Making Caches Work for Graph Analytics

Yunming Zhang, Vladimir Kiriansky,
Charith Mendis, Saman Amarasinghe
MIT CSAIL
{yunming,vlk,charithm,saman}@csail.mit.edu

Matei Zaharia
Stanford InfoLab
matei@cs.stanford.edu

***Abstract*—Large-scale applications implemented in today's high performance graph frameworks heavily underutilize modern hardware systems. While many graph frameworks have made substantial progress in optimizing these applications, we show that it is still possible to achieve up to 5× 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 the much smaller performance gap between cache and DRAM requires new designs for achieving scalable performance and low overhead. We present Cagra, a cache optimized in-memory graph framework. Cagra uses a novel technique, CSR Segmenting, to break the vertices into segments that fit in last level cache, and partitions the graph into subgraphs based on the segments. Random accesses in each subgraph are limited to one segment at a time, eliminating the much slower random accesses to DRAM. The intermediate updates from each subgraph are written into buffers sequentially and later merged using a low overhead parallel cache-aware merge. Cagra achieves speedups of up to 5× for PageRank, Collaborative Filtering, Label Propagation and Betweenness Centrality over the best published results from state-of-the-art graph frameworks, including GraphMat, Ligra and GridGraph.**

I. Introduction

High performance graph analytics has received considerable research attention, leading to a series of optimized frameworks such as GraphLab [1], Ligra [2], Galois [3] and GraphMat [4]. 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 [2], [5].

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. Cagra employs novel optimizations to achieve up to 5× speedup over state-of-the-art shared memory graph frameworks.

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.

The fastest in-memory frameworks, such as GraphMat and Ligra, do not optimize for caches aggressively. On the other hand, recent disk-based graph frameworks 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 [6], is 3× slower than in-memory frameworks that do not optimize for caches (e.g., GraphMat).

The main challenges in applying cache optimizations are achieving good multicore scalability and keeping the runtime overhead low enough to work with the smaller gap between cache and DRAM. For example, GridGraph [6], which uses a cache friendly 2D grid representation and a scatter-apply execution model, has trouble scaling efficiently beyond 4–6 cores. X-Stream [7] partitions the graph into streaming partitions that fit in cache and processes each partition with a scatter-gather execution model. However, it incurs significant runtime overhead from the additional shuffle and gather phases. To make caches work for graph analytics, we need to carefully design multiple aspects of the system, such as partitioning the graph (2D Grid, Streaming Partitions or other schemes), choosing a data format (sorted compressed graph or unsorted edge list), exploiting parallelism (either within a single partition or parallelism across multiple partitions), utilizing the multi-level cache system (per core L1, L2 or shared LLC), and minimizing overhead. Many of these decisions are also interrelated, adding another layer of complexity to the problem.

In this paper, we present Cagra, a framework that significantly speeds up graph applications on multicores using a new scalable and low-overhead technique to optimize cache usage. The core of Cagra is *compressed sparse row (CSR) segmenting*, a novel partitioning and computation design that limits all random accesses in the system to the cache, and makes all DRAM accesses sequential. Unlike disk-based systems, CSR Segmenting uses a compressed 1D-segmented graph representation to improve the scalability of parallel in-cache processing and reduce overhead. CSR Segmenting first preprocesses the graph to divide the vertex data into cache-sized segments, and partitions the edges into subgraphs based on these segments. It then processes each subgraph in parallel, making one pass through all the edges while keeping the random accesses to vertex data in the cache. The intermediate updates from each segment are locally merged and stored using a buffer to eliminate random writes to DRAM. Finally, CSR Segmenting employs a novel low overhead parallel cache-aware merge algorithm to combine the updates from all the buffers within L1 cache. With CSR segmenting, Cagra achieves up to 5× speedup over previous optimized graph frameworks, including

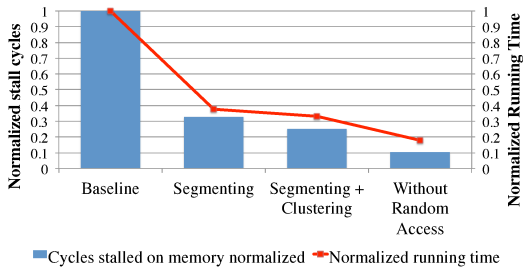


Fig. 1: Normalized running time and cycles stalled on memory for Cagra's optimizations in PageRank on the RMAT27 graph. The last bar is an incorrect program where random accesses were removed to provide a lower bound.

cache-optimized techniques such as GridGraph and Hilbert Curve Ordering [5].

CSR Segmenting can also be combined with Frequency Based Clustering, a variant of degree based graph reordering technique, to further boost cache line utilization and keep frequently accessed vertices in fast cache. In many graph applications, each random access brings only one useful vertex data 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. Cagra first applies frequency based clustering to the entire graph and then performs CSR segmenting.

We evaluate Cagra with various applications on large graphs with skewed degree distributions, as found in real world social, web and rating graphs. Cagra achieves 5× speedup for PageRank, Label Propagation, 4× speedup for Collaborative Filtering, and 2× for Betweenness Centrality over the fastest previous frameworks.

Figure 1 explains our speedups further by showing how CSR segmenting and clustering reduce the cycles stalled on memory for PageRank. The last bar shows a modified version of PageRank where all reads go to vertex 0 with *no* random access to DRAM; Cagra is within 2× of this lower bound.

In summary, our contributions are:

- We propose CSR Segmenting, which partitions graph data in a novel way and uses a cache-aware merge to process the intermediate updates. CSR Segmenting eliminates all random access to DRAM and makes all DRAM access sequential, achieving scalable cache-efficient graph processing.
- We present Cagra, a new framework that offers a programming interface to automatically apply CSR Segmenting to various graph applications.
- We evaluate Cagra on several representative applications and demonstrate significant speedup over best published results. We also provide a detailed analysis of Cagra's cache performance with stall cycles analysis and compare Cagra with cache optimized GridGraph and Hilbert Curve Ordering techniques.

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
```

II. Motivation

We will use PageRank 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.

A. 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 length $O(V)$ 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 length $O(V)$.

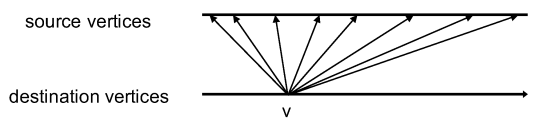
B. Memory Access Pattern

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`.

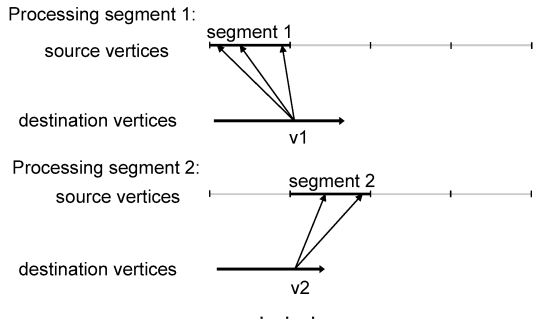
This pattern of sequentially accessed edge data and randomly accessed vertex data is common in representative graph applications. Collaborative Filtering needs to randomly read each vertex's latent factors, and Betweenness Centrality needs to randomly access the active frontier and number of paths through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

C. Random Memory Access Bottleneck

Graph applications have poor cache hit rate and are largely stalled on memory accesses, since the working set of realistic graphs is much larger than the last level cache (LLC) of current machines. For example, the Twitter graph [8] 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 doubles) and are many times larger than the 30–55 MB LLC 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



(a) Without segmenting: we iterate through the destination vertices and read data from source vertices throughout the graph, causing random accesses to touch a large working set.



(b) With segmenting, random accesses are now confined within a cache size segment.

Fig. 2: Processing each segment.

[9], 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. 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, and benefits from hardware prefetchers to further reduce latency.

III. 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 graph processing scalable across all cores and 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 accesses 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, shown in Figure 2(a).

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 2(b)). Specifically, we first do a preprocessing step by dividing the previous iteration's rank array into k segments that fit 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

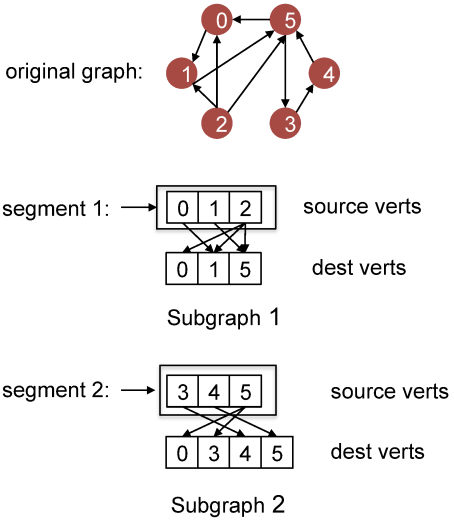


Fig. 3: Example of subgraphs created in segmenting. We split the set of vertices, 0..5, into segments {0,1,2} and {3,4,5}, then build subgraphs with the destination vertices and out-edges for each segment.

processes one subgraph at a time. Within a subgraph, it iterates over all vertices in parallel and adds 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, as shown in Figure 4.

Segmenting can also potentially be applied to the push version of the algorithms. However, we focus on the InEdge processing (pull) version as it does not require atomic synchronizations, and takes most of the execution time [2].

In summary, segmenting has the following benefits:

- Improved cache utilization: It restricts all random reads and writes to cache, and makes all accesses to DRAM sequential.
- Great scalability: Within each subgraph, threads can parallelize the execution across all vertices without using atomic operations for synchronization. Each subgraph is only partitioned on the source vertices, not on both the source and destination vertices, providing ample parallelism. The merge phase can be parallelized as well.
- 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.
- Easy to use: It can easily be applied to any algorithm that aggregates values across the graph with a clean API provided by Cagra.

We next describe and analyze the preprocessing, segment processing and cache-aware merge in more detail.

A. Preprocessing

The preprocessing algorithm is shown in Algorithm 2 and Figure 3 shows an example dividing a graph into two sub-

Algorithm 2 Preprocessing

```
Input: Number of vertices per segment N, Graph G
for v : G.vertices do
    for inEdge : G.inEdges(v) do
        segmentID ← inEdge.src/N
        subgraphs[segmentID].addInEdge(v, inEdge.src)
    end for
end for
for subgraph : subgraphs do
    subgraph.sortByDestination()
    subgraph.constructIdxMap()
    subgraph.constructBlockIndices()
    subgraph.constructIntermBuf()
end for
```

graphs based on the segments. The first step is to construct the subgraphs based on the segments. We first divide the vertices into segments such that the data for each segment fits in the cache. For each segment S, we construct a new subgraph consisting of edges whose sources are in the segment. To do this, we compute the segmentID(subgraphID) of each inEdge by dividing the sourceID of the inEdge by the number of vertices in each segment N and then add the edge to the subgraph. The edges in each subgraph are sorted by their destinations ($sortByDestination$). This step takes no time since the original graph in CSR is already sorted by destination. Then, a CSR representation will be constructed for each subgraph. The algorithm also creates an array, $intermBuf$, to hold the intermediate result for each destination vertex v. Additionally, we create an index mapping, $idxMap$, to map local index of destination vertices in the subgraph to their global index in the original graph. Finally, we create an index of blocks that stores block starts and ends used in cache-aware merge. Note that this preprocessing phase can be done in parallel by building each subgraph separately from the original CSR.

B. Parallel Segment Processing

After the preprocessing is done, the system processes each subgraph in turn, as shown in Figure 2. Within each subgraph, we parallelize the computation across different vertices. We made three key design choices in segment processing, shown in Algorithm 3.

First we exploit parallelism within a single large segment that fits in LLC, instead of across multiple smaller segments. This way, all the threads in our approach 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.

Next, we divide the graph based on only source vertices, and not on both source the destination vertices (2D partitioning), to achieve good scalability. Many other frameworks, such as GridGraph, use 2D partitioning to make sure reads and writes happen in cache. However, this approach can create a large number of subgraphs with a small number of edges, resulting in poor scalability when processing each subgraph in parallel. Instead, we limit only the reads to be in cache, allowing writes to DRAM, as long as they are sequential.

Algorithm 3 Parallel Segment Processing

```
for subgraph : subgraphs do
    parallel for v : subgraph.Vertices do
        for inEdge : subgraph.inEdges(v) do
            Process inEdge
        end for
    end parallel for
end for
```

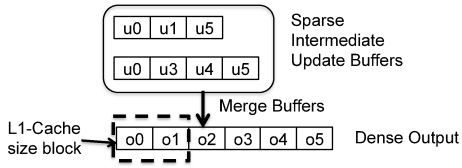


Fig. 4: Cache-aware merge

Finally, we parallelize across different vertices, but not within the same vertex. This parallelization approach takes advantage of the CSR format of each subgraph to generate large degree of parallelism without using atomics for synchronization, since the updates to each vertex in the subgraph are locally merged by the same worker thread.

C. Cache-Aware Merge

Once the per-segment passes are done, Cagra fills up the intermediate buffers for each subgraph with the updates for vertices. These buffers are sparse, holding data only for the destination vertices in each segmented subgraph. For example, in Figure 3, segmented subgraph 1 will produce a buffer with updates for vertices 0,1,5 and the second buffer for subgraph 2 will produce a buffer with updates for vertices 0,3,4,5, as shown in Figure 4.

To combine these intermediate update buffers into one dense output vector, we use a cache-aware merge algorithm (Algorithm 4). The algorithm accesses the intermediate buffers sequentially, requiring 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 buffers of intermediate results, and updates a dense vector *output* for the final output using the local to global index mapping *idxMap*. Helper data structures *blockStarts* and *blockEnds* hold the start and end local index of each output block's vertex IDs in each of the per-subgraph buffers, ensuring the random access is confined in L1 cache. Different blocks can be processed in parallel by different threads, and we use a work-stealing load balancing scheme to divide them across processors.[1]

[1]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.

Algorithm 4 Cache-Aware Merge

```
parallel for block : blocks do
    for subgraph : G.subgraphs do
        blockStart ← subgraph.blockStarts[block]
        blockEnd ← subgraph.blockEnds[block]
        intermBuf ← subgraph.intermBuf
        for localIdx from blockStart to blockEnd do
            globalIdx ← subgraph.idxMap[localIdx]
            localUpdate = intermBuf[localIdx]
            merge(output[globalIdx], localUpdate)
        end for
    end for
end parallel for
return output
```

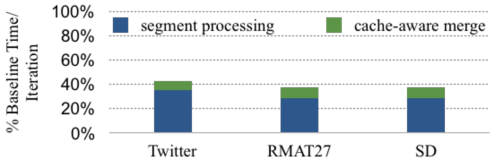


Fig. 5: Comparison of segment computation vs merge costs. Runtime% normalized to an optimized PageRank baseline without segmenting on 24 cores.

With the cache-aware merge algorithm, merging adds only a small runtime overhead. Figure 5 shows the percentage of time on segment processing and merge using 48 hyperthreads for PageRank, normalized to a baseline with all our optimizations other than segmenting.

D. Segment Size Selection

A final consideration in using segmenting is how large to make the 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 pointing 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*. 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 6 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 (24 for the Twitter graph) which are the upper bounds of q. Randomly permuting vertices results in a much worse expansion factor.

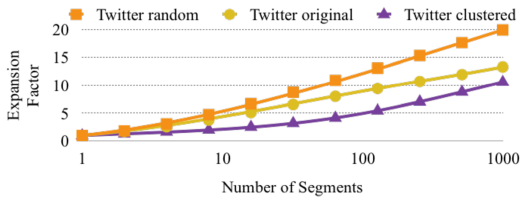


Fig. 6: Expansion factors of segmenting Twitter graph in original and random order, frequency based clustered order.

E. Analysis of Memory Access Costs

We analyze the memory access efficiency of CSR segmenting by analyzing the total traffic between LLC and DRAM and the total number of cache misses in LLC.

Traffic between LLC and DRAM: Assume we have k segments, and the expansion factor detailed in the last section is q. When processing each segment, we only need to read in V/k source vertex data and write qV/k intermediate updates buffer to memory. Cagra makes only one pass through all the edges. As a result, it incurs a total of $E + qV + V$ traffic during segment processing phase. In cache-aware merge, we first read back all the intermediate buffers (qV) to perform the merge and write back V final values, incurring $qV + V$ traffic. Summing up both phases, we get the total DRAM traffic:

$$E + 2qV + V$$

We compare Cagra's sequential and random memory traffic with other cache optimized frameworks, including GridGraph and X-Stream, in Section VII.

IV. Programming Abstraction

Cagra extends the EdgeMap and VertexMap API from Ligra to support automatic cache optimizations. Cagra processes a directed graph $G = (V, E)$, where V is the vertex set and E is the edge set. Undirected graph is represented with edges in both directions. Algoirithm 5 demonstrates the pseudocode of PageRank implemented in Cagra.

EdgeMap (G, ActiveFrontier, EdgeUpdate, Merge) Similar to the original Ligra API, `EdgeMap` traverses the edge set E, and applies an `EdgeUpdate` function to the edge if the source node is in the ActiveFrontier. Cagra requires users to provide a `Merge` function to perform cache-aware merge, as shown in Algorithm 4. Furthermore, the update function works with data values of source and destination vertices, instead of their indices as in Ligra, since the indices in the original graph's CSR are different from those in the CSR of the current segmented graph. The users direct all writes to the segmentVal.

VertexMap (G, VertexSubset, VertexUpdate) VertexMap applies the user defined VertexUpdate function to every vertex in the VertexSubset.

V. Frequency Based Clustering

CSR Segmenting can also be combined with Frequency Based Clustering, an out-degree based graph reordering technique, to further boost cache line utilization and keep frequently accessed vertices in fast cache. Frequency Based Clustering reorganizes the physical layout of the vertex data structures to improve cache utilization. It reduces overall

Algorithm 5 PageRank in Cagra

```
typedef double vertexDataType
contrib ← {1/outDegree[v], ...}
newRank ← {0.0, ...}

procedure EDGEUPDATE(bufVal, srcVal, dstVal)
    bufVal+ = srcVal
    return true
end procedure

procedure MERGE(newDstVal, bufVal)
    newDstVal+ = bufVal
end procedure

procedure VERTEXUPDATE(v)
    newRank[v] ← 0.15 + 0.85 * newRank[v]
    newRank[v] ← newRank[v]/outDegree[v]
    contrib[v] ← 0.0
    return true
end procedure

procedure PAGERANK(G, maxIter)
    iter ← 0
    A ← V
    while iter ≠ maxIter do
        A ← EdgeMap(G, A, EdgeUpdate, EdgeMerge)
        A ← VertexMap(G, A, VertexUpdate)
        Swap(contrib, newRank)
        iter ← iter + 1
    end while
end procedure
```

cycles stalled on memory by serving more random requests in fast storage.

Motivation: We make three key observations on graph access patterns and motivate frequency based clustering. First, each random read in graph applications often only uses a small portion of the cache line. For PageRank, 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. 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 [8]. 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 existing communities.

Design and Implementation: We designed frequency based clustering to group together the vertices that are frequently referenced, while preserving the natural ordering in the real world graphs. We use out-degrees to select the frequently accessed vertices because many graph algorithms use only pull based implementations, or spend a significant portion of the execution time in the pull phase. To preserve the original ordering in real world graphs, we cluster together only vertices with out-degree above the average degree of nodes. This thresholding allows us to keep some of the locality in the original ordering, yet still offering a clustering of high-out-degree vertices that maximizes the effectiveness of L1, L2, and L3 caches.

Dataset	Number of Vertices	Number of Edges
LiveJournal [10]	5 M	69 M
Twitter [8]	41 M	1469 M
RMAT 25 [12]	34 M	671 M
RMAT 27 [12]	134 M	2147 M
SD [11]	101 M	2043 M
Netflix [13]	0.5 M	198 M
Netflix2x [14]	1 M	792 M
Netflix4x [14]	2 M	1585 M

TABLE I: Real world and *synthetic* graph input datasets

We use a parallel stable sort based on vertices' out-degree/threshold to cluster together frequently referenced vertices. Next, we create a mapping from old vertex index to the newly sorted vertex index and use the mapping to update the vertex index in the `G.edgeArray`. Load balance is critical to achieving high performance with frequency based clustering. 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 scheme 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.

Combining Clustering and CSR Segmenting: Segmenting works well with frequency based clustering. Cagra first applies frequency based clustering on the entire graph and then proceeds to apply CSR segmenting. The first advantage is that Cagra can now make better use of faster higher level caches. Additionally, the clustered graph requires less extra sequential memory overhead. The expansion factors shown in Figure 6 for the clustered Twitter graph is over 2× smaller than the original graph, reducing sequential memory traffic overhead.

VI. EVALUATION

We demonstrate up to 5× speedup for various graph applications over best published results from state-of-the-art graph processing frameworks, and 3× speedup over previously expert hand optimized C++ implementations. We provide a detailed analysis on cycles stalled on memory to show that Cagra's improved cache efficiency. Finally, we show Cagra's good scalability and low runtime overhead through comparisons with other cache optimized frameworks and techniques, including GridGraph and Hilbert Curve Ordering.

A. Experimental Setup

We conducted our experiments on a dual socket system with Intel Xeon E5-2695 v2 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. The machine runs with Transparent Huge Pages (THP) enabled.

Data Sets: We used the social networks, including LiveJournal [10] and Twitter [8], the SD web graph from 2012 common crawl [11], and the RMAT graphs. We also synthesized an expanded version of the Netflix dataset.Table I summarizes the datasets that we use.

Dataset	Cagra	HandOpt C++	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 II: PageRank runtime per iteration comparisons with other frameworks and slowdown relative to Cagra

Dataset	Cagra	HandOpt C++	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 III: Collaborative Filtering runtime per iteration comparisons with GraphMat and slowdown relative to Cagra

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

B. Comparison with Hand Optimized Implementations and Other Frameworks

Tables II to V compare the running time for Cagra with that of GraphMat, Ligra and GridGraph.

Hand Optimized C++ Implementations: We used hand optimized C++ implementations for PageRank, Label Propagation and Collaborative Filtering. These implementations are based on previous work [15], and included many optimizations, such as hand-tuned work stealing load balance scheme, replacing expensive divisions to multiplications of reciprocal, working set compression by precomputing rank divided by degree, vectorization of loads, software prefetching and removal of unnecessary branches. These implementations do not apply CSR segmenting and frequency based clustering.

Existing Frameworks: GraphMat holds the record of the fastest published implementation of PageRank and Collaborative Filtering and is over 2x faster than Galois [3]. Collaborative Filtering is only implemented in GraphMat. GridGraph partitions the vertex data and the graph to improve cache performance. We used the number of partitions suggested in the GridGraph paper which we verified gave their best performance, since our machine has a similar LLC size. Ligra has the fastest implementations of Betweenness Centrality on many real world graphs thanks to its innovative push and pull switch optimization, and it is comparable to Galois on power law graphs.

Dataset	Cagra	HandOpt C++	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 IV: Label Propagation runtime per iteration comparisons with other frameworks and slowdown relative to Cagra

Dataset	Cagra	Ligra
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 V: Between Centrality runtime for 12 different starting points comparisons with Ligra and slowdown relative to Cagra

Table II to Table IV show that Cagra's 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. Betweenness Centrality has smaller working set than the other applications.

C. Analysis of Optimizations

In this section, we demonstrate the speedups of the optimizations for various applications in Fig. 7 and show the reduction in cycles stalled on memory in Fig. 8.

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, eliminating random DRAM access. The impact of segmenting on time and cycles stalled on memory per edge is evident in Fig. 8.

The cycles stalled on memory per edge for Segmenting stay low and stable for both PageRank and Label Propagation across graphs of increasing sizes. The stability comes from the fact that all random accesses are served in LLC, with almost a fixed latency. On the other hand, the hand optimized C++ 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 dropped from 46% to 10% on the Twitter graph after CSR segmenting has been applied.

Frequency Based Clustering: Clustering is effective on PageRank, Label Propagation, Betweenness Centrality. Fig. 8 demonstrates that clustering significantly reduces cycles stalled. For Betweenness Centrality, clustering can help take advantage of the higher level L2 caches as the working set is not much larger than LLC. On Collaborative Filtering, full cache lines are used for per-vertex latent factor vectors, leaving little room for cache line utilization improvements.

Combining Frequency Based Clustering and CSR Segmenting: Combining the two techniques achieved even better results on PageRank and Label Propagation because clustering

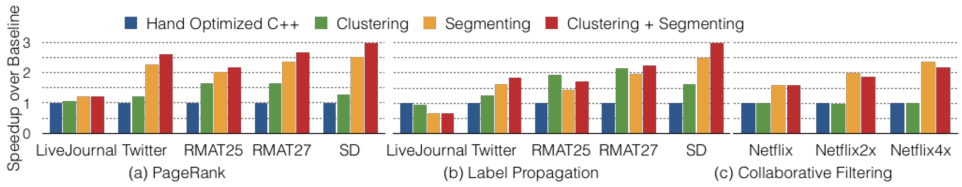


Fig. 7: Speedups of optimizations on PageRank, Label Propagation, Collaborative Filtering

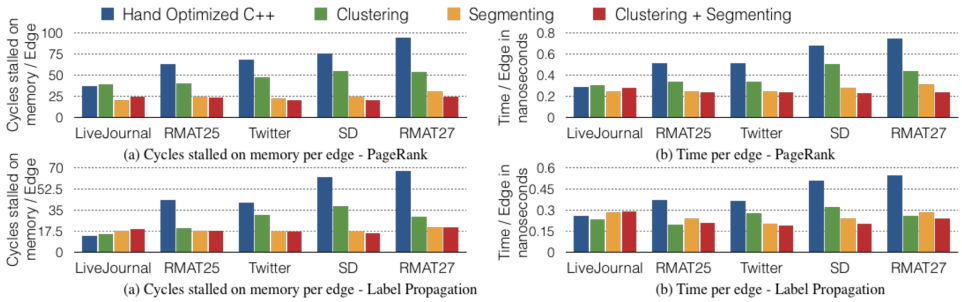


Fig. 8: Cycles stalled on memory and time per edge for PageRank and Label Propagation. Cycles stalled per edge for Clustering + Segmenting is low and stable across graphs with increasing sizes, demonstrating that random accesses are confined in LLC.

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, achieving another 20% speedup over segmenting alone on large graphs, including RMAT27 and SD.

D. Comparison with Other Cache Optimizations

In this section, we compare Cagra with Hilbert Ordering and GridGraph. Several researchers [16], [5] have proposed traversing graph edges along a Hilbert curve, creating locality in both the source vertex read from and the destination vertex written to.

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 [17] 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 9 shows the scalability on PageRank of parallel Hilbert-order implementations using a single 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 Cagra, 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. In contrast, segmenting allows multiple threads to share the *same* working set in the LLC, and continues scaling with more cores.

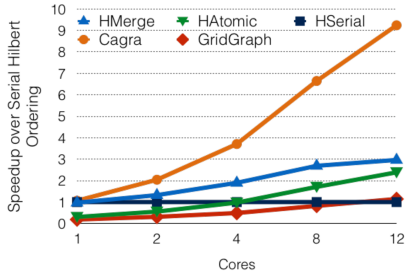


Fig. 9: PageRank speedups on Twitter graph over HSerial (serial Hilbert-order) of HAtomic (parallel Hilbert ordering with atomics), HMerge (parallel Hilbert ordering with buffers), Cagra and GridGraph.

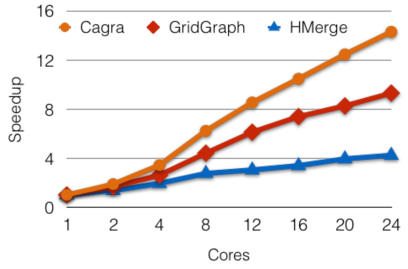


Fig. 10: Scalability for PageRank on Twitter

Fig. 10 shows that Cagra is significantly more scalable than cache optimized GridGraph or parallel Hilbert ordering and achieves 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.

E. Preprocessing Time

Table VI shows Cagra's preprocessing cost. Cagra first computes the degrees of vertices and uses a parallel prefix sum to construct the CSR of the input graph. The CSR is then processed with clustering, and later partitioned into

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 VI: Preprocessing Runtime in Seconds.

Frameworks	Cagra	GridGraph	X-Stream
Partitioned Graph	1D-segmented CSR	2D Grid	Streaming Partitions
Sequential DRAM traffic	E + (2q+1)V	E + (P+2)V	3E + KV
Random DRAM traffic	0	0	shuffle(E)
Parallelism	within 1D-segmented subgraph	within 2D-partitioned subgraph	across many streaming partitions
Runtime Overhead	Cache-aware merge	E*atomics	shuffle and gather phase

TABLE VII: 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, $E = 36V, q = 2.3, P = 32$.

compressed subgraphs during segmenting.

Cagra's preprocessing cost is small compared to its significant performance gains. We do not include the preprocessing cost in Table II to Table V, since other frameworks also incur significant preprocessing costs that were not included. GraphMat and Ligra require the construction of CSR. Cagra's clustering and segmenting cost can easily be amortized by performance gains over Ligra and GraphMat in 2–5 iterations, when applications, such as PageRank, can run for more than 20 iterations. GridGraph uses a special 2D grid representation of the graph that does not sort the edges by their destinations. However, 2D partitioning also takes significant more time than 1D partitioning used in Cagra. GridGraph's preprocessing time for Twitter graph is 130s, much more than the 17s needed by Cagra and is 9–11$\times$ slower than Cagra on PageRank. Finally, the 1D segmented graphs can also be reused across multiple graph applications, further amortizing the preprocessing cost.

VII. Related Work

There have been many projects optimizing graph computations in shared-memory systems, including Ligra, Galois, GraphMat and others [2], [3], [4]. Satish et al. [15] benchmarked many of these frameworks and found them to underperform hand-optimized code. The same authors proposed GraphMat [4], a framework based on sparse matrix operations that matched their hand-optimized benchmarks. Nonetheless, GraphMat still uses memory bandwidth inefficiently and does not optimize for cache aggressively.

GridGraph [6] and X-Stream [7] claimed their techniques for reducing random disk access can also be applied to reducing random memory access. GridGraph partitions the edges on both sources and destinations (2D partitioning) into subgraphs. Each subgraph is processed separately to make sure reads from sources and writes to destinations happen in cache. The issue with 2D partitioning is that the number of edges in each subgraph can be relatively small, making it difficult to scale efficiently beyond 4-6 cores when each subgraph is processed in parallel. Cagra produces much better parallelism with a 1D segmented CSR scheme. Additionally, GridGraph uses atomics for parallel execution, incurring significant runtime overhead. X-Stream uses streaming partitions to keep the random reads in fast storage and reduce random writes in slow storage through a scatter-shuffle-gather design. X-Stream incurs heavy overhead by requiring additional sequential memory traffic for streaming the updates, extra random memory accesses and execution time for shuffle and gather phases. Cagra completely eliminates random writes to DRAM and keeps the runtime overhead low with the cache-aware merge algorithm. Table VII shows a detailed comparison between Cagra, GridGraph and X-Stream. Techniques from other systems optimizing on the disk to memory boundary will also unlikely translate to performance gains as cache ptimizations. GraphChi [18] uses shards with Parallel Sliding Windows to keep random access low. However, it needs to stream the edges twice, and the updates from processing each interval will incur random writes in slow storage.

Polymer [19] is a NUMA-optimized framework that focuses on minimizing both remote random access and cross-NUMA-node access. It uses a 1D graph partitioning scheme, but incurs slow remote writes for vertex data updates. While we do not focus on NUMA in this work, we believe that our techniques could further improve intra-socket performance in Polymer.

Graph analytics has also been studied extensively in distributed memory systems [1], [20]. These systems partition the graph into subgraphs that can be executed in parallel. Their partitioning model optimize for minimum communication and good load balance across different partitions. In contrast, CSR segmenting processes one segment at a time and optimizes for limiting the range of random access, instead of load balance. PowerLyra also exploits the skewed degree of the graphs with differentiated processing.

Recent works have looked at speeding up graph application with vertex and edge reordering. A concurrent work [21] applied in-degree sort to many sequential graph algorithms. Frequency Based Clustering improves the performance compared to in-degree sort by preserving locality in the original ordering of the graphs. Hilbert ordering [5] is an edge ordering technique that was shown to improve the cache performance of graph algorithms. We studied Hilbert ordering extensively in Section VI-D and found that it underperforms our techniques on multicore systems.

For graphs with poor locality, such as uniform random graph, propagation blocking [22] and milk compiler [23] achieves good cache performance by reorganizing random memory accesses into sequential ones at runtime. Cagra does more in preprocessing with little runtime overhead.

Finally, graph applications like PageRank are analogous to sparse matrix-vector multiply problems, for which techniques, such as cache blocking, have been proposed [16], [24]. CSR Segmenting performs better than previous cache blocking techniques as we do not fit both the sources and destinations

(2D blocking) in cache. Our technique fit only the sources in cache (1D segmenting) 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, such as collaborative filtering, can be easily expressed as SpMv problems.

VIII. 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 show that substantial performance improvements can be obtained by eliminating random DRAM access with CSR segmenting. CSR Segmenting uses a novel 1D segmentation and cache-aware merge scheme to achieve scalable performance with low runtime overhead. We then present our framework, Cagra, that applies CSR segmenting to various graph applications with an easy to use interface. Cagra is up to 5× faster than the best published results for common graph applications from high performance graph frameworks and up to 3× faster than expert optimized C++ implementations.

IX. Acknowledgments

We thank William Hasenplaugh for discussions and suggestions on the paper, Julian Shun, MIT COMMIT group members and our reviewers for the many feedback. This research is supported by affiliate members and other supporters of the Stanford DAWN project (Intel, Microsoft, Teradata, and VMware) as well as NSF CAREER grant CNS-1651570, grant from Toyota Research Institute, DARPA grant FA8750-17-2-0126, DOE awards DE-SC008923 and DE-SC014204.

References

[1] 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.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[2] 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*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 135–146.

[3] 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*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 456–471.

[4] 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.*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015.

[5] 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.

[6] 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*, ser. USENIX ATC '15. Berkeley, CA, USA: USENIX Association, 2015, pp. 375–386. [Online]. Available: http://dl.acm.org/citation.cfm?id=2813767.2813795

[7] 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*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 472–488.

[8] 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*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 591–600.

[9] 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*, Oct 2015, pp. 56–65.

[10] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

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

[12] 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*, 2004, pp. 442–446.

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

[14] B. Li, S. Tata, and Y. Sismanis, "Sparkler: Supporting large-scale matrix factorization," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13. New York, NY, USA: ACM, 2013, pp. 625–636. [Online]. Available: http://doi.acm.org/10.1145/2452376.2452449

[15] 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*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 979–990.

[16] A.-J. Yzelman and D. Roose, "High-level strategies for parallel shared-memory sparse matrix-vector multiplication," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 1, pp. 116–125, Jan 2014.

[17] 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*, ser. Mathematics in Industry. Springer Berlin Heidelberg, 2012, vol. 17, pp. 627–633.

[18] 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*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 31–46.

[19] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 183–193.

[20] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 1:1–1:15.

[21] 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*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 1813–1828. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915220

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

[23] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: ACM, 2016, pp. 299–312. [Online]. Available: http://doi.acm.org/10.1145/2967938.2967948

[24] 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*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 38:1–38:12.