Making Caches Work for Graph Analytics

Yunming Zhang, Vladimir Kiriansky, Charith Mendis,
Saman Amarashinghe, Matei Zaharia
MIT, Stanford
BIGDATA 2017

Motivation

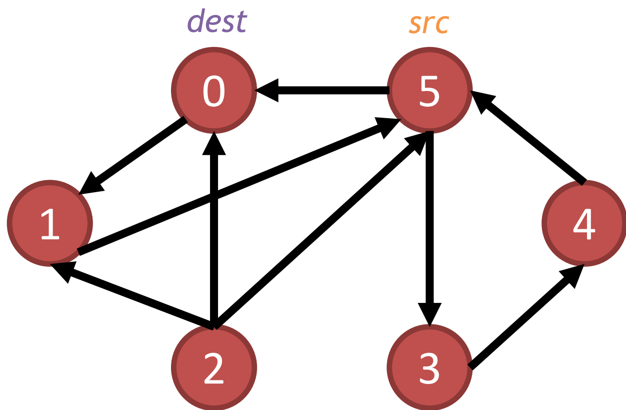
- Many graph algorithm frameworks do not focus on improving cache utilization
- As a result, graph algorithms frequently hit memory wall, resulting in low performance
- Graphs with power-law degree distribution introduces even lower cache locality, further hurting performance

This paper

- Proposes CSR segmenting methodology to constrain most of the random accesses in last level cache (LLC) by segmenting a big graph into several subgraphs with shared working set
- Extends on existing programming interface to allow CSR segmenting implementation
- Proposes frequency clustering optimization on memory data layout to further reduce memory traffic
- Shows up to 11x speedup comparing to exsiting distributed graph processing framework for popular graph processing applications

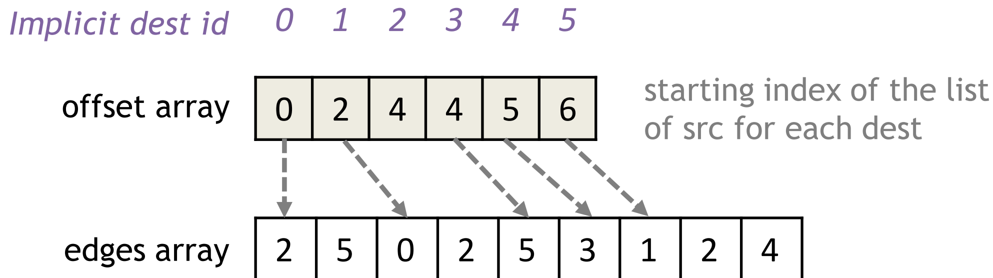
Sparse Graphs Represented in CSR Format

Graph Representation



V vertices
E edges

Compressed Graph in
Compressed Sparse Row (CSR) Representation



starting index of the list of src for each dest

Distributed Graph Algorithms Have Poor Locality

Example Distributed PageRank Algorithm

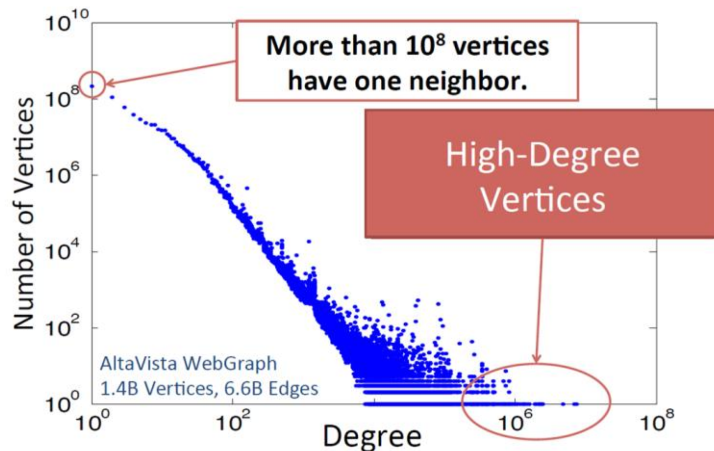
```
procedure PageRank(Graph G)
  parallel for v: G.offsetArray()
    for u: G.edgeArray[v]
      newRank[v] += G.rank[u]/G.degree[u]
  if G.rank[:] == newRank[:] return
  else
    G.rank[:] = newRank[:]
    PageRank(G)
```

Access Patterns

1. offset array: random
2. edge array: globally random, locally sequential
3. new rank array: random
4. rank array: random
5. degree: random

} app specific working set

Power-Law Degree Distribution



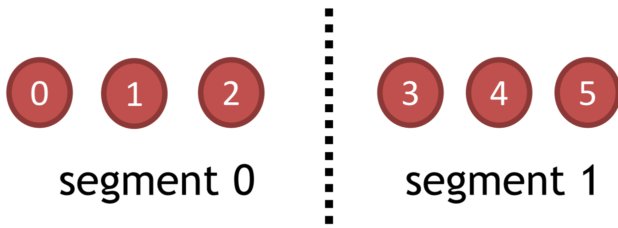
Low degree vertices -> high % of random accesses
High degree vertices -> too much data in working set to fit in cache
Bad Cache Performance
60%-80% of the CPU cycles are stalled

CSR Segmenting

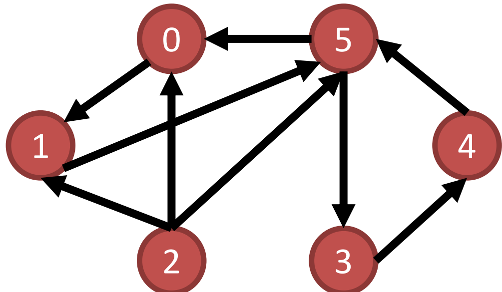
- Goal: divide the large graph into subgraph that fits in cache, perform distributed processing on each subgraph
- Benefits:
  - Improved cache unitization
    - One time DRAM loading, and then all reads and writes are in cache
  - Great scalability
    - Ample parallelism allowed within each subgraph
  - Low overhead
    - Subgraph merging only needs a small amount of extra sequential accesses
  - Widely applicable
    - Provide a clean API for implementing algorithms that needs subgraph aggregations

Step1: Preprocessing (graphical)

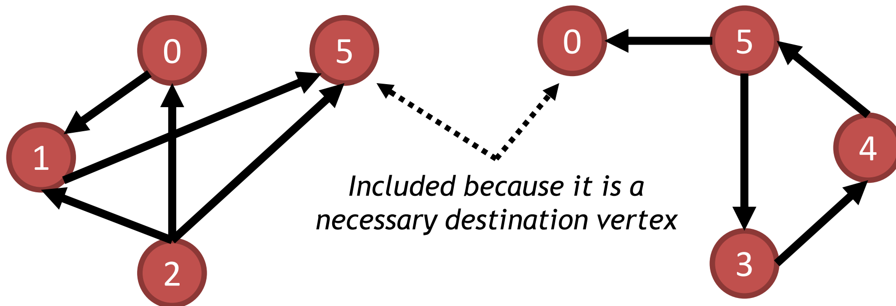
- Divide vertices into segments, such that data for each segment fit into cache



- Divide the graph into subgraph, so that the source vertices in each graph only belong to one segment



Original graph

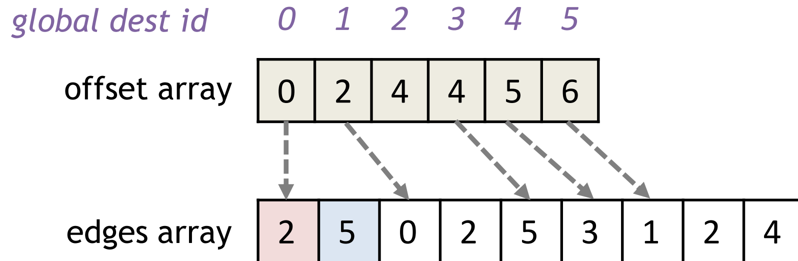


Subgraph 0

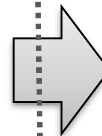
Subgraph 1

Step1: Preprocessing (CSR specific)

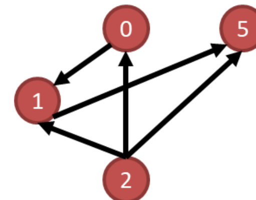
- Realization with the CSR graph representation
  - Construct CSR representations for subgraphs using the original graph CSR arrays



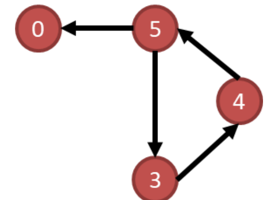
Original CSR Representation



Subgraph0
CSR

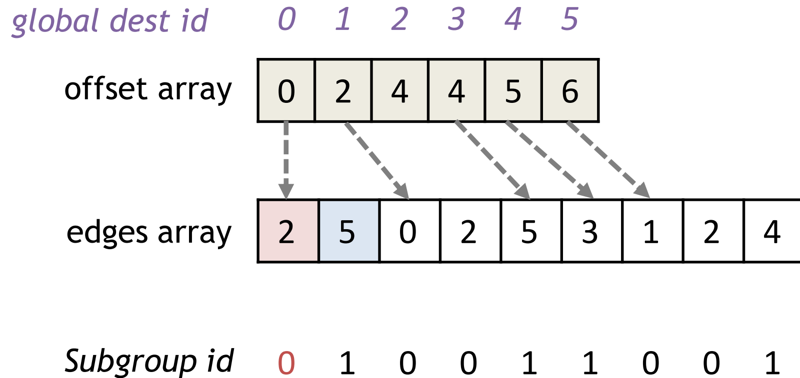


Subgraph1
CSR



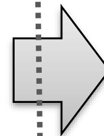
Step1: Preprocessing (CSR specific)

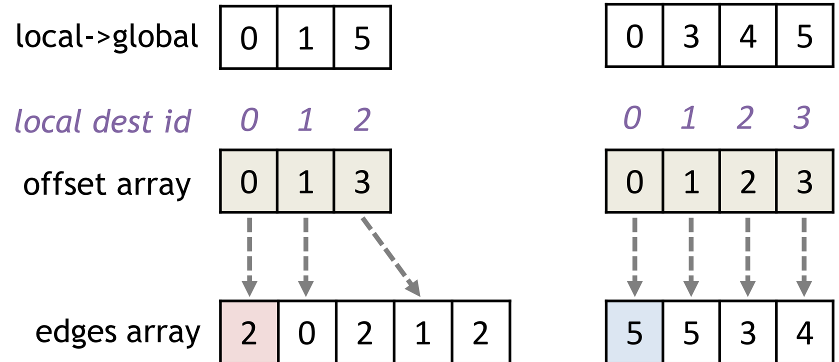
- Realization with the CSR graph representation
  - Construct CSR representations for subgraphs using the original graph CSR arrays



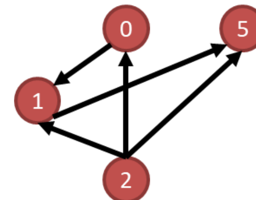
N = 3 (number of vertex in each segment)
Source id: 2, 5, …
Subgroup id: $\lfloor 2/3 \rfloor = 0$, $\lfloor 5/3 \rfloor = 1$, …

Original CSR Representation

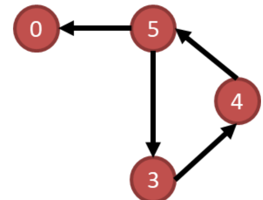




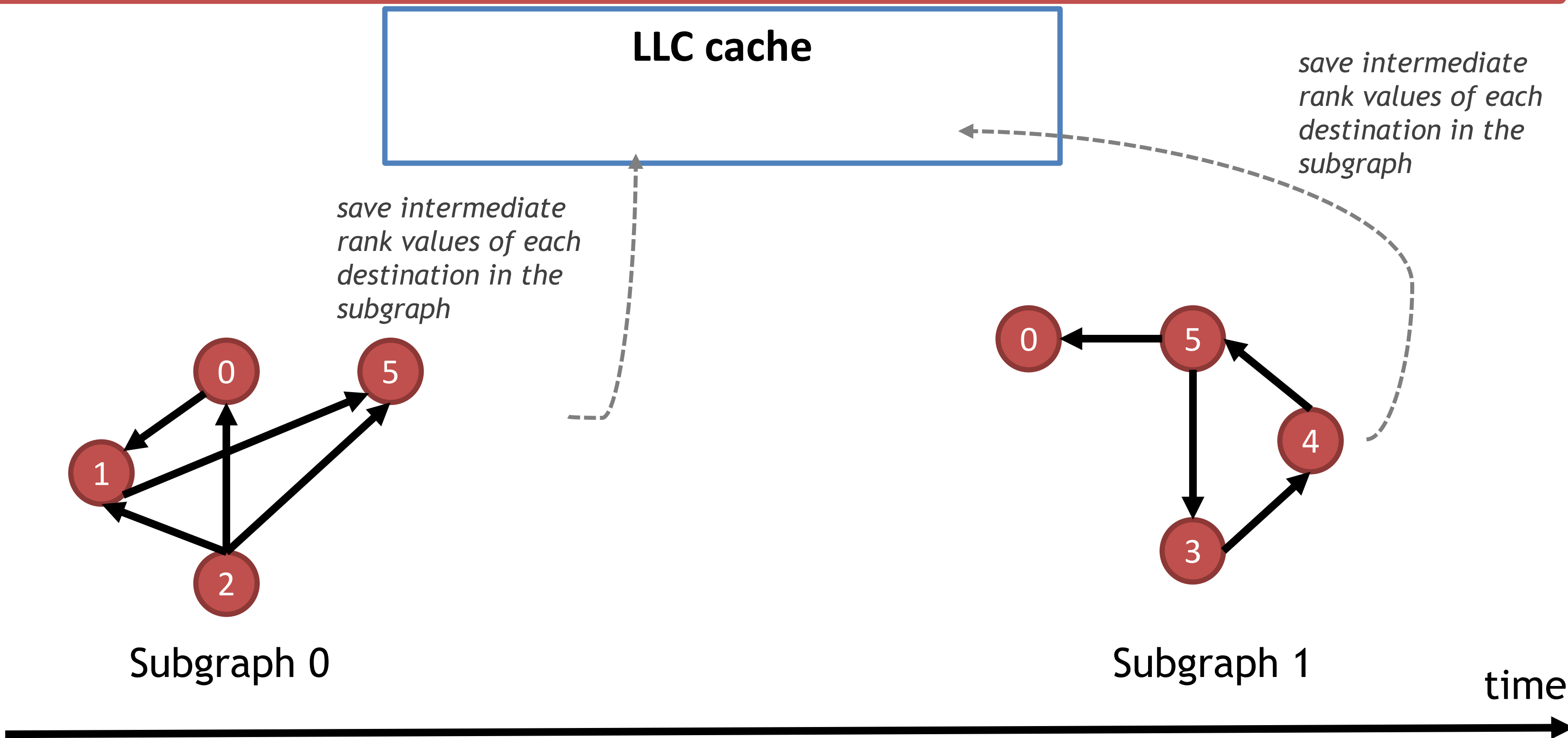
Subgraph0 CSR



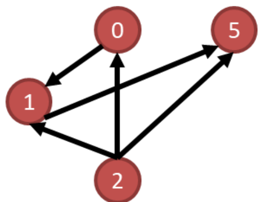
Subgraph1 CSR



Step2: Processing Subgraphs



Step2: Processing Subgraphs



LLC cache

working set arrays for
vertices 0, 1, 2

thread 0 0 E: 2->0

thread 1 1 E: 0->1 E: 2->1

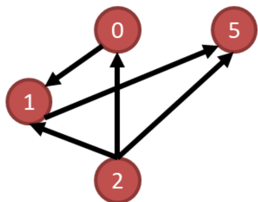
thread 2 5 E: 1->5 E: 2->5

- all of the destination vertices share the same set of source vertices (thus same working set)
- partitioning via source vertices allows ample parallelism inside each subgraph
- each thread works on a distinct (set of) destination vertices, so no synchronization needed

Subgraph 0

time

Step2: Processing Subgraphs



LLC cache

subgraph 0
interm. results

save intermediate rank values of each destination in the subgraph

thread 0 0 E: 2->0

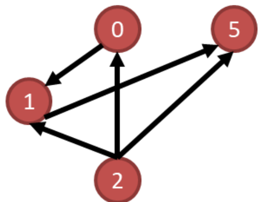
thread 1 1 E: 0->1 E: 2->1

thread 2 5 E: 1->5 E: 2->5

Subgraph 0

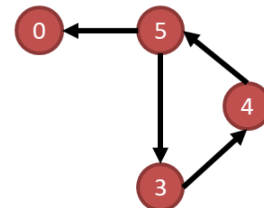
time

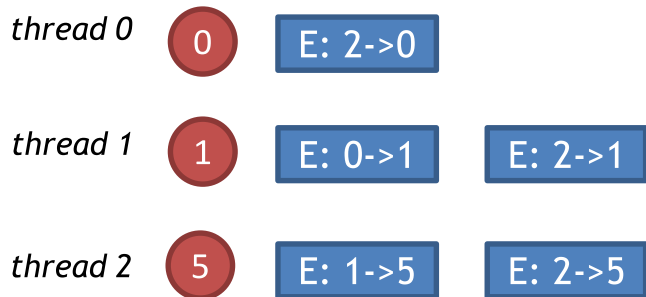
Step2: Processing Subgraphs

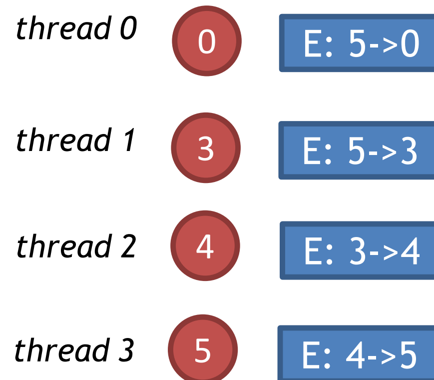


LLC cache

working set arrays for
vertices 3, 4, 5





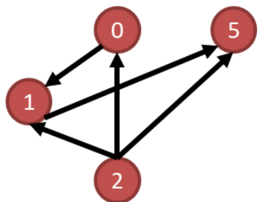


Subgraph 0

Subgraph 1

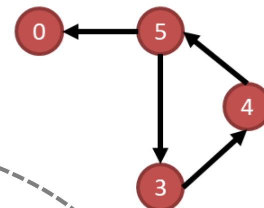
time

Step2: Processing Subgraphs



LLC cache

subgraph 1
interm. results



thread 0 0 E: 2->0

thread 1 1 E: 0->1 E: 2->1

thread 2 5 E: 1->5 E: 2->5

thread 0 0 E: 5->0

thread 1 3 E: 5->3

thread 2 4 E: 3->4

thread 3 5 E: 4->5

Subgraph 0

Subgraph 1

time

Step3: Cache-aware Merge

- The previously constructed global-to-local ID mapping allows each intermediate result to sync with the global indexing of the destination vertices

local->global

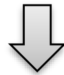
0	1	5

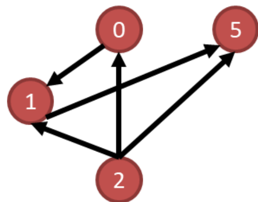
local dest id *0 1 2*

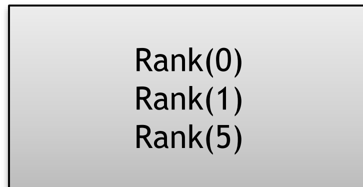
0	3	4	5

0 1 2 3

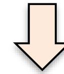
subgraph 0

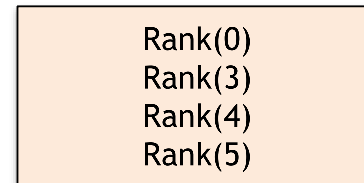


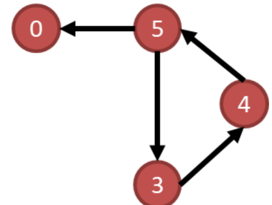




subgraph 1

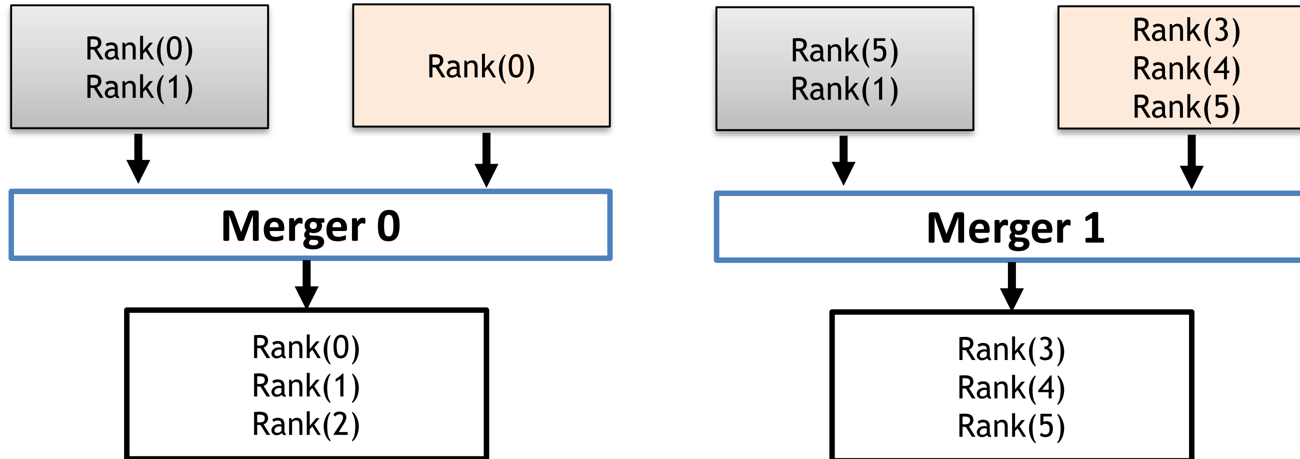






Step3: Cache-aware Merge

- Merge the intermediate results generated by each subgraph processing step



- Each merge only works on a set of vertex IDs that fit in L1 cache
- Multiple mergers work in parallel for different set of vertex IDs

Step3: Cache-aware Merge

- Merge the intermediate results generated by each subgraph processing step

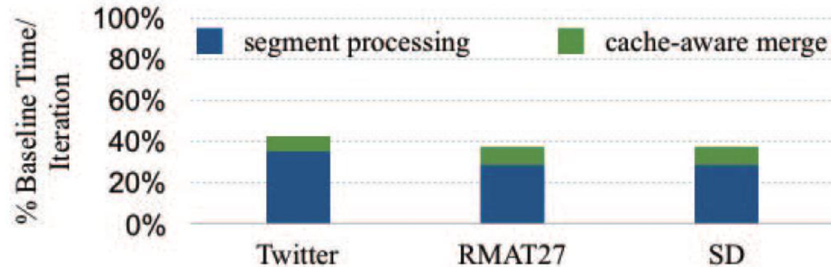


Fig. 5: Comparison of segment computation vs merge costs. Runtime% normalized to an optimized PageRank baseline without segmenting on 24 cores.

Rank(2)

Rank(5)

merging only incur a small overhead comparing to segmenting

- Each merge only works on a set of vertex IDs that fit in L1 cache
- Multiple mergers work in parallel for different set of vertex IDs

Programming Abstraction

- <u>Cagra</u>: extends on `EdgeMap` and `VetexMap` API from Ligra

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

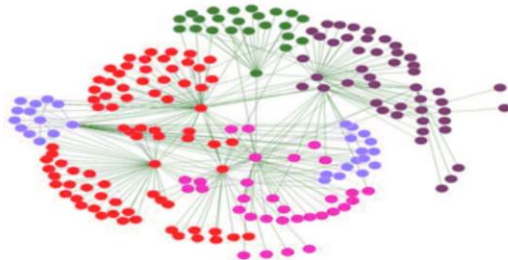
User-defined merge function that allows subgraphs to merge correctly in the framework

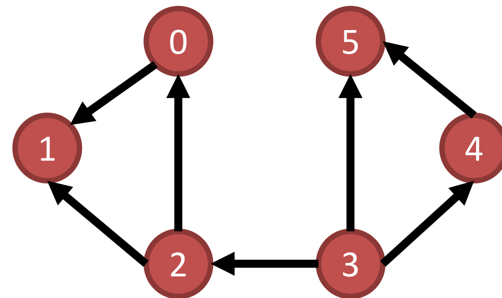
Optimization: Frequency Based Clustering

Observations

8byte useful data

64byte cache line





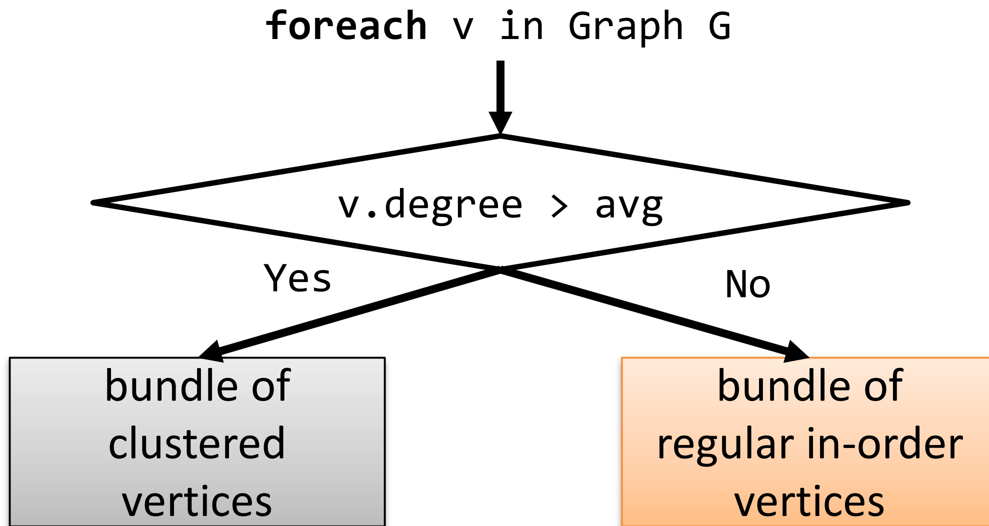
① random reads usually only utilize a small portion of the fetched cache line -> low locality

② High-degree vertices are more likely to be accessed than others

③ Natural ordering of the graph have indications of the relationships between the vertices

Optimization: Frequency Based Clustering

Group together the vertices that are frequently referenced while preserving the natural order as much as possible



Evaluation Setup

- Machine: Intel Xeon CPUs: 24 cores, 48 hyper threads
- Data Sets: social network data sets (power-law degree distribution)

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

- Applications: example applications from machine learning, graph traversals and graph analytics
  - PageRank, Label Propagation, Collaborative Filtering, Betweeness Centrality

Overall Runtime Compared to Existing Frameworks

PageRank Performance

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

Label Propagation Performance

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

Live journal dataset is small enough to fit in LLC
(Cagra becomes slower than due to extra preprocessing overhead)

Preprocessing Cost

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.

Pro:

- Small overheads introduced compared to overall runtime improvements

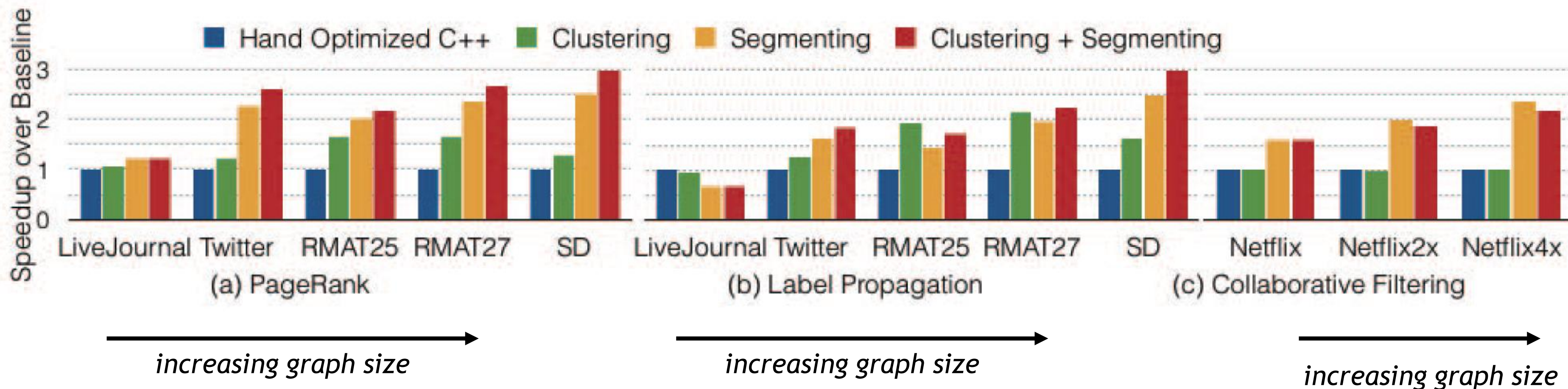
Con:

- Other framework's overhead not fully analyzed
    - GridGraph has more significant preprocessing overhead
        - 130ns for Twitter
- CSR segmenting's overhead does increase significantly when graph becomes larger

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

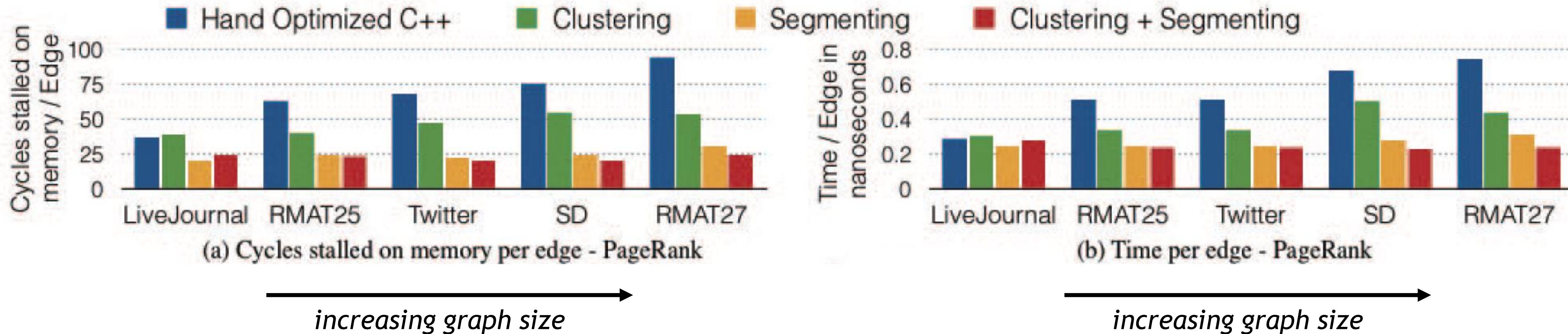
1.5x increase in number of edges
3.7x increase in preprocessing time

Contributions of Different Optimizations



CSR Segmenting alone allow speedup of more than 2x on all 3 applications

Contributions of Different Optimizations



- By constraining each subgraph inside LLC, CSR segmenting helps to keep the memory access relatively constant even if dataset size increases
- Clustering optimization is orthogonal to segmenting optimization

Summary

- Strength
  - Clear presentation of methodology
  - Evaluations show contributions of each optimization on various applications and datasets
- Weakness
  - More detailed implementation description would be helpful
  - Preprocessing cost not studied extensively