Exploiting Locality in Graph Applications

Presenter: Yunming Zhang
(joint work with many others)
Outline

• Overview

• Milk (PACT16, Vladimir Kiriansky et all)

• Cagra (BigData17, Yunming Zhang et all)

• GraphIt (OOPSLA18, Yunming Zhang et all)

• Conclusion
Outline

• Overview
• Milk
• Cagra
• GraphIt
• Conclusion
Outline

• Overview
• Milk
• Cagra
• GraphIt
• Conclusion
Sparse Data Processing

- Sparse Data (Graphs, Sparse Matrices, Tensors) are Everywhere

- Difficult to Write High-Performance Implementations
  - Hard to Parallelize (Load Balance, Synchronizations …)
  - Hard to Exploit Locality (Blocking, CacheLine utilization …)
## Locality by the Numbers

<table>
<thead>
<tr>
<th>Type</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local L1 Cache hit</td>
<td>~4 cycles (2.1 - 1.2 ns)</td>
</tr>
<tr>
<td>Local L2 Cache hit</td>
<td>~10 cycles (5.3 - 3.0 ns)</td>
</tr>
<tr>
<td>Local L3 Cache hit, line unshared</td>
<td>~40 cycles (21.4 - 12.0 ns)</td>
</tr>
<tr>
<td>Local L3 Cache hit, shared line in another core</td>
<td>~65 cycles (34.8 - 19.5 ns)</td>
</tr>
<tr>
<td>Local L3 Cache hit, modified in another core</td>
<td>~75 cycles (40.2 - 22.5 ns)</td>
</tr>
<tr>
<td>Remote L3 Cache (Ref: Fig.1 [Pg. 5])</td>
<td>~100-300 cycles (160.7 - 30.0 ns)</td>
</tr>
<tr>
<td>Local DRAM</td>
<td>~60 ns</td>
</tr>
<tr>
<td>Remote DRAM</td>
<td>~100 ns</td>
</tr>
</tbody>
</table>
Locality by the Numbers

<table>
<thead>
<tr>
<th>Type</th>
<th>Average Cycles</th>
<th>Delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local L1 CACHE hit</td>
<td>~4 cycles</td>
<td>2.1 - 1.2 ns</td>
</tr>
<tr>
<td>Local L2 CACHE hit</td>
<td>~10 cycles</td>
<td>5.3 - 3.0 ns</td>
</tr>
<tr>
<td>Local L3 CACHE hit, line unshared</td>
<td>~40 cycles</td>
<td>21.4 - 12.0 ns</td>
</tr>
<tr>
<td>Local L3 CACHE hit, shared line in another core</td>
<td>~65 cycles</td>
<td>34.8 - 19.5 ns</td>
</tr>
<tr>
<td>Local L3 CACHE hit, modified in another core</td>
<td>~75 cycles</td>
<td>40.2 - 22.5 ns</td>
</tr>
<tr>
<td>Remote L3 CACHE (Ref: Fig.1 [Pg. 5])</td>
<td>~100-300 cycles</td>
<td>160.7 - 30.0 ns</td>
</tr>
<tr>
<td>Local DRAM</td>
<td>~60 ns</td>
<td></td>
</tr>
<tr>
<td>Remote DRAM</td>
<td>~100 ns</td>
<td></td>
</tr>
</tbody>
</table>

Going to DRAM is usually 2.5-4x slower than L3 Cache
Today’s Talks

- Milk (Exploit Locality through Runtime Memory Accesses Reordering)
- Cagra (Exploit Locality through Preprocessing)
- GraphIt (Explore the Tradeoff Space between Locality, Parallelism, and Work-Efficiency)
Outline

• Overview
• Milk
• Cagra
• GraphIt
• Conclusion
Optimizing Indirect Memory References with milk

Vladimir Kiriansky, Yunming Zhang, Saman Amarasinghe

MIT

PACT ’16

September 13, 2016, Haifa, Israel
Indirect Accesses

```c
for(int i=0; i<N; i++)
    count[d[i]]++;
```
Indirect Accesses with OpenMP

01  #pragma omp parallel for
02  for(int i=0; i<N; i++)
03       #pragma omp atomic
04      count[d[i]]++;
Indirect Accesses with OpenMP

```c
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
count[d[i]]++;
```

uniform [0..100M)
8 threads, 8MB L3
Indirect Accesses with milk

01  #pragma omp parallel for milk
02  for(int i=0; i<N; i++)
03      #pragma omp atomic if(!milk)
04        count[d[i]]++;

uniform [0..100M]
8 threads, 8MB L3
No Locality?

- Cache miss
- TLB miss
- DRAM row miss
- No prefetching
No Locality?
No Locality?
No Locality?
Milk Clustering
Milk Clustering

- Cache hit
- TLB hit
- DRAM row hit
- Effective prefetching
Milk Clustering

- Cache hit
- TLB hit
- DRAM row hit
- Effective prefetching
- No need for atomics!
Outline

• Milk programming model

• milk syntax

• MILK compiler and runtime
Foundations

• Milk programming model — extending BSP

• milk syntax — OpenMP for C/C++

• MILK compiler and runtime — LLVM/Clang
Big (sparse) Data
Big (sparse) Data

• Terabyte Working Sets
  - AWS 2TB VM

• In-memory Databases, Key-value stores

• Machine Learning

• Graph Analytics
Milk — BSP extension

- Bulk-synchronous parallel (BSP) superstep - updates visible after a barrier

- Virtual processors can access only
  - One random cache line from DRAM
  - Sequential streams
  - Cache-resident data
Infinite Cache Locality in Graph Applications

- Ideal Cache Hit %
  - Temporal Locality
  - Spatial Locality

Graph Applications:
- Betweenness Centrality (BC)
- Breadth-First Search (BFS)
- Connected Components (CC)
- PageRank (PR)
- Single-Source Shortest Paths (SSSP)

Data Sets:
- Road (d=2.4)
- Twitter (d=24)
- Web (d=39)

[GAPBS]
Milk Execution Model

- Collection
- Distribution
- Delivery
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
    count[d[i]]+ += f(i);
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
count[d[i]] += f(i);

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>14</td>
<td>5</td>
<td>18</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

d

<table>
<thead>
<tr>
<th>7</th>
<th>f(0)</th>
<th>0</th>
<th>f(1)</th>
<th>14</th>
<th>f(2)</th>
<th>5</th>
<th>f(3)</th>
<th>18</th>
<th>f(4)</th>
<th>7</th>
<th>f(5)</th>
<th>0</th>
<th>f(6)</th>
<th>7</th>
<th>f(7)</th>
</tr>
</thead>
</table>

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11| 12| 13| 14| 15| 16| 17| 18| 19|
| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| red| read
```c
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
    count[d[i]] += f(i);
```

```
d | 7 0 14 5 18 7 0 7
```

```
f(0) | 7
f(1) | 0
f(2) | 14
f(3) | 5
f(4) | 18
f(5) | 7
f(6) | 0
f(7) | 7
```

```
count | 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```
```c
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
    count[d[i]] += f(i);
```

```
<table>
<thead>
<tr>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
</tbody>
</table>

```

distribution
```c
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
    count[d[i]] += f(i);
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>14</td>
<td>5</td>
<td>18</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>
```

```
0  f(1)  0  f(6)  5  f(3)  7  f(0)  7  f(5)  7  f(7)  14 f(2)  18 f(4)
```

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11| 12| 13| 14| 15| 16| 17| 18| 19|
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
    count[d[i]] += f(i);
milk syntax

- milk clause in parallel loop
- milk directive per indirect access
  - tag \( (i) \) — address to group by
  - pack \( (v) \) — additional state
pack Combiners

pack (v[:all])
pack (v:+|*|\text{min}|\text{max}|\text{any})
PageRank

vector<float> contrib, new_rank;

void PageRank_Push() {
    for (Node u=0; u < g.num_nodes(); u++) {
        float contribU = contrib[u];
        for (Node v : g.out_neigh(u))
            new_rank[v] += contribU;
    }
}
vector<float> contrib, new_rank;

void PageRank_Push() {
    #pragma omp parallel for
    for (Node u=0; u < g.num_nodes(); u++) {
        float contribU = contrib[u];
        for (Node v : g.out_neigh(u))
            #pragma omp atomic
            new_rank[v] += contribU;
    }
}
vector<float> contrib, new_rank;

void PageRank_Push() {
    #pragma omp parallel for milk
    for (Node u=0; u < g.num_nodes(); u++) {
        float contribU = contrib[u];
        for (Node v : g.out_neigh(u))

            #pragma omp atomic if(!milk)
            new_rank[v] += contribU;
    }
}

PageRank with milk

```c
vector<float> contrib, new_rank;

void PageRank_Push() {
    #pragma omp parallel for milk
    for (Node u=0; u < g.num_nodes(); u++) {
        float contribU = contrib[u];
        for (Node v : g.out_neigh(u))
            #pragma milk pack(contribU : +) tag(v)
            #pragma omp atomic if(!milk)
            new_rank[v] += contribU;
    }
}
```
MILK compiler and runtime

- Collection — loop transformation
- Distribution — runtime library
- Delivery — continuation
PageRank with milk

```cpp
vector<float> contrib, new_rank;

void PageRank_Push() {
    #pragma omp parallel for milk
    for (Node u=0; u < g.num_nodes(); u++) {
        float contribU = contrib[u];
        for (Node v : g.out_neigh(u))
            #pragma milk pack(contribU : +) tag(v)
            #pragma omp atomic if(!milk)
            new_rank[v] += contribU;
    }
}
```
PageRank: Collection

```cpp
vector<float> contrib, new_rank;

void PageRank_Push() {
    #pragma omp parallel for milk
    for (Node u=0; u < g.num_nodes(); u++) {
        float contribU = contrib[u];
        for (Node v : g.out_neigh(u))
            #pragma milk pack(contribU : +) tag(v)
    }
}
```
Tag Distribution

9-bit radix partition

L2

pails
Tag Distribution

0.5

7

p=7

17 0.1

L2

pails

...
Tag Distribution

L2

pails

\[ p=7 \]

\[
\begin{array}{c}
0.5 \\
7 \\
p=7 \\
17 \quad 0.1 \quad 7 \quad 0.5 \\
\end{array}
\]
Distribution: Pail Overflow

p=7

L2
pails

DRAM
tubs

0.2
17
0.1
7
0.5
17
0.2
Milk Delivery

DRAM

tubs

L2

17 0.2 27 0.1 7 0.1

17 0.1 7 0.5 17 0.2
Milk Delivery

```c
#pragma milk pack(contriBU : +) tag(v)
#pragma omp atomic if(!milk)
    new_rank[v] += contribU;
```
Related Work

• Database JOIN optimizations
  • [Shatdal94] cache partitioning
  • [Manegold02, Kim09, Albutiu12, Balkesen15] TLB, SIMD, NUMA, non-temporal writes, software write buffers
Overall Speedup with \textit{milk}

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Betweenness Centrality</td>
<td>2x</td>
</tr>
<tr>
<td>BFS</td>
<td>2x</td>
</tr>
<tr>
<td>CC</td>
<td>1.5x</td>
</tr>
<tr>
<td>PR  PageRank</td>
<td>3x</td>
</tr>
<tr>
<td>SSSP  Single-Source Shortest Paths</td>
<td>2.5x</td>
</tr>
</tbody>
</table>

V=32M  
8 MB L3
Overall Speedup with milk

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>2M</th>
<th>8M</th>
<th>32M</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>1.5x</td>
<td>2x</td>
<td>2.5x</td>
</tr>
<tr>
<td>BFSd</td>
<td>1.5x</td>
<td>2x</td>
<td>2.5x</td>
</tr>
<tr>
<td>BFSp</td>
<td>1.5x</td>
<td>2x</td>
<td>2.5x</td>
</tr>
<tr>
<td>CC</td>
<td>1x</td>
<td>1.5x</td>
<td>2x</td>
</tr>
<tr>
<td>PR</td>
<td>2x</td>
<td>2.5x</td>
<td>3x</td>
</tr>
<tr>
<td>SSSP</td>
<td>2x</td>
<td>2.5x</td>
<td>3x</td>
</tr>
</tbody>
</table>

8 MB L3
Stall Cycle Reduction

Baseline and milk comparison for L2 and L3 miss stalls.

L2 miss stalls:
- Baseline: 80%
- Milk: 20%

L3 miss stalls:
- Baseline: 80%
- Milk: 20%

PageRank,
V=32M, d=16 (uniform)
Indirect Access Cache Hit%

- BC
- BFS
- CC
- PR
- SSSP

Cache Hit %

- baseline
- milk

V=32M
8 MB L3
256KB L2
Higher Degree ➔ Higher Locality

- Speedup: 0x, 1x, 2x, 3x, 4x, 5x
- V=16M
- V=32M

Bar chart showing speedup for different average degrees and edge counts:
- 16M edges: 1, 2, 4, 8, 16, 32, 64
- Average Degree: 1x, 2x, 3x, 4x, 5x
- 2B edges

Notes:
- V=16M edges: 1, 2, 4, 8, 16, 32, 64
- V=32M edges: 1, 2, 4, 8, 16, 32, 64
Outline

• Overview
• Milk
• Cagra
• GraphIt
• Conclusion
Making Caches Work for Graph Analytics

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

MIT CSAIL and *Stanford InfoLab
Large Graphs

Social Graphs

Transaction Graph

Knowledge Graphs

Biological Graphs

Achieving good performance for graph applications is not easy.

- Graph applications have a lot of irregular memory accesses.
- PageRank incurs a last level cache (LLC) miss rate of 40% on Twitter graph.
- Other important applications, such as Dense Matrix Multiplication, have LLC miss rates are often be < 2%.
Outline

- Motivation
- Related Works
- Frequency based Vertex Reordering
- Cache-aware Segmenting
- Evaluation
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
        for node : graph.vertices
            newRanks[node] = baseScore + damping*newRanks[node];
        swap ranks and newRanks
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore +
                        damping*newRanks[node];
    swap ranks and newRanks
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
        for node : graph.vertices
            newRanks[node] = baseScore + damping*newRanks[node];
        swap ranks and newRanks
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
        for node : graph.vertices
            newRanks[node] = baseScore +
            damping*newRanks[node];
        swap ranks and newRanks
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore +
                        damping*newRanks[node];
    swap ranks and newRanks
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
PageRank

while ...
   for node : graph.vertices
      for ngh : graph.getInNeighbors(node)
         newRanks[node] += ranks[ngh]/outDegree[ngh];
   for node : graph.vertices
      newRanks[node] = baseScore + damping*newRanks[node];
   swap ranks and newRanks
while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore +
damping*newRanks[node];
  swap ranks and newRanks

Compressed Sparse Row (CSR)

Vertex Array  | 0 1 3 5 7
---|---
Edge Array   | 2 0 2 0 3 0 2
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore +
            damping*newRanks[node];
    swap ranks and newRanks

Compressed Sparse Row (CSR)

Vertex Array stores indices into the Edge Array. Edge Array stores neighbors’ ID in the CSR
PageRank

while ...

for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
        newRanks[node] += ranks[ngh]/outDegree[ngh];

for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];

swap ranks and newRanks

Sequential access on node when scanning through vertex array

Vertex Array

0 1 3 5 7

Edge Array

2 0 2 0 3 0 2
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += \( \frac{\text{ranks}[\text{ngh}]}{\text{outDegree}[\text{ngh}]} \);
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Irregular access on ngh’s rank and outDegree data when scanning through the edge array
PageRank

while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore +
    damping*newRanks[node];
  swap ranks and newRanks

Cache

#hits: 0
#misses: 0

0 1 2 3
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh] / outDegree[ngh];
        for node : graph.vertices
            newRanks[node] = baseScore +
                damping*newRanks[node];
        swap ranks and newRanks

Focus on the random memory accesses on ranks array

Cache

#hits: 0
#misses: 0
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Focus on the random memory accesses on ranks array

Cache

holds one cache line

#hits: 0
#misses: 0
while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += \( \text{ranks}[\text{ngh}] / \text{outDegree}[\text{ngh}] \);
  for node : graph.vertices
    newRanks[node] = baseScore + damping * newRanks[node];
  swap ranks and newRanks

Focus on the random memory accesses on ranks array

Cache
  holds one cache line

#hits: 0
#misses: 0

stored in two cache lines
while ...
   for node : graph.vertices
      for ngh : graph.getInNeighbors(node)
         newRanks[node] += ranks[ngh]/outDegree[ngh];
   for node : graph.vertices
      newRanks[node] = baseScore + damping*newRanks[node];
      swap ranks and newRanks

#misses: 0
#hits: 0
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
        for node : graph.vertices
            newRanks[node] = baseScore + damping*newRanks[node];
        swap ranks and newRanks
PageRank

while ...
for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
        newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
swap ranks and newRanks

Cache

#hits: 0
#misses: 1
while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore +
        damping*newRanks[node];
    swap ranks and newRanks

Cache

#hits: 0
#misses: 1
PageRank

while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Cache

#hits: 0
#misses: 2
while ...
   for node : graph.vertices
      for ngh : graph.getInNeighbors(node)
         newRanks[node] += ranks[ngh]/outDegree[ngh];
   for node : graph.vertices
      newRanks[node] = baseScore +
         damping*newRanks[node];
   swap ranks and newRanks

Cache

#hits: 0
#misses: 2

PageRank
while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Cache

| 0 | 1 |
#hits: 0
#misses: 2
PageRank

while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks

Cache

#hits: 0
#misses: 3

0 1 2 3
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Cache

#hits: 0
#misses: 3
PageRank

while ...
   for node : graph.vertices
      for ngh : graph.getInNeighbors(node)
         newRanks[node] += ranks[ngh]/outDegree[ngh];
      for node : graph.vertices
         newRanks[node] = baseScore + damping*newRanks[node];
      swap ranks and newRanks

Cache

#hits: 0
#misses: 4
PageRank

while ...
   for node : graph.vertices
      for ngh : graph.getInNeighbors(node)
         newRanks[node] += ranks[ngh]/outDegree[ngh];
   for node : graph.vertices
      newRanks[node] = baseScore + damping*newRanks[node];
   swap ranks and newRanks

Cache

#hits: 0
#misses: 4
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Cache

#hits: 0
#misses: 5
PageRank

while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks

Cache

#hits: 0
#misses: 5
while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
      newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Cache

#hits: 1
#misses: 5

PageRank
PageRank

while ...
   for node : graph.vertices
      for ngh : graph.getInNeighbors(node)
         newRanks[node] += ranks[ngh]/outDegree[ngh];
   for node : graph.vertices
      newRanks[node] = baseScore +
      damping*newRanks[node];
   swap ranks and newRanks

Cache

#hits: 1
#misses: 5
while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore +
        damping*newRanks[node];
    swap ranks and newRanks
PageRank

while ...
  for node : graph.vertices
    for ngh: graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore +
    damping*newRanks[node];
  swap ranks and newRanks

Cache

#hits: 1
#misses: 6

A very high miss rate
PageRank

while …

for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
        newRanks[node] += ranks[ngh]/outDegree[ngh];

for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];

swap ranks and newRanks
PageRank

while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore +
      damping*newRanks[node];
  swap ranks and newRanks

Often only use part of the cache line

Cache

#misses: 6
#hits: 1
Performance Bottleneck

• Working set much larger than cache size

• Access pattern is irregular

  • Often uses part of the cache line

  • Hard to benefit from hardware prefetching

  • TLB miss, DRAM row miss (hundreds of cycles)
Performance Bottleneck

- Working set much larger than cache size
- Access pattern is irregular
  - Often uses part of the cache line
  - Hard to benefit from hardware prefetching
  - TLB miss, DRAM row miss (hundreds of cycles)

Real-world graphs often have working set 10-200x larger than cache size
Performance Bottleneck

- Working set much larger than cache size
- Access pattern is irregular
  - Often uses part of the cache line
  - Hard to benefit from hardware prefetching
- TLB miss, DRAM row miss (hundreds of cycles)
Performance Bottleneck

- Working set much larger than cache size
- Access pattern is irregular
  - Often uses part of the cache line
- Hard to benefit from hardware prefetching
- TLB miss, DRAM row miss (hundreds of cycles)
PageRank

while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore +
    damping*newRanks[node];
  swap ranks and newRanks

on RMAT27 graph
while ...

for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
        newRanks[node] += ranks[ngh]/outDegree[ngh];

for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];

swap ranks and newRanks

Up to 80% of the cycles are spent on the collection phase due to slow irregular memory accesses on RMAT27 graph
while ... 
   for node : graph.vertices 
      for ngh : graph.getInNeighbors(node) 
         newRanks[node] += ranks[0]/outDegree[0];
   for node : graph.vertices 
      newRanks[node] = baseScore + damping*newRanks[node]; 
   swap ranks and newRanks

PageRank on RMAT27 graph
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[0]/outDegree[0];
    for node : graph.vertices
        newRanks[node] = baseScore +
                        damping*newRanks[node];
    swap ranks and newRanks

2.8x speedup if we can eliminate random memory accesses

on RMAT27 graph
while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[0]/outDegree[0];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Within 2x of no random accesses

on RMAT27 graph
Outline

• Motivation

• Related Works

• Frequency based Vertex Reordering

• Cache-aware Segmenting

• Evaluation
Related Work

• Distributed Graph Systems
  • Shared memory efficiency is a key component of distributed
graph processing systems (PowerGraph, GraphLab, Pregel...) 

• Shared-memory Graph Systems
  • Frameworks (Ligra, Galois, GraphMat ..) did not focus on
cache optimizations
  • Milk [PACT16], Propagation Blocking[IPDPS17]
  • Out-of-core Systems (GraphChi, FlashGraph, BigSparse …)
Outline

• Motivation
• Related Works
• Frequency based Vertex Reordering
• Cache-aware Segmenting
• Evaluation
Frequency based Vertex Reordering

- Key Observations
  - Cache lines are underutilized
  - Certain vertices are much more likely to be accessed than other vertices
Frequency based Vertex Reordering

• Key Observations
  • Cache lines are underutilized
  • Certain vertices are much more likely to be accessed than other vertices

• Design
  • Group together the frequently accessed nodes
  • Keep the ordering of average degree nodes
Frequency based Vertex Reordering
Frequency based Vertex Reordering

0 1 2 3

0: outdegree: 3
1: outdegree: 0
2: outdegree: 3
3: outdegree: 1
Frequency based Vertex Reordering

Group together high outdegree nodes
Frequency based Vertex Reordering

Group together high outdegree nodes
Frequency based Vertex Reordering

Group together high outdegree nodes

Reorder nodes 1 and 2
Frequency based Vertex Reordering

Group together high outdegree nodes

Reorder nodes 1 and 2
Frequency based Vertex Reordering

- Group together high outdegree nodes
- Build a new CSR, relabel corresponding edges
Frequency based Vertex Reordering

Reorganize nodes’ data
Frequency based Vertex Reordering

Reorganize nodes’ data
Frequency based Vertex Reordering

- Groups together the data of frequently accessed nodes in one cache line
- Reorganize nodes’ data
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
        for node : graph.vertices
            newRanks[node] = baseScore +
                damping*newRanks[node];
        swap ranks and newRanks

Cache

#hits: 0
#misses: 0
0 1 2 3
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Focus on the random memory accesses on ranks array
PageRank

while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks
PageRank

while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Cache

#hits: 0
#misses: 1
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
while ...

for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
        newRanks[node] += ranks[ngh]/outDegree[ngh];

for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];

swap ranks and newRanks
while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
      newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks

#misses: 2
PageRank

while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks

Cache

#hits: 1
#misses: 2
PageRank

while ... 
for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
        newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Cache

#hits: 1
#misses: 3
PageRank

while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks

Cache

|   0   |   1   |

#hits: 2
#misses: 3
PageRank

while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks
while ...
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh] / outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping * newRanks[node];
  swap ranks and newRanks

Cache

#hits: 4
#misses: 3
PageRank

while ...
   for node : graph.vertices
      for ngh : graph.getInNeighbors(node)
         newRanks[node] += ranks[ngh]/outDegree[ngh];
   for node : graph.vertices
      newRanks[node] = baseScore + damping*newRanks[node];
   swap ranks and newRanks

Cache

#hits: 4  #misses: 3
0 1 2 3

Much better than

#hits: 1  #misses: 6
0 1 2 3
PageRank

while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks

Better cache line utilization

Cache

3
0
1
2
0
1
2
3

#hits: 4
#misses: 3

#hits: 1
#misses: 6
Outline

- Motivation
- Related Works
- Frequency based Vertex Reordering
- Cache-aware Segmenting
- Evaluation
Cache-aware Segmenting

• Design
  • Partition the graph into subgraphs where the random access are limited to LLC
  • Process each partition sequentially and accumulate rank contributions for each partition
  • Merge the rank contributions from all subgraphs
Graph Partitioning
Graph Partitioning

Partitions the original graph into subgraphs that only access a subset of nodes’ data
Graph Partitioning

Partitions the original graph into subgraphs that only access a subset of nodes’ data.
Graph Partitioning

Partitions the original graph into subgraphs that only access a subset of nodes’ data.
Graph Partitioning

Partitions the original graph into subgraphs that only access a subset of nodes’ data
Graph Partitioning

Partitions the original graph into subgraphs that only access a subset of nodes’ data.
Graph Partitioning

Partitions the original graph into subgraphs that only access a subset of nodes’ data.
Graph Processing

Cache

#hits: 0
#misses: 0

0
1
2
3

0
1
0
1

2
3

2
3
Graph Processing

Cache

#hits: 0
#misses: 0

---

#misses: 0

---

#hits: 0
Graph Processing

Cache

#hits: 0
#misses: 1
Graph Processing

Cache

#hits: 0
#misses: 1

0

1

3

0

1

3

2

0

1

3

2

1

2

3
Graph Processing

Cache

#hits: 1
#misses: 1
Graph Processing

Cache

#hits: 2
#misses: 1
Graph Processing

Cache

#hits: 2
#misses: 1

0 1

0 1

0 1

0 1

2 3
Graph Processing

#misses: 1

#hits: 2

Cache

0 1
Graph Processing

#hits: 2
#misses: 1

Cache

0
1

3
0 1

0
1

3
2

0
1

2
3

134
Graph Processing

#misses: 1

#hits: 2

#misses: 1
Graph Processing

Cache

#hits: 2
#misses: 2
Graph Processing

#hits: 3
#misses: 2
Graph Processing

Cache

#hits: 4
#misses: 2
Graph Processing

#hits: 5
#misses: 2
Graph Processing

Only have 2 misses

#hits: 5
#misses: 2
Graph Processing

Better than Frequency based Reordering

#hits: 4
#misses: 3

#hits: 5
#misses: 2
Cache-aware Merge

Diagram showing a process or algorithm with interconnected nodes labeled 0, 1, 2, and 3.
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge

Diagram: Two sets of nodes and arrows connecting them, indicating a merge process. The nodes are labeled with numbers, and arrows point from one node to another, showing the flow or connection between them. The diagram is split into two sections, each depicting a different stage of the merge process.
Cache-aware Merge

0 → 1
3 ← 2

0 → 1
3 ← 2

1 2 3
0 1 2 3
Cache-aware Merge
Cache-aware Merge

Diagram showing the process of cache-aware merge with nodes and edges representing data flow.
Cache-aware Merge

0 1 2 3
Cache-aware Merge

The naive approach incurs random DRAM accesses
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge

Break down into chunks that fit in cache
Cache-aware Merge

Sum up the intermediate updates from the two subgraphs
Cache-aware Merge

Sum up the intermediate updates from the two subgraphs
Cache-aware Merge

Sum up the intermediate updates from the two subgraphs
Cache-aware Merge

Sum up the intermediate updates from the two subgraphs
Cache-aware Merge

Sum up the intermediate updates from the two subgraphs.
Cache-aware Merge

Sum up the intermediate updates from the two subgraphs
Cache-aware Merge

Sum up the intermediate updates from the two subgraphs
PageRank

while ...

    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];

    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];

    swap ranks and newRanks

on RMAT27 graph
while ...

for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
        newRanks[node] += ranks[ngh]/outDegree[ngh];

for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];

swap ranks and newRanks

35% cycle reduction
PageRank

while ... 
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks

50% cycle reduction on RMAT27 graph
PageRank

```python
while ...
    for node in graph.vertices:
        for ngh in graph.getInNeighbors(node):
            newRanks[node] += ranks[ngh] / outDegree[ngh];
    for node in graph.vertices:
        newRanks[node] = baseScore + damping * newRanks[node];
        swap ranks and newRanks
```

60% cycle reduction on RMAT27 graph
Outline

• Motivation
• Related Works
• Frequency based Vertex Reordering
• Cache-aware Segmenting
• Evaluation
## Evaluation

<table>
<thead>
<tr>
<th></th>
<th>PageRank (20 iter)</th>
<th>Label Propagation (per iter)</th>
<th>Betweenness Centrality (per start node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>5.8s</td>
<td>0.27s</td>
<td>1.21s</td>
</tr>
<tr>
<td>RMAT27</td>
<td>11.6s</td>
<td>0.52s</td>
<td>1.825s</td>
</tr>
<tr>
<td>Web Graph</td>
<td>8.6s</td>
<td>0.34s</td>
<td>0.0875s</td>
</tr>
</tbody>
</table>

Absolute Running Times on 24 core Intel Xeon E5 servers
**Evaluation**

In a single machine, we can complete 20 iterations of PageRank on 40 million nodes Twitter graph within 6s.

<table>
<thead>
<tr>
<th></th>
<th>PageRank (20 iter)</th>
<th>Label Propagation (per iter)</th>
<th>Betweenness Centrality (per start node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>5.8s</td>
<td>0.27s</td>
<td>1.21s</td>
</tr>
<tr>
<td>RMAT27</td>
<td>11.6s</td>
<td>0.52s</td>
<td>1.825s</td>
</tr>
<tr>
<td>Web Graph</td>
<td>8.6s</td>
<td>0.34s</td>
<td>0.0875s</td>
</tr>
</tbody>
</table>

Absolute Running Times on 24 core Intel Xeon E5 servers
In a single machine, we can complete 20 iterations of PageRank on 40 million nodes Twitter graph within 6s.

The best published results so far is 12.7s (Gemini OSDI 2017).

<table>
<thead>
<tr>
<th></th>
<th>PageRank (20 iter)</th>
<th>Label Propagation (per iter)</th>
<th>Betweenness Centrality (per start node)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Twitter</strong></td>
<td>5.8s</td>
<td>0.27s</td>
<td>1.21s</td>
</tr>
<tr>
<td><strong>RMAT27</strong></td>
<td>11.6s</td>
<td>0.52s</td>
<td>1.825s</td>
</tr>
<tr>
<td><strong>Web Graph</strong></td>
<td>8.6s</td>
<td>0.34s</td>
<td>0.0875s</td>
</tr>
</tbody>
</table>

**Absolute Running Times on 24 core Intel Xeon E5 servers**
## Evaluation

<table>
<thead>
<tr>
<th></th>
<th>PageRank (20 iter)</th>
<th>Label Propagation (per iter)</th>
<th>Betweenness Centrality (per start node)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Twitter</strong></td>
<td>5.8s</td>
<td>0.27s</td>
<td>1.21s</td>
</tr>
<tr>
<td><strong>RMAT27</strong></td>
<td>11.6s</td>
<td>0.52s</td>
<td>1.825s</td>
</tr>
<tr>
<td><strong>Web Graph</strong></td>
<td>8.6s</td>
<td>0.34s</td>
<td>0.0875s</td>
</tr>
</tbody>
</table>

Very fast execution on label propagation used in Connected Components and SSSP (Bellman-Ford)

Absolute Running Times on 24 core Intel Xeon E5 servers
PageRank

![Bar chart showing slowdown to Ours for different datasets and frameworks: Twitter, RMAT25, RMAT27, SD. The x-axis represents the datasets, and the y-axis represents the slowdown. The frameworks compared are Ours, HandOptC++, GraphMat, Ligra, and GridGraph.](image-url)
PageRank

Intel expert hand optimized version and state-of-the-art graph frameworks are 2.2-11x slower than our version.
Label Propagation

![Label Propagation Diagram]

- Twitter
- RMAT25
- RMAT27
- SD

- Ours
- HandOptC++
- Ligra

Slowdown to Ours

- Twitter: 1.75
- RMAT25: 3.5
- RMAT27: 5.25
- SD: 7

162
Label Propagation

Intel expert hand optimized version and state-of-the-art graph frameworks are 1.7-6.7x slower than our version
Evaluation

![Chart showing cycles stalled on memory/edge for different datasets: LiveJournal, RMAT25, Twitter, SD, and RMAT27. The chart compares 'Hand Optimized C++' and 'Ours'.]
Cycles stalled on memory per edge increases as the size of the graph increases
Evaluation

Cycles stalled on memory per edge stays constant as the size of the graph increases.
Summary

• Performance Bottleneck of Graph Applications
• Frequency based Vertex Reordering
• Cache-aware Segmenting
Outline

• Overview
• Milk
• Cagra
• GraphIt
• Conclusion
Outline

• Overview
• Milk
• Cagra
• GraphIt
• Conclusion
Conclusion

• Locality is very important

• Performance optimizations are all about tradeoffs (always engineer for your specific applications and data)

• Separation of algorithm from optimization (gets good programmability without sacrificing too much performance)
Ongoing Projects

• Optimizing Performance for Ordered Graph Algorithms
• Running Graph Algorithms on GPU
• Applying Graph Optimizations to Sparse Matrix Linear Algebra
• Subgraph Matching