Making Caches Work for Graph Analytics

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

Reviewed by Miranda Cai

Graph Frameworks Are Limited

- Current graph frameworks do not reach full hardware potential
- Some frameworks store on disk
  - High overhead
- Others store in memory
  - Every access is a random access to DRAM
  - Not cache optimized
- 60-80% of cycles are stalled on memory access

Cagra

Idea: A graph framework that fully utilizes the cache to **eliminate all DRAM random accesses** and make **all DRAM accesses sequential**.

Main Contributions:

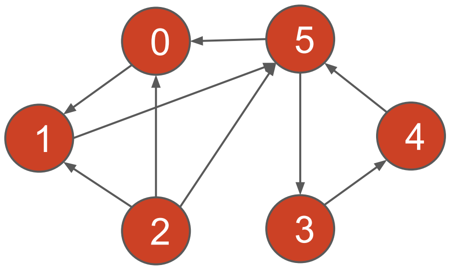
- CSR Segmenting
  - Partitioning system
- Cagra Framework
  - Ex: PageRank application
- Performance Benefits

Preprocessing using CSR Segmenting

Algorithm 2 Preprocessing

Input: Number of vertices per segment N, Graph G

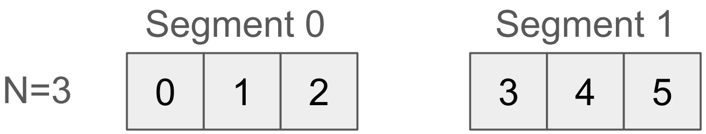
for $v : G.vertices$ **do**
    for $inEdge : G.inEdges(v)$ **do**
        $segmentID \leftarrow inEdge.src/N$
        $subgraphs[segmentID].addInEdge(v, inEdge.src)$
    end for
end for
for $subgraph : subgraphs$ **do**
    $subgraph.sortByDestination()$
    $subgraph.constructIdxMap()$
    $subgraph.constructBlockIndices()$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

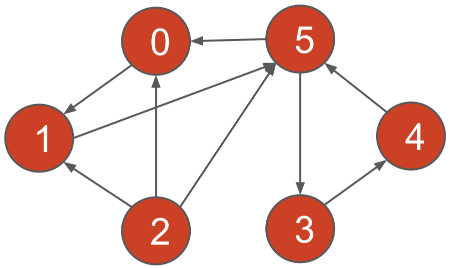


Preprocessing using CSR Segmenting

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

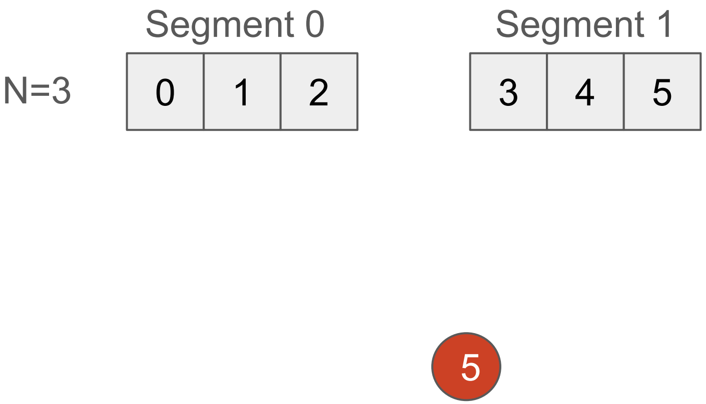


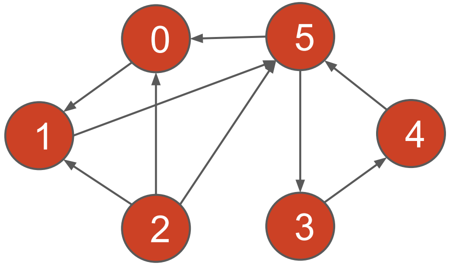


Preprocessing using CSR Segmenting

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

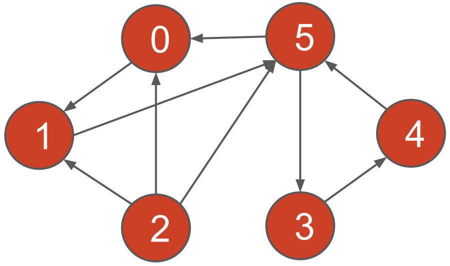


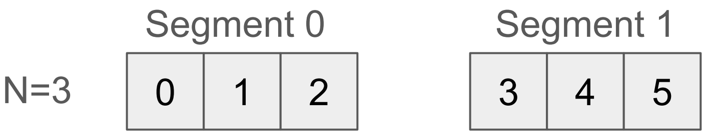


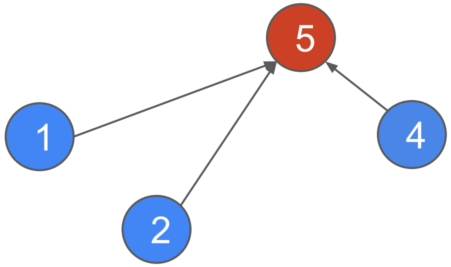
Preprocessing using CSR Segmenting

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



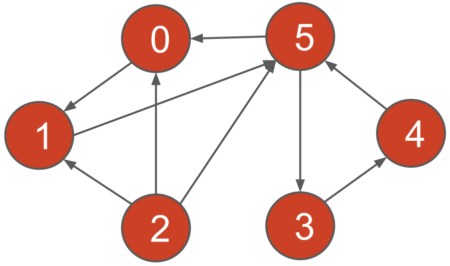


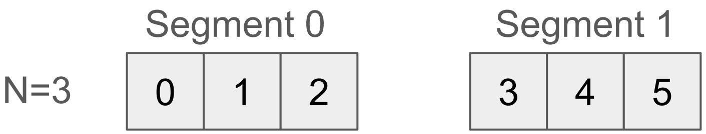


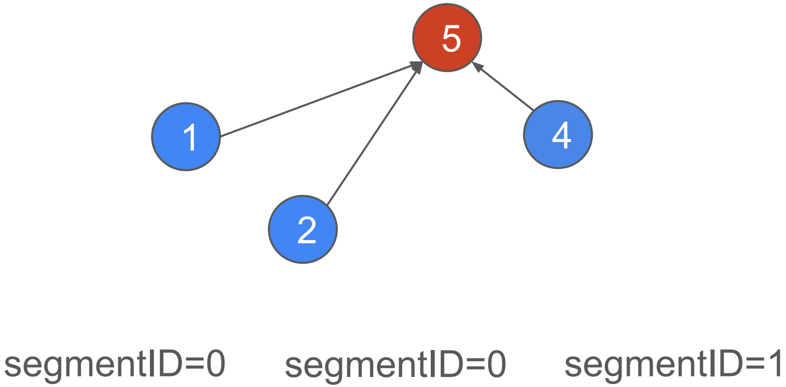
Preprocessing using CSR Segmenting

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
```





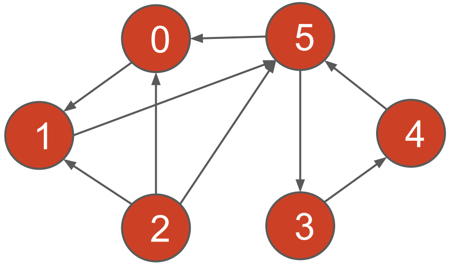


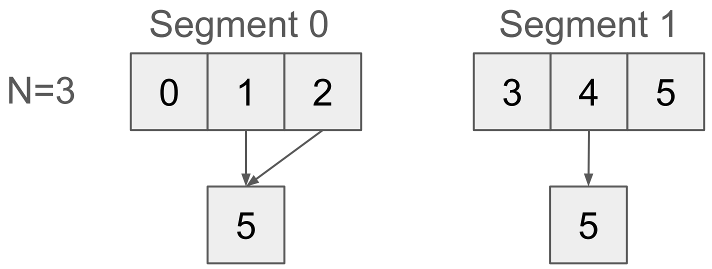
Preprocessing using CSR Segmenting

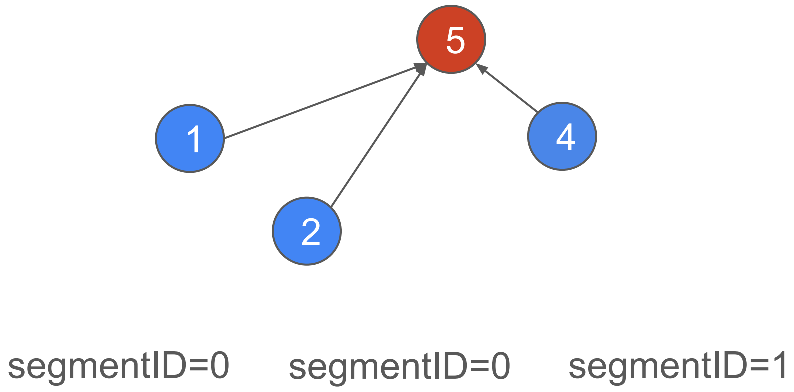
Algorithm 2 Preprocessing

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
```



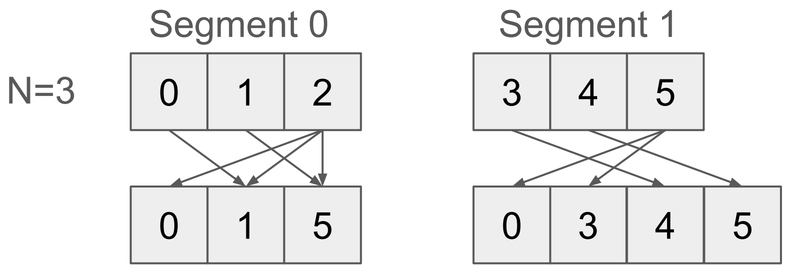


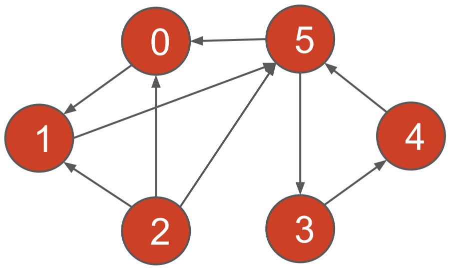


Preprocessing using CSR Segmenting

Algorithm 2 Preprocessing

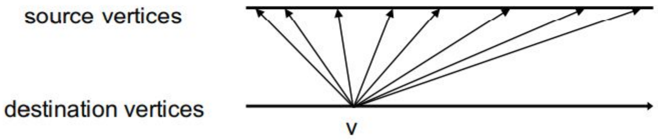
Input: Number of vertices per segment N, Graph G
for $v : G.vertices$ **do**
    for $inEdge : G.inEdges(v)$ **do**
        $segmentID \leftarrow inEdge.src/N$
        $subgraphs[segmentID].addInEdge(v, inEdge.src)$
    end for
end for
for $subgraph : subgraphs$ **do**
    $subgraph.sortByDestination()$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



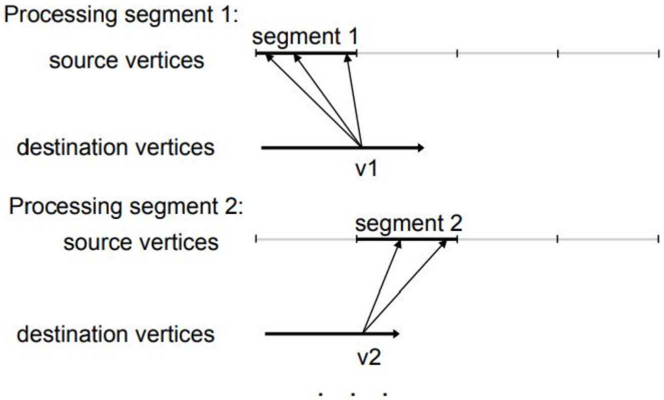


CSR Segmenting Cache Benefits

Without segmenting, need to load all source vertices



With segmenting, load segments that fit into a cache



Processing Segments in Parallel

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

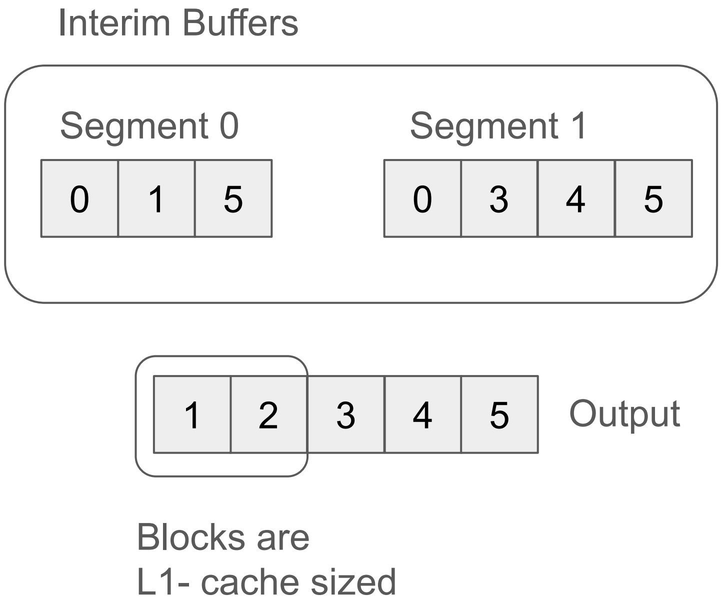
Process in segments since each segment fits in cache. Then every vertex within the same segment share the same working set.

Return: Fills up $subgraph.interimBuf$ with processed edges.

Merge Interim Buffers into Final Output

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 $output$



Segment Size Selection

- Trade-off when choosing segment size
  - Smaller segments → Lower random access latency, More interim buffer merges
  - Larger segments → Higher random access latency, Less interim buffer merges
- Experiments show L3 cache (LLC) is the best
- Expansion factor metric

$$q = s_{\text{adj}} / s$$

where s = no. of vertices per segment, s_{adj} = avg no. of edges to segment

q describes avg no. of segments that contribute data to each vertex, which is same as the no. of merges per vertex

Memory Access Costs Analysis

k	segments
q	expansion factor
V/k	no. source vertices per segment
qV/k	no. interim buffer updates per segment

Memory Access Costs Analysis

k	segments
q	expansion factor
V/k	no. source vertices per segment
qV/k	no. interim buffer updates per segment

Phase 1 Traffic: $E + V + qV$

Phase 2 Traffic: $V + qV$

Memory Access Costs Analysis

k	segments
q	expansion factor
V/k	no. source vertices per segment
qV/k	no. interim buffer updates per segment

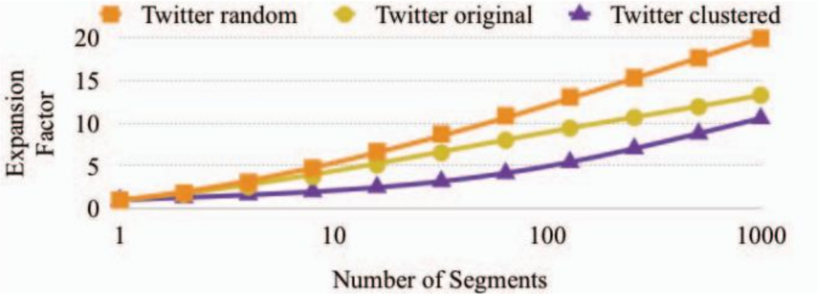
Phase 1 Traffic: $E + V + qV$

Phase 2 Traffic: $V + qV$

Total Traffic: $\boldsymbol{E + 2qV + 2V}$

Frequency Based Clustering

- Before CSR segmenting, reorder the vertices such that high degree vertices are clustered together
- Only vertices with degree > avg degree get clustered
- Most of the original locality is preserved
- The advantages:
  - Most graphs follow power-law degree distribution
  - Better cache-line utilization
  - Keep frequently accessed vertices in fast cache



PageRank Algorithm

Algorithm 5 PageRank in Cagra

typedef double vertexDataType
$contrib \leftarrow$ {1/outDegree[v], ...}
$newRank \leftarrow$ {0.0, ...}

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

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

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

procedure PAGERANK($G, maxIter$)
  $iter \leftarrow 0$
  $A \leftarrow V$
  while $iter \neq maxIter$ **do**
    $A \leftarrow EdgeMap(G, A, EdgeUpdate, EdgeMerge)$
    $A \leftarrow VertexMap(G, A, VertexUpdate)$
    $Swap(contrib, newRank)$
    $iter \leftarrow iter + 1$
  end while
end procedure

Example of easy to implement algorithm using Cagra Interface, some ideas borrowed from Ligra.

Evaluation

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

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

Cagra shows up to 5x speed up against the most competitive existing frameworks, and performs better on larger graphs.

Evaluation

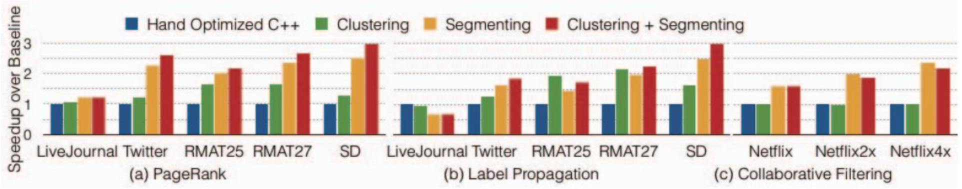


Fig. 7: Speedups of optimizations on PageRank, Label Propagation, Collaborative Filtering

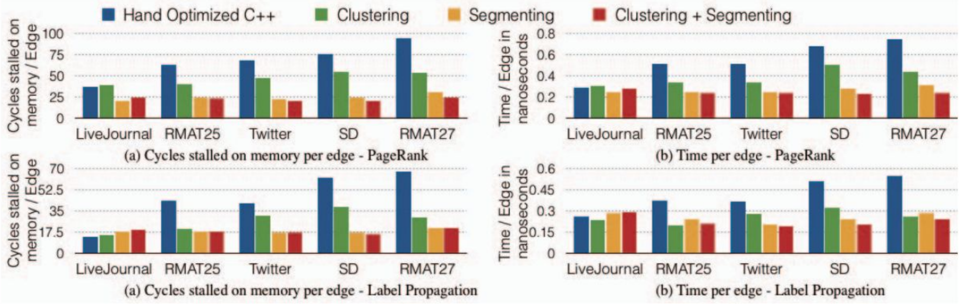
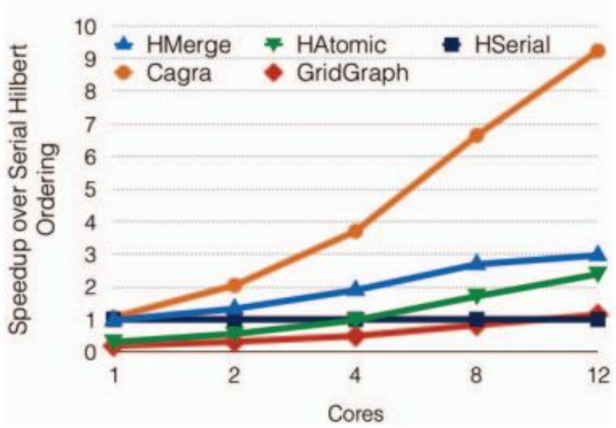
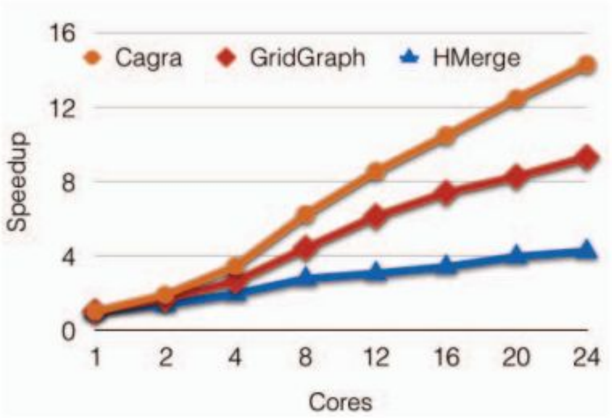


Fig. 8: Cycles stalled on memory and time per edge for PageRank and Label Propagation. Cycles stalled per edge for Clustering + Segmenting is low and stable across graphs with increasing sizes, demonstrating that random accesses are confined in LLC.

Segmenting on its own already provides 2x speedup. Cycles stalled on memory per edge increases for graph size on Hand Optimized C++, but stays consistent for Cagra with segmenting.

Evaluation





Cagra is much more scalable than other cache optimized frameworks like GridGraph and Hilbert ordering ones.

Evaluation

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, $E = 36V, q = 2.3, P = 32$.

Preprocessing time is insignificant. GridGraph's preprocessing time was up to 9-11x slower than Cagra's.

Comparison to Existing Models

- GridGraph, X-Stream
  - Use 2D partitioning into subgraphs
  - Some subgraphs can be small → bad scalability
  - High overhead run-times
- Disk-based systems (GraphChi)
  - Slow compared to cache optimizations
- Distributed Systems
- Hilbert Ordering
  - Edge traversal method
  - Cache contention → bad scalability

Conclusion

Strengths

- Novel Approach and optimizations that meshed well together
- Very in depth evaluation

Weaknesses

- Only algorithms with certain features were used for comparison

Future Directions

- Introducing more parallelism
- Minimizing preprocessing time