Optimizing Cache Performance for Graph Analytics

Yunming Zhang, Vladimir Kiriansky, Charith Mendis
Matei Zaharia, Saman Amarasinghe
MIT CSAIL, Cambridge, MA, USA
{*yunming*, *vlk*, *charithm*, *saman*, *matei*}@csail.mit.edu

Abstract

Modern hardware systems are heavily underutilized when running large-scale graph applications. While many in-memory graph frameworks have made substantial progress in optimizing these applications, we show that it is still possible to achieve up to 4x speedups over the fastest frameworks by greatly improving cache utilization. Previous systems have applied out-of-core processing techniques from the memory/disk boundary to the cache/DRAM boundary. However, we find that blindly applying such techniques is ineffective because of the much smaller performance gap between DRAM and cache. We present two techniques that take advantage of the cache with minimal or no instruction overhead. The first, frequency based clustering, groups together frequently accessed vertices to improve the utilization of each cache line with no runtime overhead. The second, CSR segmenting, partitions the graph to restrict all random accesses to the cache, makes all DRAM access sequential, and merges partition results using a very low overhead cache-aware merge. Both techniques can be easily implemented on top of optimized graph frameworks. Our techniques combined give speedups of up to 4× for PageRank, Label Propagation and Collaborative Filtering, and 2× for Betweenness Centrality over the best published results.

1. Introduction

High performance graph analytics has received considerable research attention, leading to a series of optimized frameworks such as GraphLab [16], Ligra [24], Galois [20] and GraphMat [27]. Increasingly, many of these frameworks target a single multicore machine, because a single machine has the smallest communication overhead and memories have grown to the point where many graphs can fit on one server [18, 24].

Given the effort in this field, it is natural to ask whether current frameworks are close to hardware limits. Perhaps surprisingly, we find that they are not. We present several optimizations that improve performance by 2–4× over the best published results for common applications.

The core problem is that graph applications have poor cache utilization. They do very little computation per byte accessed, and a large fraction of their memory requests are random. Random accesses to a working set that does not fit in cache make the entire cache hardware subsystem ineffective. Without effective use of the cache to mitigate the processor-DRAM gap, CPUs are stalled on high-latency random accesses to DRAM. Indeed, we find that today's fastest frameworks spend 60–80% of their cycles stalled on memory access.

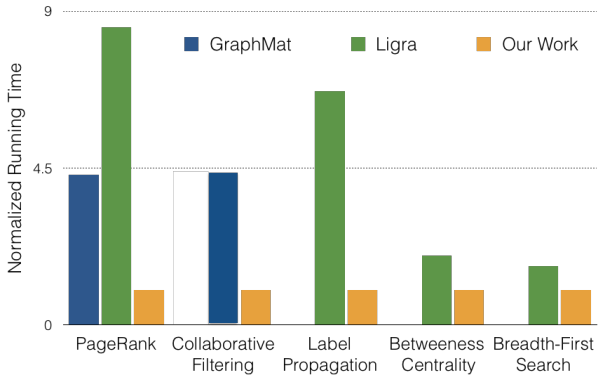


Figure 1: Running time with our techniques vs. best published implementations in current frameworks on RMAT27.

The fastest in-memory frameworks, such as GraphMat [27] and Ligra [24], do not optimize for caches aggressively. On the other hand, disk-based graph frameworks [22, 33] have applied techniques developed for the memory/disk boundary to the cache/DRAM boundary. However, we find that even the fastest of these frameworks, GridGraph, is 3x slower than high performance in-memory frameworks that do not even optimize for cache and over 11× slower than our optimized implementations. The problem is that the performance gap between cache and memory is much smaller than the gap between memory and disk. As a result, it is much harder to tradeoff CPU cycles for cache utilization.

In this paper, we present two such techniques, *frequency based clustering* and *compressed sparse row (CSR) segmenting*, that can achieve significant speedups by improving cache utilization with minimal runtime overhead.

Frequency based clustering improves cache utilization by keeping more of the frequently accessed vertices in the cache without incurring any runtime overhead. In many graph ap-

plications, each random access to DRAM brings only one useful vertex in each cache line. Taking advantage of the power-law degree distribution, we pack popular vertices together in memory to improve overall cache line utilization. Unlike other graph ordering techniques, clustering uses very little preprocessing time and preserves some of the locality in the original ordering of real world graphs by keeping the structure of average degree vertices intact.

CSR segmenting further improves performance by serving all random accesses from the cache, and making all DRAM access sequential. It uses a novel data structure and cache-aware merge algorithm that incur very little runtime overhead, allowing it to perform well at the cache/DRAM boundary. Segmenting first preprocesses the graph to divide the vertex data into cache-sized segments, and partitions the edges based on these segments. It then processes each segment in parallel, making one pass through all the edges while keeping the random accesses to vertex data in the cache. Finally, CSR segmenting builds a data structure that makes it possible to merge the updates from each segment within L1 cache in parallel at a significantly lower overhead than previous approaches.

We implement our techniques in a framework exposing an interface similar to Ligra, and we demonstrate their contributions to significant performance gains on a variety of representative applications compared to the best published results. As shown in Figure 1, our optimizations provide up to a 4.3× speedup for PageRank over Intel's GraphMat, 8.8× over Ligra, up to 6× speedup for Label Propagation, 4× speedup for Collaborative Filtering, and 2× for Betweenness Centrality. We also see similar speedups over the recently proposed Hilbert curve ordering for graph data [18], as we discuss in Section 5.4.[1]

Figure 2 further analyzes the speedup of each technique on PageRank. We see that both clustering and segmenting reduce cycles stalled on memory, and running time falls proportionately. The last bar shows a modified version of PageRank where all reads go to vertex 0, so there is *no* random access to DRAM (but of course the result is incorrect); our optimized version is within 2× of this lower bound.

In summary, our contributions are:

- We present two effective cache optimization techniques that achieve low enough overhead to speed up in-memory graph frameworks:
  - Frequency Based Clustering, which improves cache line utilization with no runtime overhead.
  - CSR Segmenting, which partitions graph data in a novel way to remove random access to DRAM and allow low-overhead merging of updates.

[1] In essence, although Hilbert order gives an effective single-threaded cache-oblivious algorithm, on multicores each core loads a different portion of the graph into cache. In contrast, our segmenting technique lets all cores share the *same* working set in the cache.

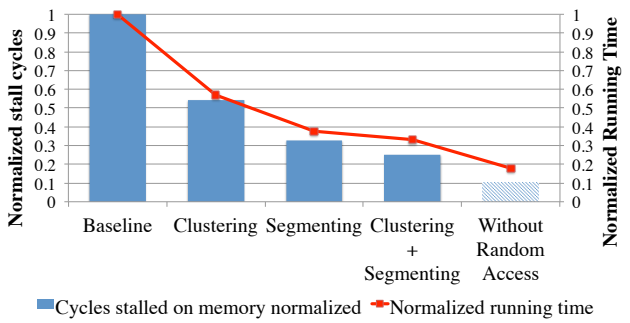


Figure 2: Running time and cycles stalled on memory normalized to the baseline for each of our optimizations in PageRank on the RMAT27 graph. The last bar is an incorrect program where random accesses were removed to provide a lower bound.

- We implement and evaluate these techniques on several representative applications. Our optimizations yield significant speedup over the fastest in-memory graph frameworks.

2. Motivation

We will use PageRank [21] listed in Algorithm 1 as a running example to motivate our optimizations for graph processing. PageRank iteratively updates the rank of each vertex based on the rank and degree of its neighbors. The performance characteristics of PageRank can generalize to a large number of graph applications.

2.1 Graph and Vertex Data Representation

Graph frameworks typically store graph in Compressed Sparse Row (CSR) format. Assuming the graph has V vertices and E edges, CSR format would create a vertex array, `G.vertexArray`, of of size $O(V)$ and an edge array, `G.edgeArray`, of size $O(E)$. Vertex Array stores the indices of the first neighbor of each vertex in the Edge Array and use that to access the neighbor list of each vertex. Application specific data is stored as separate arrays. In the case of PageRank, vertex data is stored as arrays `newRank`, `rank` and `degree` of size $O(V)$.

2.2 Memory Access Pattern

The algorithm sequentially reads size $O(E)$ data. By going over every vertex in order, the algorithm issues sequential read requests to `G.edgeArray` and sequential writes requests to `newRank`. The algorithm randomly reads $O(E)$ times from size $O(V)$ vertex data, including `rank` and `degree`. These read requests are random because we cannot predict the values of `u`.

This pattern of sequentially accessed edge data and randomly accessed vertex data is common in representative graph applications. Collaborative Filtering needs to randomly read each vertex's latent factors, and Betweenness Centrality needs to randomly access the active frontier and number of paths through each vertex.

Algorithm 1 PageRank

```
1 procedure PAGERANK(Graph G)
2     parallel for v : G.vertexArray do
3         for u : G.edgeArray[v] do
4             G.newRank[v] +=
5                 G.rank[u] / G.degree[u]
6         end for
7     end parallel for
8 end procedure
```

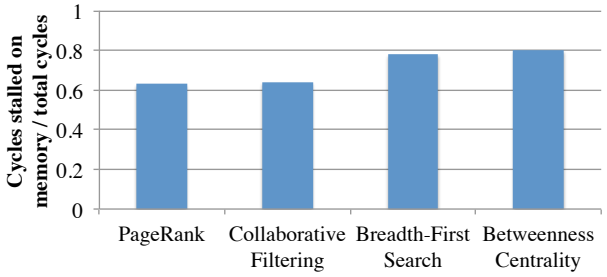


Figure 3: Memory stalls in several applications.

2.3 Random Memory Access Bottleneck

Graph applications have poor cache hit rate and are largely stalled on memory accesses, since the working set when processing realistic graphs is much larger than the last level cache (LLC) of current machines. For example, the 2010 Twitter graph [12] has 41 million vertices and 1.5 billion edges. The `rank` and `degree` arrays, which together form the working set that is randomly accessed, are 656 MB (assuming 64-bit floating point numbers) and are many times larger than the 30–45 MB last-level cache of current CPUs. Even though there is a higher than expected hit rate due to the power-law degree distribution and the community structures in the graph [3], we still find the LLC miss rate for PageRank to be more than 45%.

As a result of the high cache miss rate, our performance profile shows graph applications are spending 60-80% of their cycles stalled on memory access as shown in Figure 3. Random memory access becomes the major bottleneck because random access to DRAM is 6-8x more expensive than random access to LLC or sequential accesses to DRAM. Sequential access to DRAM effectively uses all memory bandwidth consumed by cacheline reads, and benefits from hardware prefetchers to further reduce latency.

3. Frequency Based Clustering

Frequency based clustering reorganizes the physical layout of the vertex data structures to improve cache utilization. It reduces overall cycles stalled on memory by serving more slow random requests in cache, instead of in DRAM.

3.1 Key Observations

We make three key observations on graph access patterns:

First, each random read only uses a small portion of the cache line. The size of the vertex data is 8 bytes for a rank represented as a double, using only 1/8 of a common 64 byte cache line. Since there is little spatial locality, the other elements in the cache line are often not used. This is true for many other graph applications, such as Label Propagation that reads an integer type vertex label.

Second, certain vertices are much more likely to be accessed than others in power law distributed graphs, where a small number of vertices have a large number of edges attached to them [12]. Thus, a large fraction of random read requests will concentrate on a small subset of vertices. These skewed out-degree graphs include social networks, web graphs, and many networks in biology.

Because of the above observations, if we store the vertices in a random order, each high out-degree vertex will likely be on a different cache line in the vertex data array (`rank` in PageRank). The cache line will be "polluted" by the data from low out-degree vertices when it is brought in.

A third observation is that the original ordering of vertices in real world graphs often exhibit some locality. Vertices that are referenced together are sometimes placed close to each other due to the communities existing in these graphs. For example, PageRank on the original ordering of vertices on Twitter graph [12] is 50% faster than a random ordering. As a result, it is also important to utilize the original ordering for improved performance.

3.2 Design

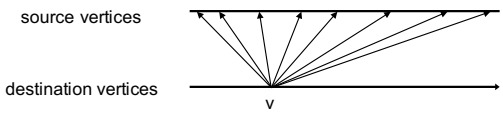
Based on the previous observations, we designed frequency based clustering to group together the vertices that are frequently referenced, while preserving the nature ordering in the real world graphs.

We use out-degrees to select the frequently accessed vertices because many state-of-the-art implementations of graph algorithms use only pull based implementations, or spend a significant portion of the execution time in the pull phase, such as BFS and BC with direction optimization [2]. Furthermore, if we optimize for the push phases by using in-degree of vertices, we incur significant penalty from false sharing when a large number of write requests are directed to the same cache line.

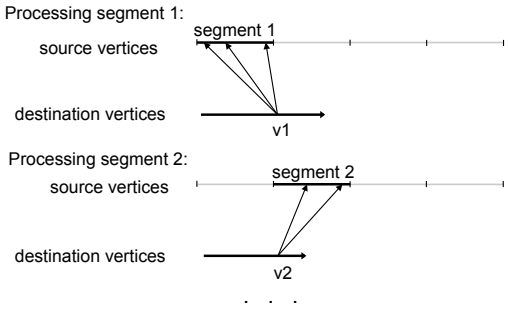
To preserve the original ordering in real world graphs, we cluster together only vertices with out-degree above a threshold, leaving the ordering of other vertices intact. We found that a good threshold is the average degree of nodes. This thresholding allows us to keep some of the locality in the original ordering, yet still offering a cache-oblivious clustering of high-out-degree vertices that maximizes the effectiveness of L1, L2, and L3 caches. More analysis is shown in Section 5.4.

3.3 Implementation

We implemented frequency based clustering in three steps. First we use a parallel stable sort based on vertices' out-



(a) Execution without segmenting: as we iterate through the destination vertices, each vertex v reads data from source vertices throughout the graph, causing random accesses to touch a large working set.



(b) With segmenting, we first iterate through all vertices to apply updates from segment 1, then iterate through to apply updates from segment 2, etc. Random access is now confined within a segment.

Figure 4: CSR Segmenting optimization.

degree/threshold to cluster together frequently referenced vertices. Next, we create a mapping from old vertex index to the newly sorted vertex index. Using the mapping, we go through the `G.edgeArray` to update the index of each vertex, creating a new CSR.

Load balance is critical to achieving high performance with frequency based clustering. Many in-memory graph frameworks, including Ligra, see significant slow down with frequency based clustering, using the default vertex based load balance scheme. This is because the high-degree vertices packed together and the thread responsible for the part of the vertex array containing high out-degree vertices may perform much more work than other threads.

We implemented a work-estimating load balancing that partitions the vertex array based on the number of edges within each task, which reflects how many random reads it will make to the `rank` array. The task then processes its range of vertices if the cost is sufficiently small, or divides into two sub-tasks otherwise.

4. CSR Segmenting

CSR Segmenting improves cache utilization by working on one cache-sized *segment* of vertex data at a time. To make CSR segmenting work for the cache/memory hierarchy, we have to keep runtime overhead low with carefully designed preprocessing, segment processing and cache-aware merging. This technique can be applied to a wide class of computations that aggregate values over the neighbors of each vertex in the graph.

First, consider the PageRank algorithm in Algorithm 1. To compute the new ranks, each vertex randomly access a large array (the ranks of all vertices on the pervious iteration) to find its neighbors' rank. If this array does not fit in the CPU cache, many random accesses go to DRAM.

With segmenting, we partition the graph into subgraphs and make one pass through all the subgraphs. When processing each subgraph, we confine our random accesses to a cache-sized segment (Figure 4). Specifically, we first do a preprocessing step by dividing the previous iteration's rank array into k segments that fits in the CPU's last-level cache. We then construct subgraphs by grouping together all the edges whose sources are in the same segment and construct a data structure for the destination vertices. CSR Segmenting processes one subgraph at a time. Within a subgraph, it iterates over all vertices in parallel and add their contributions from the segment. Some destination vertices would potentially be duplicated across different subgraphs. As a result, we use parallel cache-aware merge to combine the contributions for each vertex from all segments that have edges to it. With this approach, reads and writes to DRAM are both sequential and random access is confined within the cache.

In summary, segmenting has the following benefits:

- Improved cache utilization: It restricts all random accesses to cache, and makes all accesses to DRAM sequential.
- Low overhead: Cache-aware merge only needs a small amount of extra sequential memory accesses and performs the merge in L1 cache in parallel. Processing the graph requires only one sequential pass through all the edges.
- Great parallelism: Within each subgraph, threads can parallelize the execution across all vertices without using atomic operations for synchronization. The merge phase can be parallelized as well.
- Easy to use: It can easily be applied to any algorithm that aggregates values across the graph, as we will discuss by applying it inside Ligra.

We next describe the segmenting process in more detail, starting with preprocessing (Section 4.1), computation within a segment (Sections 4.2), and finally our cache-aware merge algorithm (Section 4.3). We then discuss tradeoffs in choosing the size of segments and the benefits of partitioning after frequency based clustering (Section 4.4). Finally, we discuss how to apply segmenting to other graph computations through an extension of the Ligra API (Section 4.5).

4.1 Preprocessing

The first step for applying segmenting is to preprocess the graph to determine the vertices and edges affected by each segment. This process works as follows:

1. Divide the vertices into segments such as the data for each segment fits in the cache. (Section 4.4 describes tradeoffs in the segment size, e.g., which level of the cache to use.)

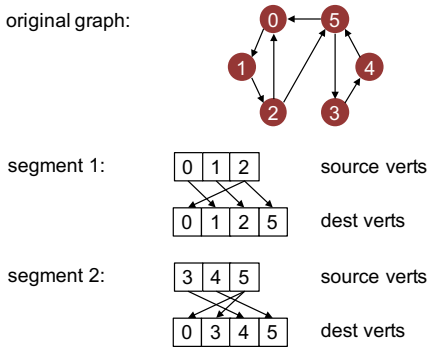


Figure 5: Example of subgraphs created in segmenting. We split the set of vertices, 0..5, into {0,1,2} and {3,4,5}, then build a data structure with the destination vertices and out-edges for each segment.

2. For each segment S, determine the vertices and edges that are adjacent to those in S (i.e., edges with sources in S). A new CSR will be constructed for the vertices in the segment and their adjacent edges. We also create an array to hold the intermediate result for each adjacent (destination) vertex v.
3. Create an index vector with the index of each adjacent (destination) vertex in the original graph, which will be used to combine intermediate results in the merge phase.

Figure 5 shows an example dividing a graph into two segments, and the adjacent vertices and edges for each segment.

Note that this preprocessing phase can be done in parallel, by building each segment separately from the original CSR. In our implementation we find that segmenting only takes time proportional to a few PageRank iterations, shown in Section 5.6.

4.2 Parallel Segment Processing

After the preprocessing is done, the system runs each segment's aggregation in turn. Within each segment, we parallelize the computation across threads by dividing the CSR edge list in the same way as the whole-graph PageRank.

By parallelizing work within a segment, we only need to create a relatively small number of segments, where each segment fits in the last level cache (LLC) and contains a large number of edges. This way, we keep the preprocessing time low, generate good parallelism and avoid high merging cost. Additionally, all the threads share the *same* working set, i.e., the source vertex data in this segment, which is read-only. Thus, adding more threads does not create cache contention.

We also experimented with parallelizing the processing of multiple smaller segments. Each smaller segment's working set fits in L2 cache, instead of LLC, for even lower random access latency. However, we found that the merging overhead that comes with a large number of smaller segments becomes a significant performance bottleneck. As a result, we only exploit parallelism within a single segment.

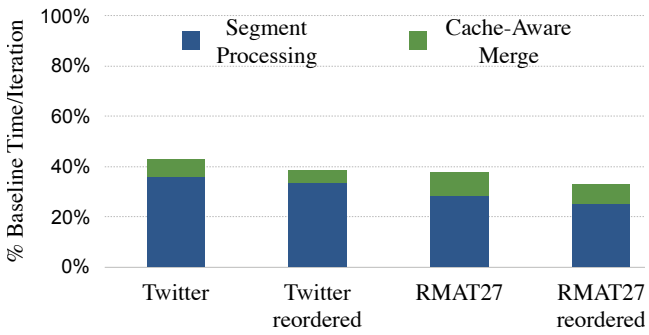


Figure 6: Comparison of segment computation vs merge costs. Runtime% normalized to an optimized PageRank baseline without segmenting on 24 cores.

4.3 Cache-Aware Merge

Once the per-segment passes are done, the system has an intermediate vector with the updates for each vertex from each segment. These intermediate result vectors are sparse, holding data only for the vertices adjacent to each segment. For example, in Figure 5, segment 1 will produce a vector with updates for vertices 0, 1, 2, 5, and segment 2 will produce a vector with updates for vertices 0, 3, 4, 5.

To combine these intermediate results, we use a cache-aware merge algorithm that accesses the input vectors sequentially, requires no branches, and runs in parallel. We divide the range of vertex IDs into L1-cache-sized *blocks*. Then, for each block, a worker thread reads the updates for that range of vertex IDs from the sparse vectors of intermediate results, and updates a dense vector for the final output. A helper data structure holds the start and end index of each output block's vertex IDs in each of the per-segment vectors. Different blocks can be processed in parallel by different threads, and we use a work-stealing load balancing scheme to divide them across processors.[2]

With the cache-aware merge algorithm, merging is not a major portion of execution time. Figure 6 shows the percentage of elapsed time on segment-local computation and merge using 48 hyperthreads for PageRank, normalized to a baseline with all our optimizations other than segmenting. Other overhead includes all other time within each iteration other than edge processing, e.g. the per-vertex division to compute contributions.

4.4 Segment Size Selection

A final consideration in using segmenting is how large to make segments. In general, there is a tradeoff with segment size. Smaller segments will fit into a lower-level cache (e.g., L1 or L2), reducing random access latency. However, smaller segments will also result in more merges for the same destination vertex, because the source vertices point-

[2] One benefit of this approach is that each thread usually processes multiple consecutive blocks, further increasing the range of sequential access for both reads and writes.

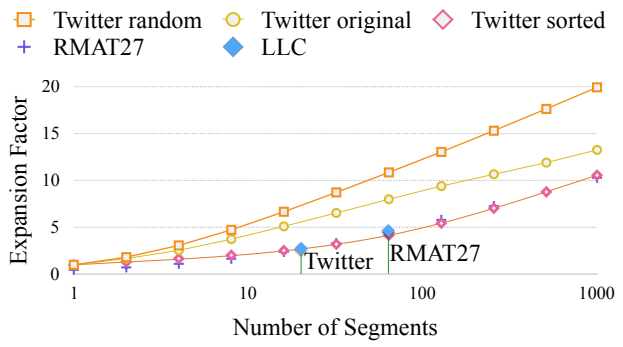


Figure 7: Expansion factors of segmenting for RMAT27 graph (V=134M, d=16), and Twitter graph (V=41M, d=36) in original order, sorted by outdegree, and in a random permutation. LLC sized segments for a 64-bit double per vertex of each graph are marked.

ing to it will be in multiple segments. Across the applications we evaluated, we found that sizing the segments to fit in last level (L3) cache provided the best tradeoff.

To further understand the impact of segment size, we define a metric called the *expansion factor* for a segmented graph. Let s be the number of vertices in each segment, and s_{adj} be the average number of vertices adjacent to each segment, that is, with edges from the segment to them. Then we define $q = s_{\text{adj}}/s$ as the expansion factor. The expansion factor describes how many segments, on average, contribute data to each vertex, and hence how many merge operations happen for each vertex.

Figure 7 shows the expansion factors as a function of number of segments for several graphs while varying number of vertices, average degrees, and vertex order. For PageRank calculations where we need 8-byte data per vertex, a 30 MB LLC cache can fit 4M vertices. For these workloads the expansion factors are less than 5, which is much less than the number of segments or the average degree of the graph which are the upper bounds of q. Randomly permuting vertices or edges, as used in distributed frameworks' partitioning schemes to improve load balance, as illustrated with the randomized Twitter graph, results in a much worse expansion factor.

Segmenting combines well with frequency based clustering as seen in the the expansion factors for the Twitter graph from the original dataset order, and with vertices reordered by outdegree. In power law graphs like social networks, many vertices are only connected to a small number of popular vertices, and so frequency based clustering causes many of these to only read from the first segments. Load balancing across segments is not a goal for our multicore setting, since each segment is processed by all cores.

4.5 Programming Interface

We extended Ligra to do segmenting automatically by changing the EdgeMap API interface and having users provide an additional merge function. Algorithm 2 shows the new update function for optimized PageRank, where we aggregate srcVal (rank/degree) from neighbors.

Algorithm 2 Update Function for Optimized PageRank

```
▷ T is the Value Type specified by user, double in this case
1 procedure UPDATE (T segmentVal, T srcVal, T oldDstVal)
2     segmentVal += srcVal
3 end procedure
4 procedure MERGE ( T newDstVal, T segmentVal)
5     newDstVal += segmentVal
6 end procedure
```

Ligra's original API function `EdgeMap` traverses a given subset of edges, while calling a user-defined update function with the indices of source and destination vertices. Users were allowed to read and update their own data inside these functions by directly indexing the input and output arrays.

We change the API of the update function to work with values of source and destination vertices, instead of their indices, since the indices in the original graph's CSR are different from those in the CSR of the current segmented graph. The users need to specify a value type T and direct all writes to the segmentVal. We also provide access to OldDstVal for applications that need to read the old value of the destination vertices to perform the updates, such as Collaborative Filtering.

We require the specification of a merge function to perform cache-aware merge. In segment processing, we aggregate values locally within each segment with the update function. Cache-aware merge then uses the merge function to aggregate the intermediate results (segmentVal) across different segments into the final output value (newDstVal).

5. Evaluation

5.1 Experimental Setup

We conducted our experiments on a dual socket system with Intel Xeon E5-2695 v2 (IvyBridge) CPUs 12 cores for a total of 24 cores and 48 hyperthreads, with 30 MB last level cache in each socket. The system has 128GB of DDR3-1600 memory. While theoretical peak memory system bandwidth is 102GB/s, practically achievable sequential read bandwidth from each socket is 47GB/s, and when using NUMA interleaving the peak total bandwidth is 79GB/s. The machine runs Ubuntu 14.04, with Linux kernel 3.13, with Transparent Huge Pages (THP) enabled. For parallel execution, we used Intel's Cilk Plus compiler and runtime system from Intel C++ Composer XE 2015 v15.0.3. All code is optimized at highest optimization with `-O3 -ipo` flags. We used Ligra released as of Sep 11th 2015 and GraphMat 1.0 for performance comparisons. We used the best configurations noted by Ligra and GraphMat release. We collected cycles stalled on memory for the applications using `perf`.

Dataset	Number of Vertices	Number of Edges
LiveJournal [8]	5 M	69 M
Twitter [12]	41 M	1469 M
RMAT 25 [6]	34 M	671 M
RMAT 27 [6]	134 M	2147 M
SD [1]	101 M	2043 M
Netflix [4]	0.5 M	198 M
Netflix2x [15]	1 M	792 M
Netflix4x [15]	2 M	1585 M

Table 1: Real world and *synthetic* graph input datasets

Data Sets: We used a mixture of publicly available real world data and synthetic datasets, whose working sets range from mostly fit in the last level cache (LLC) to much larger than the LLC. We used the social networks, including LiveJournal [8] and Twitter [12], the SD web graph from 2012 common crawl [1], and the RMAT graphs generated from the Graph500 [6] benchmark generator with parameters (a=0.57, b=c=0.19, d=0.05). We also synthesized an expanded version of the Netflix dataset to more accurately reflect realistic number of users. While it is reported that Netflix has over 30 million users and 36,000 movies, the public Netflix dataset has only 0.5 million users and 17,770 movies. To expand the dataset while preserving the degree distribution of the original graph, we doubled the number of users and number of movies and quadrupling the number of users, while maintaining similar patterns of reviews [15]. Table 1 summarizes the datasets that we use.

Applications: We choose a representative set of applications from domains such as machine learning, graph traversals and graph analytics.

PageRank, Label Propagation and Collaborative Filtering are dominated by unpredictable vertex data accesses. The algorithms do not require any vertices' activeness checking, similar to other applications such as Triangle Counting. Additionally, PageRank, Label Propagation and Collaborative Filtering take a number of iterations to complete, justifying the preprocessing time.

BC represents the applications that involve vertices' activeness checking and making unpredictable access to vertices' data, such as single source shortest path (SSSP) and PageRank Delta. Betweenness Centrality also takes a large number of iterations, making a case for additional preprocessing time.

Breadth First Search represents a unique class of applications that only need to do activeness checking on the vertices without accessing vertices' data. This application has the smallest working set in graph applications.

5.2 Comparison with Other Frameworks

Tables 2 to 6 compare the running time for our optimized graph algorithms with that of GraphMat, Ligra and GridGraph.

Our Baselines: Our baselines for PageRank, Label Propagation and Collaborative Filtering are hand coded implementations that included many state-of-the-art optimizations, and are often faster than best results from existing frameworks. Our PageRank calculated the contribution (rank/degree) of each vertex beforehand to reduce the number of random accesses and divide instructions. We use Ligra as our BFS and BC baseline to take full advantage of the push and pull switch optimization, missing from GraphMat and GridGraph.

Optimized Versions: Optimized versions apply cache optimizations to the graph applications, using our extended version of Ligra. We compare our optimized versions to

Dataset	Optimized Version	Our Baseline	GraphMat	Ligra	GridGraph
Live Journal	0.017s (1.00×)	0.031s (1.79×)	0.028s (1.66×)	0.076s (4.45×)	0.195 (11.5×)
Twitter	0.29s (1.00×)	0.79s (2.72×)	1.20s (4.13×)	2.57s (8.86×)	2.58 (8.90×)
RMAT 25	0.15s (1.00×)	0.33s (2.20×)	0.5s (3.33×)	1.28s (8.53×)	1.65 (11.0×)
RMAT 27	0.58s (1.00×)	1.63s (2.80×)	2.50s (4.30×)	4.96s (8.53×)	6.5 (11.20×)
SD	0.43 (1.00×)	1.33 (2.62×)	2.23 (5.18×)	3.48 (8.10×)	3.9 (9.07×)

Table 2: PageRank runtime per iteration comparisons with other frameworks and slowdown (against each as a baseline)

Dataset	Optimized Version	Our Baseline	GraphMat
Netflix	0.20s (1×)	0.32s (1.56×)	0.5s (2.50×)
Netflix2x	0.81s (1×)	1.63s (2.01×)	2.16s (2.67×)
Netflix4x	1.61s (1×)	3.78s (2.80×)	7s (4.35×)

Table 3: Collaborative Filtering runtime per iteration comparisons with GraphMat and slowdown (against each as a baseline)

Dataset	Optimized Version	Our Baseline	Ligra
Live Journal	0.02s (1×)	0.01s (0.68×)	0.03s (1.51×)
Twitter	0.27s (1×)	0.51s (1.73×)	1.16s (3.57×)
RMAT 25	0.14s (1×)	0.33s (2.20×)	0.5s (3.33×)
RMAT 27	0.52s (1×)	1.17s (2.25×)	2.90s (5.58×)
SD	0.34 (1×)	1.05 (3.09×)	2.28 (6.71×)

Table 4: Label Propagation runtime per iteration comparisons with other frameworks and slowdown (against each as a baseline)

Dataset	Optimized Version	Ligra (Baseline)
LiveJournal	1.2s (1×)	1.2s (1.00×)
Twitter	14.6s (1×)	17.5s (1.19×)
RMAT 25	7.08s (1×)	11.1s (1.56×)
RMAT 27	21.9s (1×)	42.8s (1.95×)
SD	15.0(1×)	19.7 (1.31×)

Table 5: Between Centrality runtime for 12 different starting points comparisons with Ligra and slowdown (against each as a baseline)

Dataset	Optimized Version	Ligra (Baseline)
LiveJournal	0.36s (1×)	0.33s (0.93×)
Twitter	2.91s (1×)	3.18s (1.09×)
RMAT 25	1.14s (1×)	1.42s (1.24×)
RMAT 27	4.53s (1×)	7.02s (1.54×)
SD	9.08s (1×)	10.8s (1.18×)

Table 6: BFS runtime for 12 different starting points comparisons with Ligra and slowdown (against each as a baseline).

Frameworks	Running Time	Slow Down
GridGraph	12.86s	5.04×
X-Stream	18.22s	7.1×
GraphMat	4.20s	1.64×
Optimized Version	2.55s	1.00×

Table 7: Best reported execution time for 20 iterations of in-memory PageRank on the LiveJournal Graph on i2.xlarge with 4 cores

high-performance baselines to avoid exaggerating our performance gains due to the overheads in existing frameworks

Existing Frameworks: GraphMat holds the record of the fastest published implementation of PageRank and Collaborative Filtering and is over 2x faster than Galois [20]. Collaborative Filtering is only implemented in GraphMat. GridGraph partitions the vertex data and the graph to improve cache performance. We found that the number of partitions suggested in the GridGraph paper gave the best performance, since our machine has a similar LLC size. Ligra has the fastest implementations of Betweenness Centrality and Breadth First Search (on many real world graphs) thanks to its innovative push and pull switch optimization and is very comparable to Galois on power law graphs.

In Table 7, we compare our optimized PageRank with GraphMat and the best running time reported for GridGraph [33] and X-Stream [22] on the LiveJournal graph. The experiments conducted on an i2.xlarge instance, where the whole graph fits in the memory of this system, show that the cache optimized GridGraph and X-Stream are significantly slower than our cache optimized version.

Table 2 to Table 4 show that our optimized PageRank, Collaborative Filtering and Label Propagation's performance improves with the size of the graph. For PageRank, we only achieved 1.6x speedup on the LiveJournal graph compared to GraphMat because the graph is relatively small and most of the frequently referenced data fits in the last level cache. However, for RMAT27, the graph is large enough that the frequently accessed data sets cannot be stored in cache without clustering. As a result, optimizing locality with frequency based clustering and CSR segmenting achieves 4.30x speedup.

Table 5 and Table 6 show that the speedups of our optimized BFS and BC implementations improves as the graph sizes increase. For the same reason, we are not seeing a significant speed up on the LiveJournal graph, because the frequently referenced nodes fit in the last level cache. Apart from being a larger graph, we are better on the RMAT27 graph compared to the Twitter and SD graph because the original graph is already ordered in a way that groups together neighbors while RMAT27 has a random ordering.

5.3 Impact of Each Optimization

In this section, we demonstrate the speedups of the optimizations for various applications in Figure 8 and show the reduction in cycles stalled on memory in Figure 9, Table 8 and Table 9.

Dataset	LiveJournal	Twitter	RMAT 25	RMAT 27
Baseline	475	8,120	5,510	23,264
Clustering	485	7,682	3,250	11,918
Bitvector	431	6,241	3,716	12,578
Clustering + Bitvector	441	5,943	2,643	9,152

Table 8: Total cycles stalled on memory in billions for the optimizations on BC

Dataset	LiveJournal	Twitter	RMAT 25	RMAT 27
Baseline	123	1,519	693	3,711
Clustering	129	1,338	425	2,056
Bitvector	112	1,081	398	2,316
Clustering + Bitvector	108	1,023	306	1,728

Table 9: Total cycles stalled on memory in billions for the optimizations on BFS

Frequency Based Clustering: Clustering is effective on PageRank, Label Propagation, Betweenness Centrality and Breadth-first search because these applications have large working sets, made up of vertex data. Clustering improves the application's running time by reducing cycles stalled on memory accesses as shown in Table 8 and Table 9. Figure 9 demonstrates that clustering significantly reduces cycles stalled per edge for PageRank, and the reduction cycles stalled correlates well with the reduction in running time. On Collaborative Filtering full cache lines are used for per-vertex latent factor vectors, leaving little room for cache line utilization improvements.

For BFS and BC, using bitvector to keep track of the active vertices set is another cache optimization many frameworks adopt for improved performance [23, 27]. We implemented this optimization in Ligra to compare with frequency based clustering. Figure 8 shows that clustering can be as good as bitvector compression without modifying the Ligra framework. Combining clustering with bitvector, we gain an additional 20 percent speedup. Clustering is less effective for LiveJournal and Twitter because they are already in BFS based order that matches these access patterns.

CSR Segmenting: Segmenting alone accelerates PageRank, Label Propagation and Collaborative Filtering by more than 2x. Segmented algorithm serves all of the random read requests in LLC (Last Level Cache), eliminating random DRAM access. The impact of segmenting on time and cycles stalled on memory per edge is evident in Figure 9.

The cycles stalled on memory per edge stay relatively stable for both PageRank and Label Propagation, showing good scalability of the algorithm with regard to the size of the graph. The stability comes from the fact that all random accesses are served in LLC, with almost a fixed latency. On the other hand, the baseline and clustering's cycles stalled on memory per edge increases as we increase the size of the graph because more random reads are served in DRAM. We have also measured the LLC miss rate and find that it

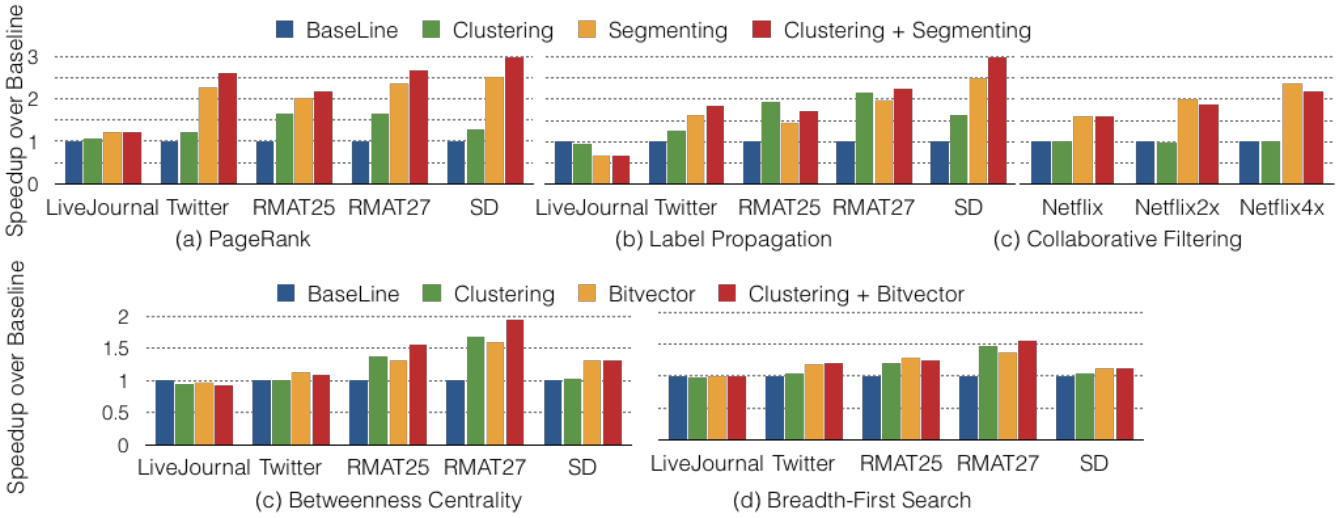


Figure 8: Speedups of optimizations on PageRank, Label Propagation, Collaborative Filtering, Betweenness Centrality and BFS

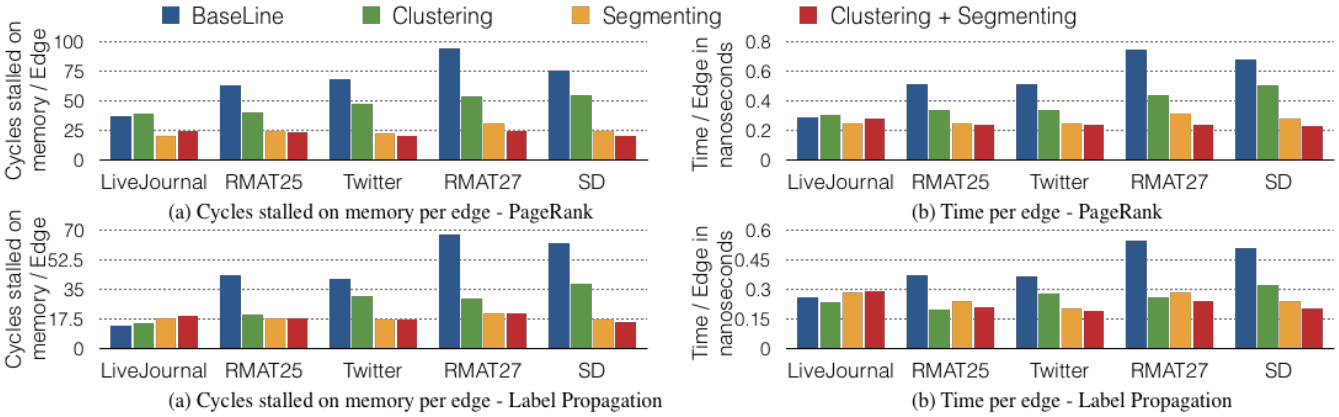


Figure 9: Cycles stalled on memory and time per edge for PageRank and Label Propagation

dropped from 46% to 10% on the Twitter graph after CSR segmenting has been applied.

The speedups with BFS and BC are not as significant as with PageRank because the working set is not large enough to benefit from CSR segmenting The vertex data for BFS and BC already fits in LLC, with no random access to DRAM. However, we do expect to see performance improvement with even larger graphs. We are also only optimizing for the pull portion of the computations.

Combining Frequency Based Clustering and CSR Segmenting: Combining the two techniques achieved even better results on PageRank and Label Propagation because clustering can further make better use of L2 cache within each segment that fit in LLC. The combined technique can further reduce 10-20% of cycles stalled on memory per edge.

5.4 Comparison with Other Orderings

In this section, we compare CSR segmenting with Hilbert order and frequency based clustering with naive in-degree and out-degree sorting.

Several researchers [18, 30] have proposed traversing graph edges along a Hilbert curve, creating locality in both the source vertex read from and the destination vertex written to. While implementation simplicity and good single threaded performance are key benefits, we do not find parallel Hilbert traversal competitive to our techniques.

On a single core, processing the edge list in Hilbert order matches the serial performance of segmenting with clustering. On multiple cores, however, we found that the technique did not scale as well as our approaches.

We tested two approaches to parallelize Hilbert-ordered updates. The first, labeled HAtomic, uses atomic compare-and-swap updates. While this approach scales linearly with the number of threads, performance of atomic operations is 3× worse than non-atomic operations. The second, HMerge, uses an approach from [31] that creates per-thread private vectors to write updates to, and merges them at the end. Only 5% of the runtime is spent on merging the private vectors.

Figure 10 shows the scalability on PageRank of sequential and parallel Hilbert-order implementations using a single

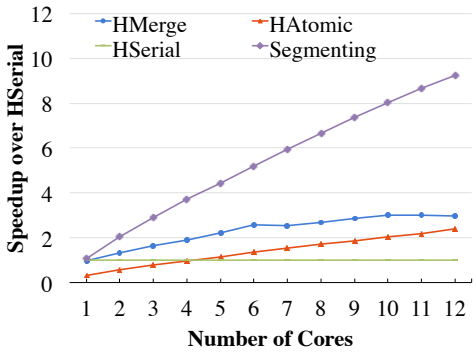


Figure 10: PageRank scalability comparing Hilbert-order HSerial (single output vector) as baseline, vs HAtomic (atomic updates to a shared output buffer), HMerge (merging per-worker output vectors), and Segmenting. Twitter graph edges reordered in Hilbert order, or outdegree for Segmenting.

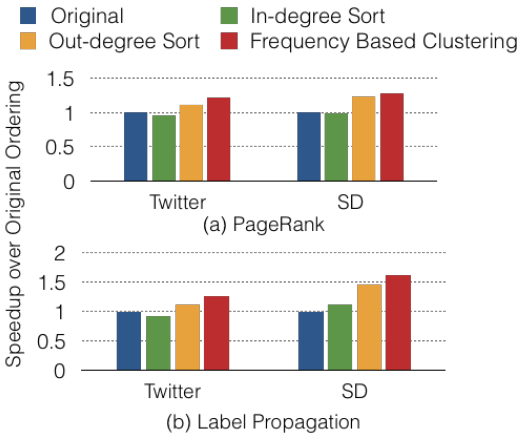


Figure 11: Comparison with other Orderings

NUMA socket. When using all 12 cores, the best runtimes of HSerial (5.4s), HAtomic (2.3s), and HMerge (1.8s) are 3× slower than Segmenting, which takes 0.5 seconds.

The main reason that Hilbert ordering does not scale well is cache contention. Each core has a private L2 cache, however, the Last Level Cache is competitively shared. While Hilbert ordering helps increase locality for each thread, because the threads work on independent regions, they compete for the LLC. Peak performance for HMerge is reached with 10 cores, likely limited by the 20-way cache associativity of the LLC, since each worker thread needs to access both source and destination vector lines. In contrast, segmenting allows multiple threads to share the *same* working set in the LLC, and continues scaling with more cores.

Figure 11 shows that, for real world graphs Twitter and SD, in-degree sorting is not as effective as out-degree based techniques. Frequency based clustering achieves better performance than out-degree sorting alone by preserving some locality in the original ordering of average degree vertices.

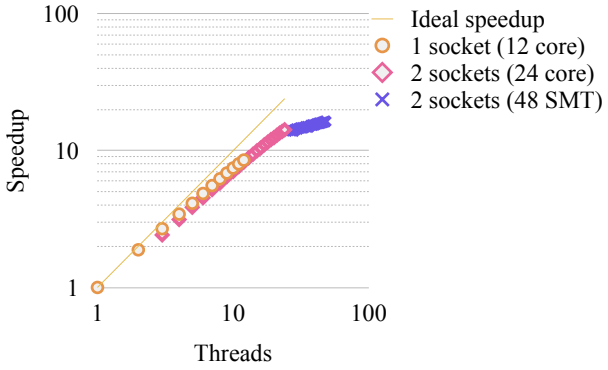


Figure 12: Speedups for PageRank on Twitter (original) with 1 socket, 2 sockets, and using all SMT threads on 2 sockets

5.5 Scalability

Figure 12 shows the scalability of PageRank in more detail compared to our sequential code. We observe 8.5× speedup using 12 cores on the same NUMA socket, 14× speedup with 24 cores interleaved across two sockets, and 16× speedup with all 48 SMT using both hyperthreads per core.

5.6 Preprocessing Time

Table 10 shows the one-time CPU cost of clustering and segmentation. Most of the in-memory frameworks, including GraphMat and Ligra, assume that the graph is already in an adjacency list format, such as CSR. Segmented graphs can be cached and mapped directly from storage for other uses of the same graph that need the same number of segments. GridGraph has a preprocessing step that converts the edge list into a 2D grid. The single threaded grid building process in GridGraph takes an enormous amount of time (193 s for Twitter graph).

Dataset	Clustering	Segmenting	Build CSR
LiveJournal	0.1 s	0.2 s	0.48 s
Twitter	0.5 s	3.8 s	12.7 s
RMAT 27	1.4 s	6.3 s	39.3 s

Table 10: Preprocessing Runtime in Seconds.

We show the time it takes to apply the parallel stable coarse sort based on outDegree to each of the graph. For PageRank and Betweenness Centrality, the preprocessing overhead is small compared to the performance gains. Assuming it typically takes at least 40 iterations for PageRank to converge, the preprocessing time would be well worth it. For RMAT27, the 1.41s preprocessing gives a 0.68s reduction (1.63s to 0.95s) per iteration. For Betweenness Centrality, the algorithm also needs to run for a large number of iterations to get an accurate measurement and the 0.50s preprocessing would save 0.29s per iteration for Twitter graph.

Preprocessing overhead of Hilbert edge reordering is comparable to frequency based clustering, since we need to sort all edges.

Frameworks	Our techniques	GridGraph	X-Stream
Partitioned Graph	segmented CSR	2D Grid	Streaming Partitions
Sequential DRAM traffic	E + 2qV	E + (P+2)V	3E + KV
Random DRAM traffic	0	0	shuffle(E)
Synchronization Overhead	0	E*atomics	0

Table 11: Comparisons with other frameworks optimized for cache. E is the number of edges, V is the number of vertices, q is the expansion factor for our techniques, P is the number of partitions for GridGraph, K is the expansion factor for X-Stream. On Twitter graph, $E = 36V, q = 2.3, P = 32$.

6. Related Work

There have been many projects optimizing graph computations in shared-memory systems, including Ligra, Galois, GraphMat and others [14, 20, 24, 27]. Satish et al. [23] benchmarked many of these frameworks and found them to underperform hand-optimized code. The same authors proposed GraphMat [27], a framework based on sparse matrix operations that matched their hand-optimized benchmarks and reached memory bandwidth limits. Nonetheless, GraphMat still uses memory bandwidth inefficiently and does not optimize for cache aggresively.

GridGraph [33] and X-Stream [22] claimed their techniques for reducing disk access can also be applied to reduce random memory access. Surprisingly, we found that these frameworks are over 3× slower than GraphMat, even when the graphs fit in memory, as shown in section 5.2. Table 11 shows a detailed comparison between their approaches and our techniques. Major sources of slowdowns with costs proportional to the number of edges include excessive sequential memory traffic, additional random DRAM traffic, or atomic updates which are 3× more expensive.

Techniques from other systems optimizing on the disk to memory boundary will also unlikely translate to performance gains as cache optimizations. FlashGraph [32] stores a sorted edge list and partitions the graph in 2D, while we only partition the graph in 1D for better load balance and lower runtime overhead. TurboGraph [9] and GraphChi [13] make several sequential passes over the edges, where we make only one pass.

Graph analytics has also been studied extensively in distributed memory systems like Pregel [17] and GraphLab [16]. These systems partition the graph into subgraphs that can be executed in parallel. Their partitioning model is very different from the segmenting technique we use. Distributed memory systems optimize for minimum communication and good load balance across different partitions that are expected to execute in parallel. In contrast, CSR segmenting processes one segment at a time and optimizes for limiting the range of random access, instead of load balance.

Recent works have looked at speeding up graph application with vertex and edge reordering. A concurrent work [28] applied in-degree sort to many graph algorithms. We show that our frequency based clustering achieves much better performance compared to in-degree sort in Section 5.4. Other techniques studied in the paper incur hard-to-amortize preprocessing overhead, up to 1.5 hrs for Twitter graph. We focus on lightweight techniques with low preprocessing overhead. Degree based reordering has also been used for reducing algorithmic complexity for Triangular Counting [25], while we focus only on the cache performance improvement. Hilbert ordering [18] is an edge ordering technique that was shown to improve the cache performance of single threaded PageRank. We studied Hilbert ordering extensively in Section 5.4 and found that it underperforms our techniques on multicore systems.

Graph compression [5, 7, 11, 26] algorithms utilize vertex reordering heavily. These compression techniques group vertices close to their neighbors, potentially improving the spatial locality of many graph algorithms with significant preprocessing overhead. Frequency based clustering should be able to work with these orderings to achieve even better cache performance.

Finally, graph applications like PageRank are analogous to sparse matrix-vector multiply problems, for which many data layouts and parallelization techniques have been studied [19, 29, 30]. Matrix reordering and cache blocking [10] are designed for similar purposes as frequency based clustering and CSR segmenting, but with a few key differences. Previous matrix reordering techniques have not focused on exploiting the power law degree distribution or inherent ordering found in the real world social and web graphs for improved cache performance. Furthermore, CSR segmenting performs better than previous cache blocking techniques as we do not attempt to fit both the sources (vector) and destinations (corresponding matrix rows) in cache. Our technique fit only the sources (vector) in cache and store the writes sequentially in large buffers, which are later processed using cache-aware merge. This approach allows us to generate greater parallelism, reduce preprocessing time and keep runtime overhead low. Additionally, not all applications that we studied can be easily expressed as SpMv problems. Our work applies similar techniques to the much broader class of algorithms expressible in graph frameworks.

7. Conclusion

Graph analytics are an essential part of modern data analysis workflows, leading to significant work to optimize them on shared-memory machines. Graph applications inherently appear to have poor cache utilization, requiring a large number of random DRAM accesses. In this paper, we showed that substantial improvements can be made over current frameworks through low overhead cache optimizations. We described two effective techniques, frequency based clustering

and CSR segmenting, which yield speedups of up to 4× for PageRank, Label Propagation and Collaborative Filtering, and up to 2× for Betweenness Centrality over the fastest in-memory frameworks. These techniques are broadly applicable inside current graph frameworks and parallelize well on multicores. They work across irregular real world graphs to reduce time stalled on memory and improve performance.

References

[1] Sd-arc web data commons hyperlink graph 2012. http://webdatacommons.org/hyperlinkgraph.

[2] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389013`.

[3] 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*, pages 56–65, Oct 2015. .

[4] J. Bennett, S. Lanning, and N. Netflix. The netflix prize. In *In KDD Cup and Workshop in conjunction with KDD*, 2007.

[5] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 679–688, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. ISBN 0-89871-538-5. URL `http://dl.acm.org/citation.cfm?id=644108.644219`.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, pages 442–446, 2004.

[7] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 219–228, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9. . URL `http://doi.acm.org/10.1145/1557019.1557049`.

[8] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011. .

[9] 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*, KDD '13, pages 77–85, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2174-7. . URL `http://doi.acm.org/10.1145/2487575.2487581`.

[10] E. jin Im and K. Yelick. Optimizing sparse matrix vector multiplication on smps. In *In Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998. ISSN 1064-8275. . URL `http://dx.doi.org/10.1137/S1064827595287997`.

[12] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[13] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[14] M. S. Lam, S. Guo, and J. Seo. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-4909-3. . URL `http://dx.doi.org/10.1109/ICDE.2013.6544832`.

[15] B. Li, S. Tata, and Y. Sismanis. Sparkler: Supporting large-scale matrix factorization. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 625–636, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1597-5. . URL `http://doi.acm.org/10.1145/2452376.2452449`.

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[17] 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, pages 135–146, New York, NY, USA, 2010. ACM.

[18] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015.

[19] Y. Nagasaka, A. Nukada, and S. Matsuoka. Cache-aware sparse matrix formats for kepler gpu. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 281–288, Dec 2014. .

[20] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[22] 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*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[23] 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*, SIGMOD '14, pages 979–990, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2376-5. .

[24] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[25] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 149–160, April 2015. .

[26] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *Proceedings of the 2015 Data Compression Conference*, DCC '15, pages 403–412, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8430-5. . URL `http://dx.doi.org/10.1109/DCC.2015.8`.

[27] 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. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.

[28] 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*, SIGMOD '16, pages 1813–1828, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. . URL `http://doi.acm.org/10.1145/2882903.2915220`.

[29] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3. .

[30] A.-J. Yzelman and D. Roose. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *Parallel and Distributed Systems, IEEE Transactions on*, 25(1):116–125, Jan 2014.

[31] A.-J. N. Yzelman and R. H. Bisseling. A cache-oblivious sparse matrixvector multiplication scheme based on the hilbert curve. In *Progress in Industrial Mathematics at ECMI 2010*, volume 17 of *Mathematics in Industry*, pages 627–633. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-25099-6.

[32] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

[33] X. Zhu, W. Han, and W. Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 375–386, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-225. URL `http://dl.acm.org/citation.cfm?id=2813767.2813795`.