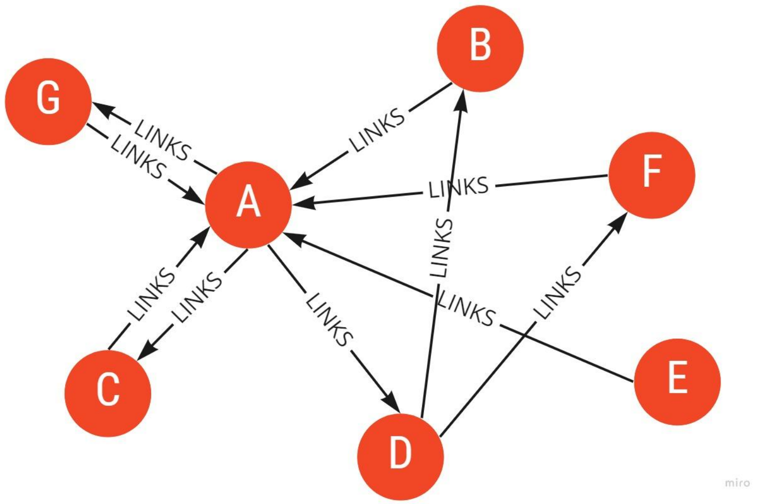
Making Caches Work for Graph Analytics

By: Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, Matei Zaharia

Presented By: Kevin Tong, MIT 6.506

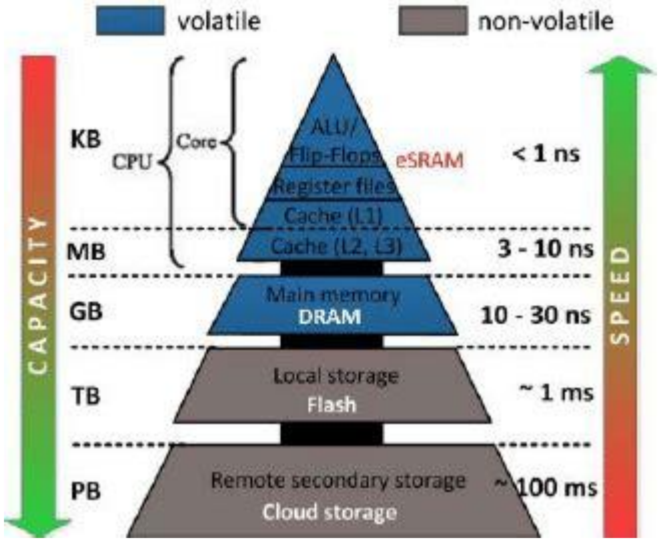
What is Graph Analytics?

- Graph Analytics is a form of data analysis used in many fields (business, financial, biological, social networks, etc.).
- Computes information in graph networks.
- Examples: PageRank algorithm.



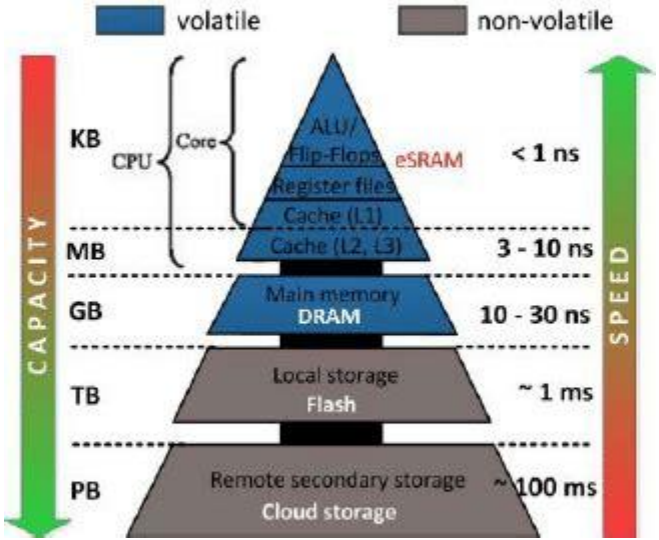
Caches Review

- Computer memory has many layers.
- The fastest access is in cache.
- The next fastest is main memory (DRAM).
- Software performance can be improved by utilizing the caches more.



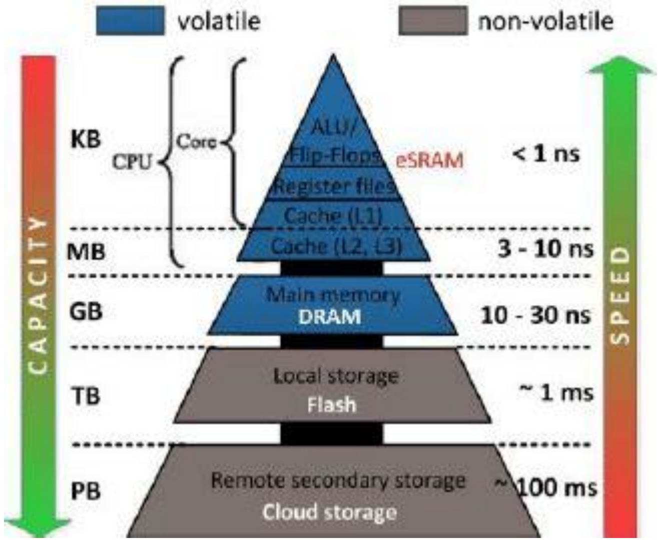
Problem Overview

- There are many existing optimized graph frameworks
  - GraphLab
  - Ligra
  - Galois
  - GraphMat
  - etc.
- The fastest frameworks have 60-80% of cycles stalled on memory access to DRAM.



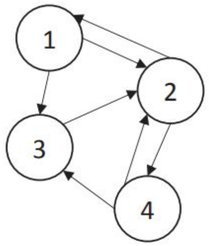
Problem Causes

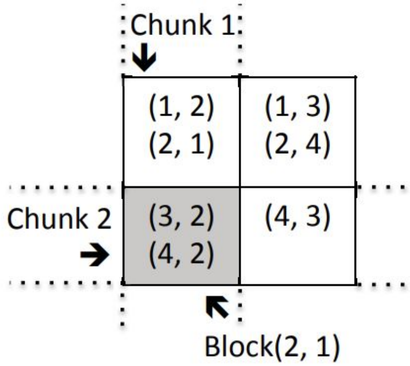
- The cache is not optimized aggressively (Might be using L3 cache and DRAM a lot, but not L1/L2).
- When we increase the number of cores, the performance does not scale well.
- The runtime overhead from running secondary computations is too high.



Problem Example: GridGraph

- Implementation:
    - Organizes edges into “grid” (rows determine source vertex, columns indicate destination vertex)
    - Computes data at vertex and streams to edges.
    - Applies updates instantaneously from edge streams.
- Problems:
    - Does not scale well beyond 4-6 cores due to cache contention





Problem Example: X-Stream

- Implementation
  - Performs computations from the edges of the graph
  - Keeps in-streams and out-streams partitioned to fit in cache to store updates
  - Streams the updates to the update in-stream
  - Shuffles the updates from the in-stream to corresponding destination out-streams
  - Applies the updates from the out-streams to corresponding vertices
- Problem
  - Incurs significant runtime overhead from shuffle and gather phase

Considerations

- Partition graph into smaller sections
    - 2D grid
    - Streaming Partitions
- Store in a certain data format
    - Sorted compressed graph
    - Unsorted edge list
- Exploit parallelism
    - Across single partition
    - Across multiple partitions
- Utilize entire cache system
    - L1, L2, shared LLC
- Minimize overhead incurred

Solution: Cagra

- Cagra is a novel graph analytic framework
- Attains speed-up over 2 times faster than the fastest frameworks at the time

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×)

Cagra Performance on PageRank compared to other frameworks

Solution: Cagra

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×)

Cagra Performance on Label Propagation compared to other frameworks

Solution: 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×)

Cagra Performance on Collaborative Filtering compared to other frameworks

Solution: Cagra

Dataset	Cagra	Ligra
LiveJournal	1.2s $(1\times)$	1.2s $(1.00\times)$
Twitter	14.6s $(1\times)$	17.5s $(1.19\times)$
RMAT 25	7.08s $(1\times)$	11.1s $(1.56\times)$
RMAT 27	21.9s $(1\times)$	42.8s $(1.95\times)$
SD	15.0$(1\times)$	19.7 $(1.31\times)$

Cagra Performance on Between Centrality compared to other frameworks

Cagra Overview

1. Cagra divides graph into subgraphs through compressed sparse row (CSR) segmenting in preprocessing
2. Cagra processes subgraphs in parallel
3. Intermediate results are locally merged and stored in buffers
4. Parallel cache-aware merge is used to combine buffers within L1 cache

Compressed Sparse Row (CSR) Segmenting

Motivation: Page Rank

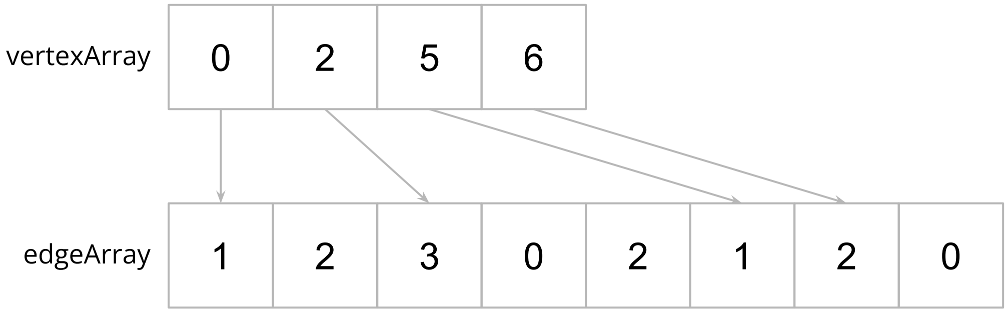
- Each vertex (destination) computes rank based on neighbors (sources)
- Common pattern seen in graph algorithms (Collaborative Filtering, Betweenness Centrality)

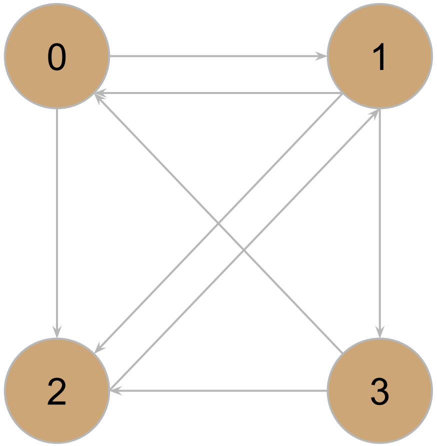
Algorithm 1 PageRank

```
procedure PAGERANK(Graph G)
    parallel for v : G.vertexArray do
        for u : G.edgeArray[v] do
            G.newRank[v] +=
                G.rank[u] / G.degree[u]
        end for
    end parallel for
end procedure
```

CSR Format

- vertexArray with O(V) length
- edgeArray with O(E) length
- Application-specific data in separate array





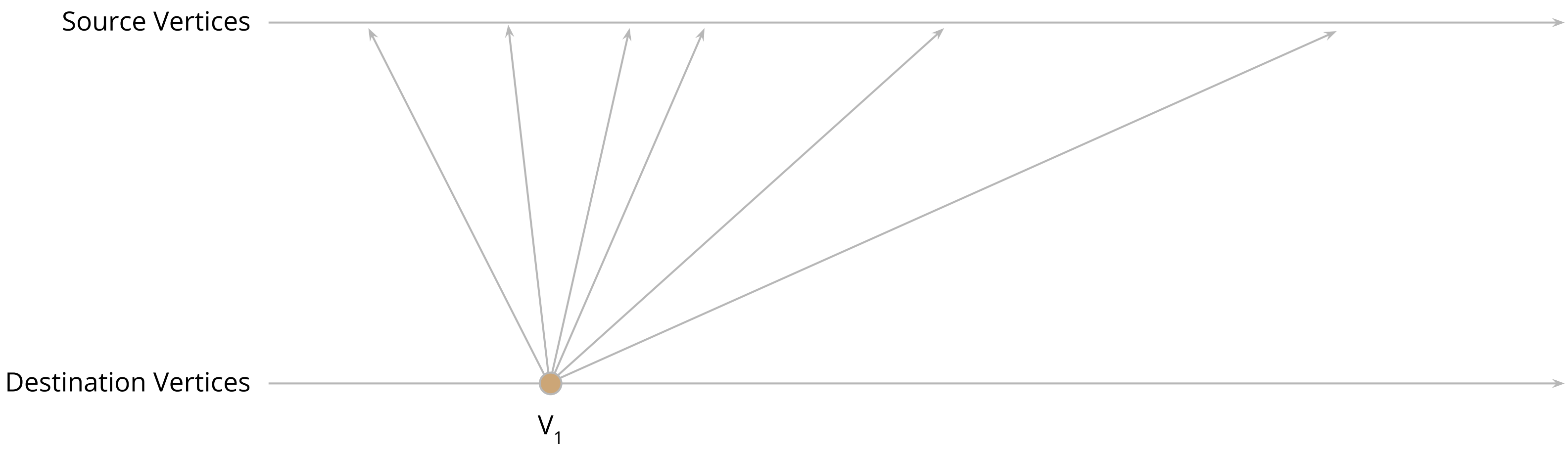
Problem: Random reads

- Each vertex, v, accesses neighbors, u
- Can't predict u, so each read to rank and degree is random
- Bad use of cache

Algorithm 1 PageRank

```
procedure PAGERANK(Graph G)
    parallel for v : G.vertexArray do
        for u : G.edgeArray[v] do
            G.newRank[v] +=
                G.rank[u] / G.degree[u]
        end for
    end parallel for
end procedure
```

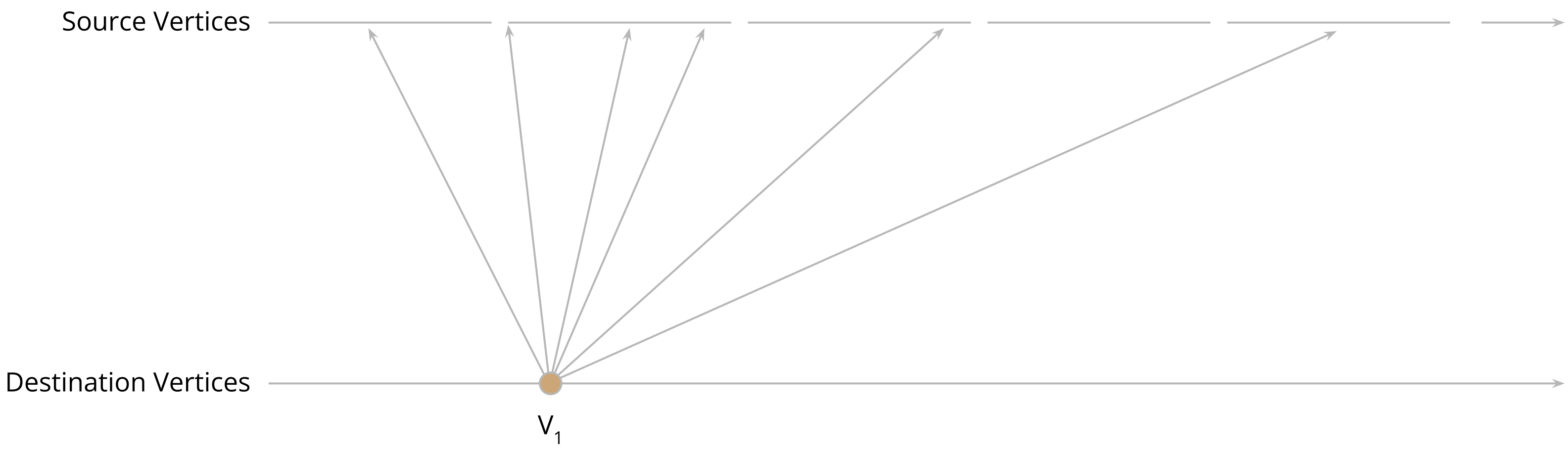
Illustration



CSR Segmenting

- Breaks up graph into cache-sized segments of vertex data (preprocessed)
- Performance is scalable across all cores
- Incurs low runtime overhead

Illustration



Preprocessing

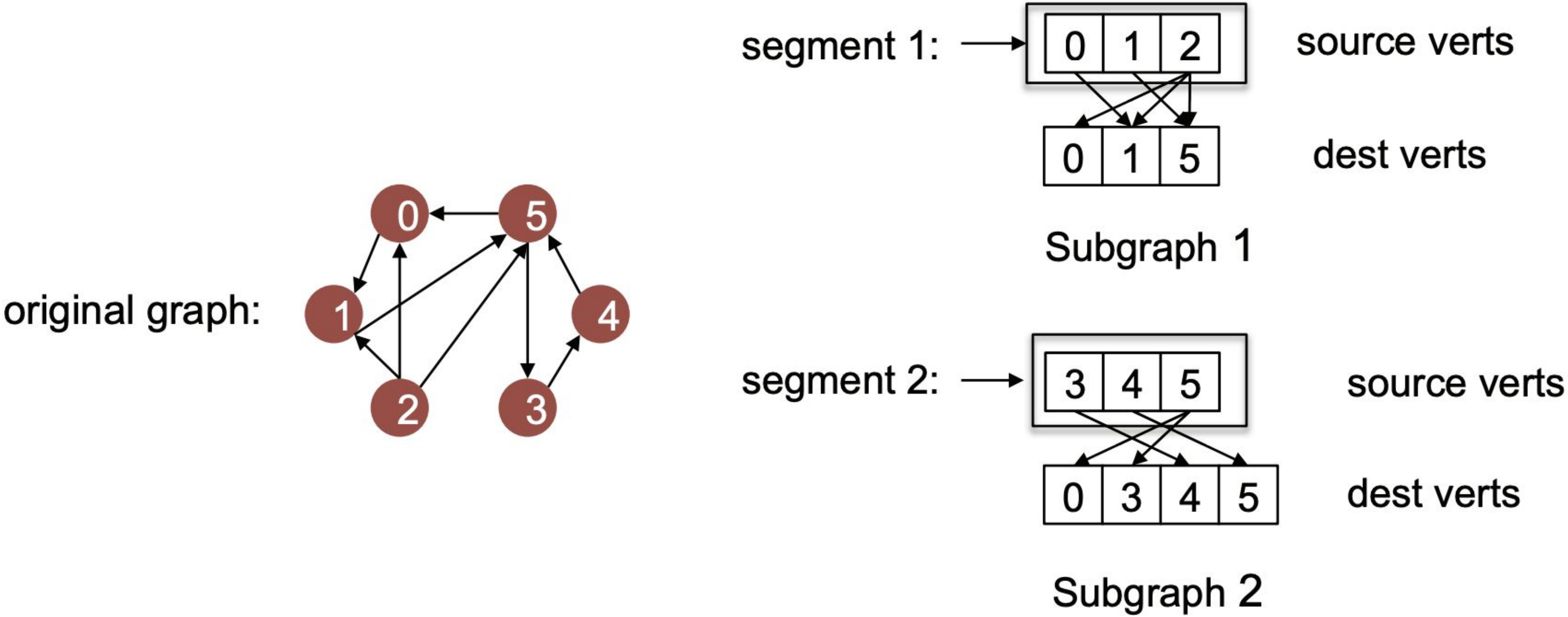
- Breaks graph into several subgraphs based on segments
- Segments contain
  - Idx map from local to global
  - Intermediate buffer
  - BlockIndices for merge

Algorithm 2 Preprocessing

Input: Number of vertices per segment N, Graph G

for $v : G.vertices$ **do**
    for $inEdge : G.inEdges(v)$ **do**
        $segmentID \leftarrow inEdge.src/N$
        $subgraphs[segmentID].addInEdge(v, inEdge.src)$
    end for
end for
for $subgraph : subgraphs$ **do**
    $subgraph.sortByDestination()$
    $subgraph.constructIdxMap()$
    $subgraph.constructBlockIndices()$
    $subgraph.constructIntermBuf()$
end for

CSR Segmenting



Parallel Segment Processing

- Parallelism exploited on single large segment
  - Threads share same working set
  - More threads does not create cache contention
- In comparison to multiple smaller segments
  - Smaller segment’s working set fit in L2 cache
  - Merging overhead becomes bottleneck

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

Comparison with 2D Partitioning

- Cagra partitions only on source vertices
- Benefits:
    - This produces less subgraphs, leading to better scalability when processing
    - This leads to a faster merge since there are less subgraphs to merge in the end

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
```

Parallelism Across Vertices

- Parallelism only done across vertices, not within single vertex
  - Takes advantage of CSR format
  - No need for atomics for synchronization
  - Updates to each vertex merged locally by same worker thread

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
```

Cache-aware Merge

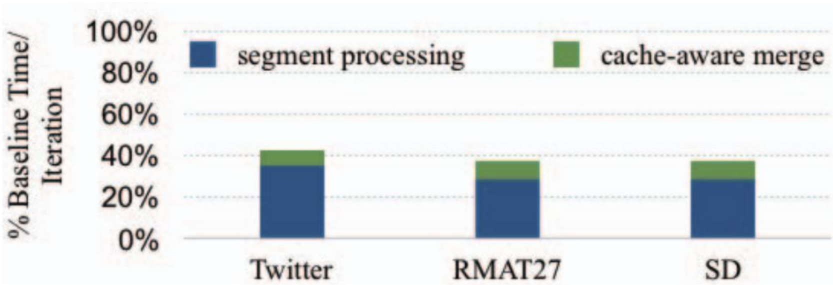
- After computation, we need to merge results
- IntermBufs are merged into one dense output vector
- The buffers are accessed sequentially
- Range of Vertex IDs is divided into L1-cache-sized blocks

Algorithm 4 Cache-Aware Merge

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

Cache-aware Merge Results

- The cache-aware merge algorithm has small runtime overhead



CSR Segmenting Results

- Improved cache utilization, accesses to DRAM sequential
- Scalability
  - Threads can parallelize execution within subgraphs
  - No need for atomic operations or synchronization
  - Merge phase can be parallelized
- Low overhead
  - Cache-aware merge requires little extra sequential memory accesses
  - Merges in L1 cache in parallel
  - Single sequential pass through edges
- Easy to use
  - Applies to a large variety of algorithms

Segment Size Tradeoff

- As seen, the Cagra framework sees a tradeoff with segment size
- Smaller segments
  - Fit into lower level cache
  - Reduced random access latency
  - Incur more overhead from merges for same destination
- Authors found sizing segments to fit in L3 cache led to best tradeoff

Frequency-Based Reordering

- Cagra reorganized source vertices based on frequency
    - Number of out-edges
- Higher frequency -> Faster higher level cache
- Cluster vertices with above average out degree
- Parallel stable sort
- Indices mapped
- Vertices updated in EdgeArray
- Tasks may spawn subtasks

Evaluation: Traffic between LLC and DRAM

- Segment Processing
  - Cagra reads in V source vertex data
  - Writes qV intermediate updates (q is average number of vertices adjacent to a segment)
  - Goes through all edges once
  - Incurs E + qV + V traffic total
- Cache-aware Merge
  - Reads all intermediate buffers (qV)
  - Writes V final values
  - Incurs qV + V traffic
- Total
  - In total, Cagra sees E + 2qV + V traffic to DRAM

Evaluation: Traffic between LLC and DRAM

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

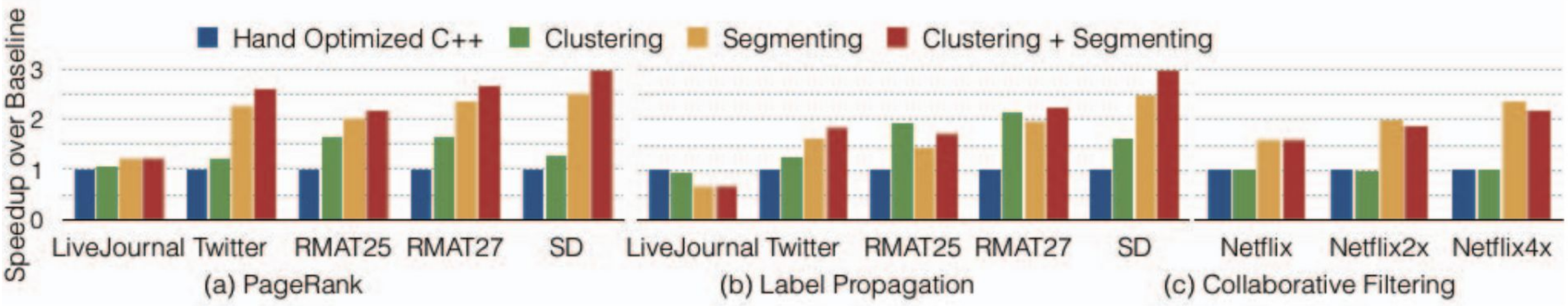
TABLE VII: Comparisons with other frameworks optimized for cache. E is the number of edges, V is the number of vertices, q is the expansion factor for our techniques, P is the number of partitions for GridGraph, K is the expansion factor for X-Stream. On Twitter graph, $E = 36V, q = 2.3, P = 32$.

Evaluation: Comparison

- Experiments run on dual socket system with Intel Xeon E5-2695 v2 CPUs
  12 cores for total of 24 cores and 48 hyperthreads
- 30 MB last level cache in each socket
- 128GB DDR3-1600 memory
- Transparent Huge Pages (THP) enabled

Evaluation: Speedup and Cache Misses

- CSR Segmenting
  - Saw more than 2x speedup in PageRank, Label Propagation and Collaborative Filtering
  - Eliminated random DRAM accesses
  - LLC miss rate dropped from 46% to 10% on Twitter graph



Open Questions

- A natural question that arises is how we can improve Cagra to be cache-oblivious in its merge algorithm