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 existing 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

Implicit dest id

offset array

edges array

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

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
Step 1: Preprocessing (graphical)

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

```
0 1 2
segment 0

3 4 5
segment 1
```

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

Original graph

Subgraph 0

Subgraph 1

Included because it is a necessary destination vertex
Step 1: Preprocessing (CSR specific)

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

Original CSR Representation

<table>
<thead>
<tr>
<th>global dest id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset array</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>edges array</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Subgraph 0 CSR

Subgraph 1 CSR
Step 1: Preprocessing (CSR specific)

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

Original CSR Representation

\[ \text{global dest id: } 0, 1, 2, 3, 4, 5 \]
\[ \text{offset array: } 0, 2, 4, 4, 5, 6 \]
\[ \text{edges array: } 2, 5, 0, 2, 5, 3, 1, 2, 4 \]

Subgroup id: 0, 1, 0, 0, 1, 1, 0, 0, 1

\[ N = 3 \text{ (number of vertex in each segment)} \]
Source id: 2, 5, ...
Subgroup id: \[ \lfloor 2/3 \rfloor = 0, \lfloor 5/3 \rfloor = 1, \ldots \]
Step 2: Processing Subgraphs

Subgraph 0

Subgraph 1

LLC cache

save intermediate rank values of each destination in the subgraph
Step2: Processing Subgraphs

LLC cache
working set arrays for vertices 0, 1, 2

- 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
Step 2: Processing Subgraphs

Subgraph 0

thread 0
- 0 (E: 2->0)

thread 1
- 1 (E: 0->1, E: 2->1)

thread 2
- 5 (E: 1->5, E: 2->5)

LLC cache
- subgraph 0 interm. results

save intermediate rank values of each destination in the subgraph

time
Step 2: Processing Subgraphs

**LLC cache**
working set arrays for vertices 3, 4, 5

- **Thread 0**
  - Time 0: E: 2→0
  - Time 1: E: 0→1

- **Thread 1**
  - Time 1: E: 0→1
  - Time 2: E: 2→1

- **Thread 2**
  - Time 2: E: 1→5
  - Time 3: E: 2→5

- **Thread 3**
  - Time 3: E: 5→0
  - Time 4: E: 5→3
  - Time 5: E: 3→4
  - Time 6: E: 4→5

---

Subgraph 0

Subgraph 1
Step 2: Processing Subgraphs

Subgraph 0

- 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 1

- thread 0
  - 0
  - E: 5->0
- thread 1
  - 3
  - E: 5->3
- thread 2
  - 4
  - E: 3->4
- thread 3
  - 5
  - E: 4->5

LLC cache

- subgraph 1
- interm. results

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.
**Step 3: Cache-aware Merge**

- Merge the intermediate results generated by each subgraph processing step

![Diagram showing the process of cache-aware merge]

- 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
Step 3: 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

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

- merging only incur a small overhead comparing to segmenting
Programming Abstraction

• **Cagra**: extends on *EdgeMap* and *VertexMap* API from Ligra

```
procedure PAGE-RANK(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
```

User-defined merge function that allows subgraphs to merge correctly in the framework
Optimization: Frequency Based Clustering

Observations

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

8-byte useful data

64-byte cache line
Optimization: Frequency Based Clustering

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

\[
\text{foreach } v \text{ in Graph } G
\]

\[
v.\text{degree} > \text{avg}
\]

Yes \rightarrow bundle of clustered vertices

No \rightarrow bundle of regular in-order vertices
Evaluation Setup

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of Vertices</th>
<th>Number of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal [10]</td>
<td>5 M</td>
<td>69 M</td>
</tr>
<tr>
<td>Twitter [8]</td>
<td>41 M</td>
<td>1469 M</td>
</tr>
<tr>
<td>RMAT 25 [12]</td>
<td>34 M</td>
<td>671 M</td>
</tr>
<tr>
<td>RMAT 27 [12]</td>
<td>134 M</td>
<td>2147 M</td>
</tr>
<tr>
<td>Netflix [13]</td>
<td>0.5 M</td>
<td>198 M</td>
</tr>
<tr>
<td>Netflix2x [14]</td>
<td>1 M</td>
<td>792 M</td>
</tr>
<tr>
<td>Netflix4x [14]</td>
<td>2 M</td>
<td>1585 M</td>
</tr>
</tbody>
</table>

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, Betweenness Centrality
Overall Runtime Compared to Existing Frameworks

PageRank Performance

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Cagra</th>
<th>HandOpt C++</th>
<th>GraphMat</th>
<th>Ligra</th>
<th>GridGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live Journal</td>
<td>0.017s</td>
<td>0.031s</td>
<td>0.028s</td>
<td>0.076s</td>
<td>0.195</td>
</tr>
<tr>
<td>(1.00×)</td>
<td>(1.79×)</td>
<td>(1.66×)</td>
<td>(4.45×)</td>
<td>(11.5×)</td>
<td></td>
</tr>
<tr>
<td>Twitter</td>
<td>0.29s</td>
<td>0.79s</td>
<td>1.20s</td>
<td>2.57s</td>
<td>2.58</td>
</tr>
<tr>
<td>(1.00×)</td>
<td>(2.72×)</td>
<td>(4.13×)</td>
<td>(8.86×)</td>
<td>(8.90×)</td>
<td></td>
</tr>
<tr>
<td>RMAT 25</td>
<td>0.15s</td>
<td>0.33s</td>
<td>0.5s</td>
<td>1.28s</td>
<td>1.65</td>
</tr>
<tr>
<td>(1.00×)</td>
<td>(2.20×)</td>
<td>(3.33×)</td>
<td>(8.53×)</td>
<td>(11.0×)</td>
<td></td>
</tr>
<tr>
<td>RMAT 27</td>
<td>0.58s</td>
<td>1.63s</td>
<td>2.50s</td>
<td>4.96s</td>
<td>6.5</td>
</tr>
<tr>
<td>(1.00×)</td>
<td>(2.80×)</td>
<td>(4.30×)</td>
<td>(8.53×)</td>
<td>(11.20×)</td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>0.43s</td>
<td>1.33s</td>
<td>2.23</td>
<td>3.48</td>
<td>3.9</td>
</tr>
<tr>
<td>(1.00×)</td>
<td>(2.62×)</td>
<td>(5.18×)</td>
<td>(8.10×)</td>
<td>(9.07×)</td>
<td></td>
</tr>
</tbody>
</table>

Label Propagation Performance

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Cagra</th>
<th>HandOpt C++</th>
<th>Ligra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live Journal</td>
<td>0.02s (1×)</td>
<td>0.01s (0.68×)</td>
<td>0.03s (1.51×)</td>
</tr>
<tr>
<td>Twitter</td>
<td>0.27s (1×)</td>
<td>0.51s (1.73×)</td>
<td>1.16s (3.57×)</td>
</tr>
<tr>
<td>RMAT 25</td>
<td>0.14s (1×)</td>
<td>0.33s (2.20×)</td>
<td>0.5s (3.33×)</td>
</tr>
<tr>
<td>RMAT 27</td>
<td>0.52s (1×)</td>
<td>1.17s (2.25×)</td>
<td>2.90s (5.58×)</td>
</tr>
<tr>
<td>SD</td>
<td>0.34s (1×)</td>
<td>1.05s (3.09×)</td>
<td>2.28s (6.71×)</td>
</tr>
</tbody>
</table>

TABLE II: PageRank runtime per iteration comparisons with other frameworks and slowdonw relative to Cagra

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

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Clustering</th>
<th>Segmenting</th>
<th>Build CSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal</td>
<td>0.1 s</td>
<td>0.2 s</td>
<td>0.48 s</td>
</tr>
<tr>
<td>Twitter</td>
<td>0.5 s</td>
<td>3.8 s</td>
<td>12.7 s</td>
</tr>
<tr>
<td>RMAT 27</td>
<td>1.4 s</td>
<td>6.3 s</td>
<td>39.3 s</td>
</tr>
</tbody>
</table>

TABLE VI: Preprocessing Runtime in Seconds.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of Vertices</th>
<th>Number of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal [10]</td>
<td>5 M</td>
<td>69 M</td>
</tr>
<tr>
<td>Twitter [8]</td>
<td>41 M</td>
<td>1469 M</td>
</tr>
<tr>
<td>RMAT 25 [12]</td>
<td>34 M</td>
<td>671 M</td>
</tr>
<tr>
<td>RMAT 27 [12]</td>
<td>134 M</td>
<td>2147 M</td>
</tr>
</tbody>
</table>

1.5x increase in number of edges
3.7x increase in preprocessing time
Contributions of Different Optimizations

Runtime Speedups of Optimizations on Page Rank, Label Propagation, and Collaborative Filtering

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

Memory Access Time Related Results

- 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