Making caches work for graph analytics

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

Historical Context

Paper at 2017 Paper @ BigData

Yunming Zhang (Google TPU compiler stuff; MIT PhD 2020, adv. Julian, Saman)

Vladimir Kiriansky (VMWare; MIT B.Sc., M.Eng. 2003, PhD 2019, adv. Saman)

Charith Mendis (Prof. UIUC, Tensor compiler stuff, MIT PhD 2020, adv. Saman)

Saman Amarasinghe (Prof @ MIT for 25+y; Compiler, system stuff)

Matei Zaharia (Prof @ MIT->Stanford->UCB 10y, Billionaire, Cofounder & CTO of Databricks, PhD @ Stanford 2013, Creator of Apache Spark)

William Hasenplaugh (D.E.Shaw Research, MIT PhD 2016, adv. Charles Eric Leiserson)

Julian Shun (Hi!)

Where we’re at in class

- Graph
- Non-Graph

- Distributed
- (Single machine) Parallelized
- External (storage) algorithm
- **Cache level optimization**

Motivation

Graph algorithms are irregular

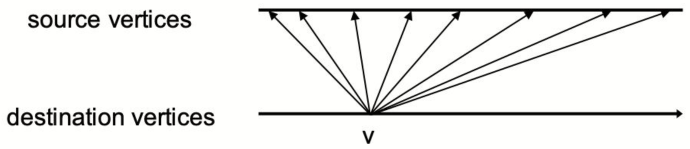
- Lots of random memory access
- Hard to utilize cache well

PageRank on V=41M, E=1.5B (~656MB)

- LLC (L3, ~30-55MB) miss rate **>45%**
- **60-80%** cycles spent stalled for memory access

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
```



Objective & Overview

Build ‘Cagra’ framework

EdgeMap (G, ActiveFrontier, EdgeUpdate, Merge)

VertexMap (G, VertexSubset, VertexUpdate)

How?

CSR Segmenting - Split the graph into subgraphs, to fit in cache

Frequency Based Clustering - Reorder CSR to maximize ‘local’ edges

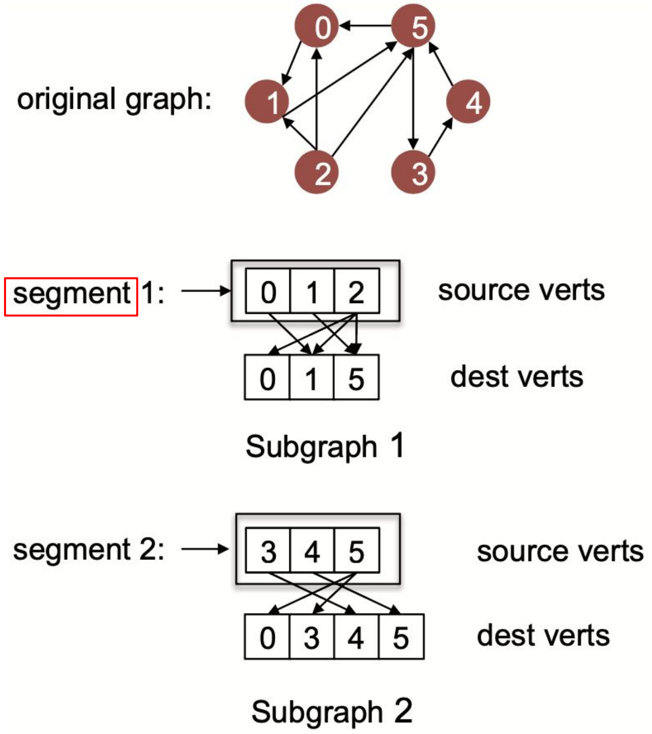
CSR Segmenting – Split Subgraph

Given (inEdge) CSR, split edges by **source**

Keep all edges (and destinations) in subgraph

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



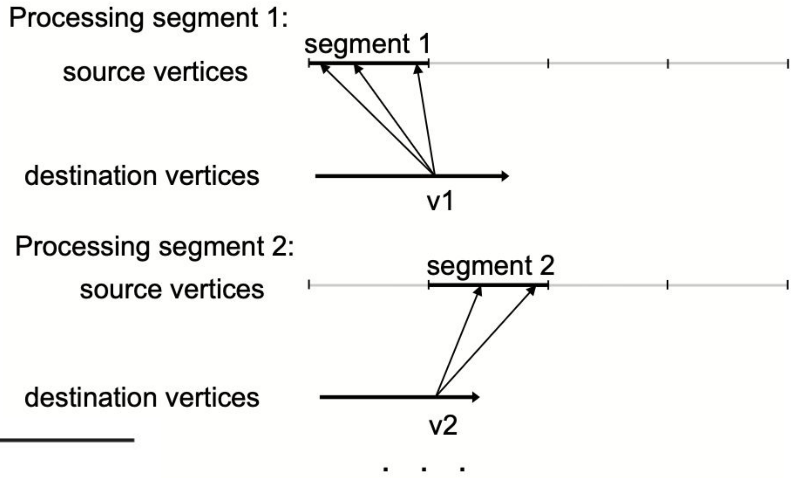
CSR Segmenting – Per-Segment Processing

For each subgraph, parallelize on destination

All sources in segment fits in cache

inEdges are in order of destination

Keep results for each dest in each subgraph



Algorithm 3 Parallel Segment Processing

for $subgraph : subgraphs$ **do**
  $\boxed{\textbf{parallel for}}$ $v : subgraph.Vertices$ **do**
    for $inEdge : subgraph.inEdges(v)$ **do**
      Process $inEdge$
    end for
  end parallel for
end for

CSR Segmenting – Merging

Each dest have multiple partial results

- Merge results from subgraphs
- Destination blocks fit in L1 Cache

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
```

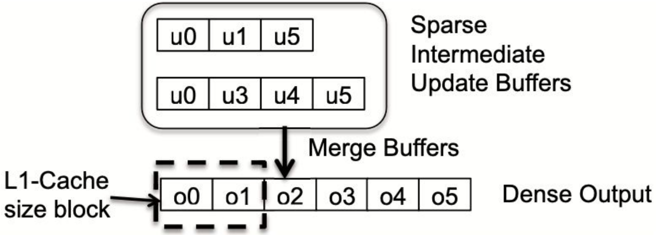


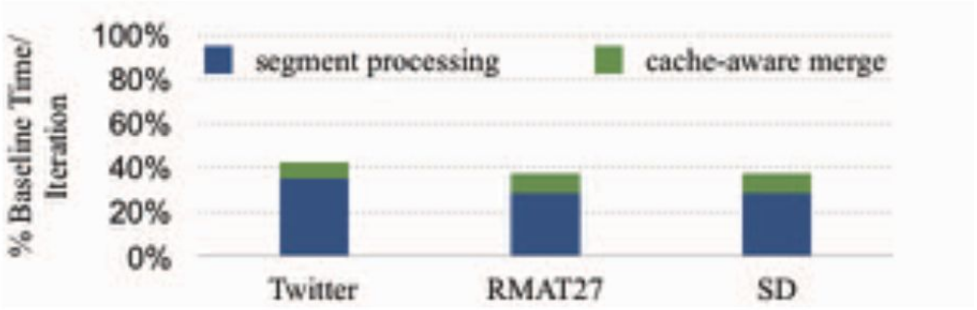
Fig. 4: Cache-aware merge

Parameters & Analysis

Source segments to fit in LLC (L3 Cache)
  For PageRank, 30MB LLC can fit 4M vertices

How many merge happens for each vertex?
  Expansion Factor q = (average inDeg for segment) / (average |V| for segment)
  Rough proxy for number of merges

LLC <> DRAM traffic
  k segments, expansion factor q
  E + 2qV + V



Frequency Based Clustering

Motivation:

want to maximize edges within subgraph,

reduce inter-segment merging (=extra work, DRAM access)

Stable sort vertices based on [outdegree/threshold]

Cluster hot nodes together, while keeping some natural order

Got 2x higher expansion factor, reducing merges

Experimental Evaluation

5x faster than papers, 3x than engineers

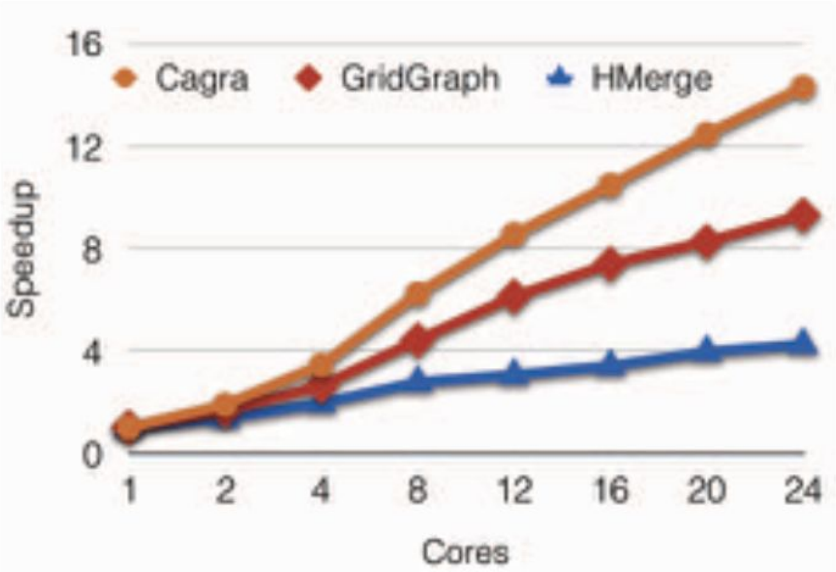


Fig. 10: Scalability for PageRank on Twitter

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
```

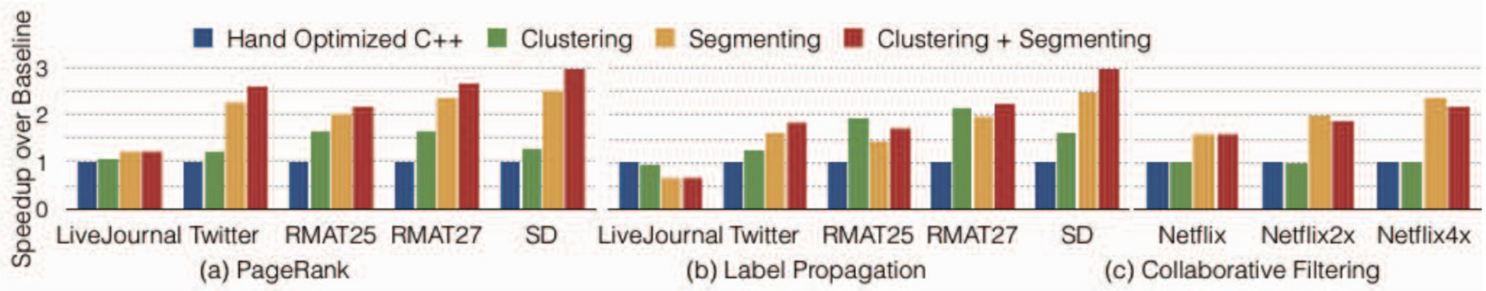
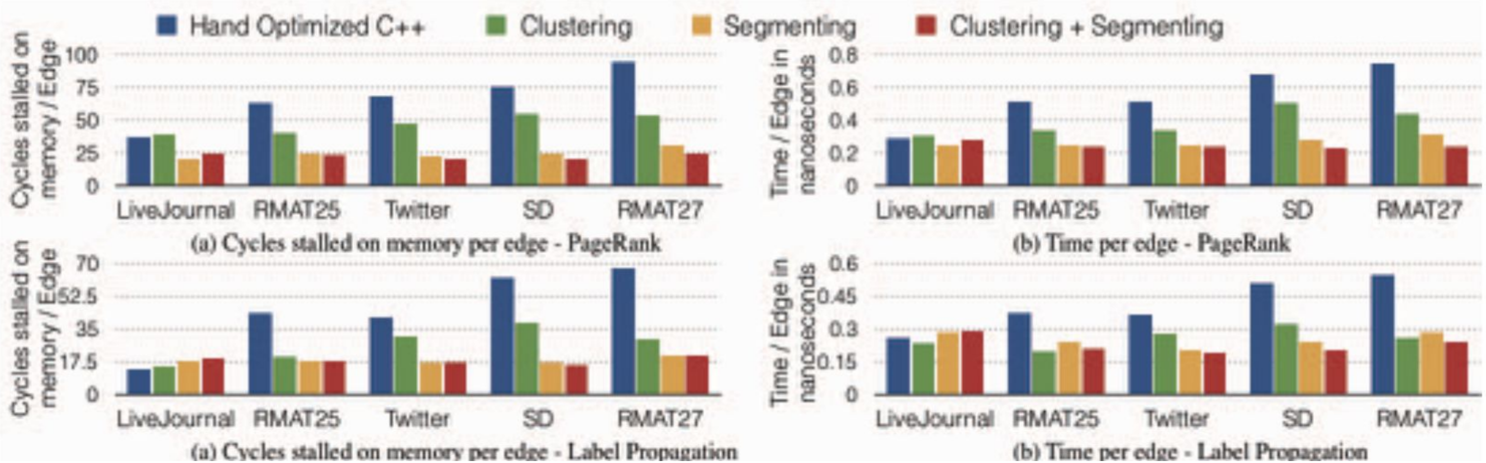


Fig. 7: Speedups of optimizations on PageRank, Label Propagation, Collaborative Filtering



Reviewing the paper

No push based (atomic?)

e.g. For BFS, we can’t do dense method

No destination vertex condition check

  Very good for PageRank and algos using most edges most times

Can be used in distributed systems as well?

Merge step might be inconvenient