathan Beckmann for early feedback on the ideas. This work is supported in part by National Science Foundation grant XPS-1629196 and the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS).

References

[1] S. Beamer, K. Asanovic, and D. Patterson, “Locality exists in graph processing: Workload characterization on an ivy bridge server,” in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pp. 56–65, IEEE, 2015.

[2] F. McSherry, M. Isard, and D. G. Murray, “Scalability! but at what cost?,” in *HotOS*, 2015.

[3] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, “Making caches work for graph analytics,” in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 293–302, Dec 2017.

[4] H. Wei, J. X. Yu, C. Lu, and X. Lin, “Speedup graph processing by graph ordering,” in *Proceedings of the 2016 International Conference on Management of Data*, pp. 1813–1828, ACM, 2016.

[5] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, “Chronos: a graph engine for temporal graph analysis,” in *Proceedings of the Ninth European Conference on Computer Systems*, p. 1, ACM, 2014.

[6] H. Kwak, C. Lee, H. Park, and S. Moon, “What is Twitter, a social network or a news media?,” in *WWW ’10: Proceedings of the 19th international conference on World wide web*, (New York, NY, USA), pp. 591–600, ACM, 2010.

[7] J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution: Densification and shrinking diameters,” *ACM Trans. Knowl. Discov. Data*, vol. 1, Mar. 2007.

[8] 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 *26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC17)*, 2017.

[9] S. Beamer, K. Asanović, and D. Patterson, “Direction-optimizing breadth-first search,” *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[10] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory,” in *ACM Sigplan Notices*, vol. 48, pp. 135–146, ACM, 2013.

[11] Y. Zhang, V. Kiriansky, C. Mendis, M. Zaharia, and S. Amarasinghe, “Optimizing cache performance for graph analytics,” *arXiv preprint arXiv:1608.01362*, 2016.

[12] A.-L. Barabási, “Scale-free networks: a decade and beyond,” *science*, vol. 325, no. 5939, pp. 412–413, 2009.

[13] M. Girvan and M. E. Newman, “Community structure in social and biological networks,” *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[14] G. Karypis and V. Kumar, “Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0,” 1995.

[15] J. Petit, “Experiments on the minimum linear arrangement problem,” *Journal of Experimental Algorithmics (JEA)*, vol. 8, pp. 2–3, 2003.

[16] E. Cuthill and J. McKee, “Reducing the bandwidth of sparse symmetric matrices,” in *Proceedings of the 1969 24th national conference*, pp. 157–172, ACM, 1969.

[17] J. Banerjee, W. Kim, S.-J. Kim, and J. F. Garza, “Clustering a dag for cad databases,” *IEEE Transactions on Software Engineering*, vol. 14, no. 11, pp. 1684–1699, 1988.

[18] D. Lasalle and G. Karypis, “Multi-threaded graph partitioning,” in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS ’13, (Washington, DC, USA), pp. 225–236, IEEE Computer Society, 2013.

[19] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali, “Parallelization of reordering algorithms for bandwidth and wavefront reduction,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 921–932, IEEE Press, 2014.

[20] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pp. 22–31, IEEE, 2016.

[21] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[22] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[23] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, (Amsterdam, The Netherlands, The Netherlands), pp. 107–117, Elsevier Science Publishers B. V., 1998.

[24] U. Meyer and P. Sanders, "Delta-stepping: A parallel single source shortest path algorithm," in *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, (London, UK, UK), pp. 393–404, Springer-Verlag, 1998.

[25] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*, pp. 1343–1350, ACM, 2013.

[26] N. Z. Gong and W. Xu, "Reciprocal versus parasocial relationships in online social networks," *Social Network Analysis and Mining*, vol. 4, no. 1, p. 184, 2014.

[27] "Web Data Commons hyperlink graph 2014." http://webdatacommons.org/hyperlinkgraph/. Accessed: 2017-11-21.

[28] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[29] J. Leskovec, A. Krevl, and S. Datasets, "Stanford large network dataset collection, 2011," *URL: http://snap. stanford. edu/data/index. html*, 2014.

[30] L. Rudolph and Z. Segall, "Dynamic decentralized cache schemes for mimd parallel processors," *SIGARCH Comput. Archit. News*, vol. 12, pp. 340–347, Jan. 1984.

[31] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pp. 207–216, IEEE, 2010.

[32] "Likwid Marker API with C/C++." https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerC#problems. Accessed: 2018-05-25.

[33] K. Lakhotia, S. Singapura, R. Kannan, and V. Prasanna, "Recall: Reordered cache aware locality based graph processing," in *High Performance Computing (HiPC), 2017 IEEE 24th International Conference on*, pp. 273–282, IEEE, 2017.

[34] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World wide web*, pp. 587–596, ACM, 2011.

[35] Y. Lim, U. Kang, and C. Faloutsos, "Slashburn: Graph compression and mining beyond caveman communities," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, pp. 3077–3089, 2014.

[36] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–13, IEEE, 2016.

[37] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[38] S. Beamer, K. Asanović, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pp. 820–831, IEEE, 2017.

[39] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, (New York, NY, USA), pp. 37:1–37:12, ACM, 2016.

[40] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455, ACM, 2013.

[41] A. Mukkara, N. Beckmann, and D. Sanchez, "Cache-guided scheduling: Exploiting caches to maximize locality in graph processing,"

[42] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[43] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, (New York, NY, USA), pp. 22:1–22:12, ACM, 2013.

[44] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs.," in *OSDI*, vol. 12, p. 2, 2012.

[45] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[46] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1222–1230, ACM, 2012.

[47] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, (New York, NY, USA), pp. 333–342, ACM, 2014.

[48] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Accelerating graph analytics by utilising the memory locality of graph partitioning," in *Parallel Processing (ICPP), 2017 46th International Conference on*, pp. 181–190, IEEE, 2017.

[49] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning.," in *USENIX Annual Technical Conference*, pp. 375–386, 2015.

[50] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 472–488, ACM, 2013.

[51] A. Kyrola, G. E. Blelloch, C. Guestrin, *et al.*, "Graphchi: Large-scale graph computation on just a pc.," in *OSDI*, vol. 12, pp. 31–46, 2012.

[52] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 77–85, ACM, 2013.

[53] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan, "Managing large graphs on multi-cores with graph awareness," 2012.

[54] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *Data Compression Conference (DCC), 2015*, pp. 403–412, IEEE, 2015.

[55] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey, "Graphpad: Optimized graph primitives for parallel and distributed platforms," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pp. 313–322, IEEE, 2016.

[56] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[57] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, *et al.*, "The tao of parallelism in algorithms," in *ACM Sigplan Notices*, vol. 46, pp. 12–25, ACM, 2011.

[58] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," *SIGPLAN Not.*, vol. 50, pp. 183–193, Jan. 2015.

[59] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 979–990, ACM, 2014.

[60] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 166–177, IEEE Press, 2016.