Optimizing Cache Performance for Graph Analytics

Yunming Zhang 6.886 Presentation
Goals

• How to optimize in-memory graph applications

• How to go about performance engineering just about anything
In-memory Graph Processing

- Compare to Disk / DRAM boundary (GraphChi, BigSparse, LLAMA..), Cache / DRAM boundary has
  - Much smaller latency gap (L3 cache 10-30 ns, DRAM 80-100 ns, Flash 100,000 ns (100 ms))
  - Much larger memory bandwidth (DRAM >100GB/s, Flash 6GB/s)
  - Much smaller granularity (64 bytes cache lines vs 4k or 2 MB pages)
Outline

• Performance Analysis for Graph Applications
• Milk / Propagation Blocking
• Frequency based Clustering
• CSR Segmenting
• Summary
Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server

Scott Beamer, Krste Asanović, David Patterson

Mostly borrowed from the authors’ IISWC presentation
Motivation

• What is the performance bottleneck for graph applications running in memory?

• How much performance can we gain?

• How can we achieve the performance gains?
Graph Algorithms Are Random?
Graph Algorithms Are Random?

“Thus, the low speedup of OOO execution is due solely to a lack of memory bandwidth required to service the repeated last level cache misses caused by the random access memory pattern of the algorithm.”

PPoPP 2011
“Thus, the low speedup of OOO execution is due solely to a lack of memory bandwidth required to service the repeated last level cache misses caused by the random access memory pattern of the algorithm.”
Graph Algorithms Are Random?

“Thus, the low speedup of OOO execution is due solely to a lack of memory bandwidth required to service the repeated last level cache misses caused by the random access memory pattern of the algorithm.”

PPoPP 2011

“First, the memory bandwidth of the system seems to limit performance”

SPAA 2010
Graph Algorithms Are Random?

“Thus, the low speedup of OOO execution is due solely to a lack of memory bandwidth required to service the repeated last level cache misses caused by the random access memory pattern of the algorithm.”

PPoPP 2011

“First, the memory bandwidth of the system seems to limit performance”

SPAA 2010
Current Graph Architecture Wisdom?

Current Wisdom
Current Graph Architecture Wisdom?

Current Wisdom

- Random memory access pattern
Current Graph Architecture Wisdom?

Current Wisdom

- Random memory access pattern
- Limited by memory bandwidth
Current Graph Architecture Wisdom?

Current Wisdom

- Random memory access pattern
- Limited by memory bandwidth
- Will be plagued by low core utilization
Current Graph Architecture Wisdom?

Current Wisdom

- Random memory access pattern
- Limited by memory bandwidth
- Will be plagued by low core utilization

Cray XMT Design
Current Graph Architecture Wisdom?

Current Wisdom

- Random memory access pattern
- Limited by memory bandwidth
- Will be plagued by low core utilization

Cray XMT Design

- No caches
Current Graph Architecture Wisdom?

Current Wisdom
- Random memory access pattern
- Limited by memory bandwidth
- Will be plagued by low core utilization

Cray XMT Design
- No caches
- Heavy multithreading
Are graph applications really memory bandwidth bounded?

- Is cache really completely useless in graph computations?
Results from Characterization
Results from Characterization

- No single representative workload
Results from Characterization

- No single representative workload
  - need suite
Results from Characterization

- No single representative workload
  - need suite
- Out-of-order core not limited by memory bandwidth for most graph workloads
Results from Characterization

- No single representative workload
  - need suite
- Out-of-order core not limited by memory bandwidth for most graph workloads
  - can improve by changing only processor
Results from Characterization

- No single representative workload
  - need suite
- Out-of-order core not limited by memory bandwidth for most graph workloads
  - can improve by changing only processor
- Many graph workloads have good locality
Results from Characterization

- No single representative workload
  - need suite

- Out-of-order core not limited by memory bandwidth for most graph workloads
  - can improve by changing only processor

- Many graph workloads have good locality
  - caches help! try to avoid thrashing
Target Graph Algorithms

Most popular based on 45-paper literature survey:
- Breadth-First Search (BFS)
- Single-Source Shortest Paths (SSSP)
- PageRank (PR)
- Connected Components (CC)
- Betweenness Centrality (BC)
Target Graph Frameworks
Target Graph Frameworks

- **Galois** (custom parallel runtime) - UT Austin
  - specialized for irregular fine-grain tasks
Target Graph Frameworks

- **Galois** (custom parallel runtime) - UT Austin
  - specialized for irregular fine-grain tasks

- **Ligra** (Cilk) - CMU
  - applies algorithm in push or pull directions
Target Graph Frameworks

- **Galois** (custom parallel runtime) - UT Austin
  - specialized for irregular fine-grain tasks
- **Ligra** (Cilk) - CMU
  - applies algorithm in push or pull directions
- **GAP** Benchmark Suite (OpenMP) - UCB
  - written directly in most natural way for algorithm, not constrained by framework
<table>
<thead>
<tr>
<th>Graph</th>
<th># Vertices</th>
<th># Edges</th>
<th>Degree</th>
<th>Diameter</th>
<th>Degree Dist.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roads of USA</td>
<td>23.9M</td>
<td>58.3M</td>
<td>2.4</td>
<td>High</td>
<td>const</td>
</tr>
<tr>
<td>Twitter Follow Links</td>
<td>61.6M</td>
<td>1468.4M</td>
<td>23.8</td>
<td>Low</td>
<td>power</td>
</tr>
<tr>
<td>Web Crawl of .sk Domain</td>
<td>50.6M</td>
<td>1949.4M</td>
<td>38.5</td>
<td>Medium</td>
<td>power</td>
</tr>
<tr>
<td>Kronecker Synthetic Graph</td>
<td>128.0M</td>
<td>2048.0M</td>
<td>16.0</td>
<td>Low</td>
<td>power</td>
</tr>
<tr>
<td>Uniform Random Graph</td>
<td>128.0M</td>
<td>2048.0M</td>
<td>16.0</td>
<td>Low</td>
<td>normal</td>
</tr>
</tbody>
</table>

Graphs can have very different degree distributions, diameters and other structural characteristics.
How does the memory system work?
How does the memory system work?

- Executing a load that access DRAM requires:
How does the memory system work?

- Executing a load that access DRAM requires:
  1. Execution reaches load instruction (**fetch**)}
How does the memory system work?

- Executing a load that access DRAM requires:
  1. Execution reaches load instruction (fetch)
  2. Space in the instruction window
How does the memory system work?

- Executing a load that access DRAM requires:
  1. Execution reaches load instruction (**fetch**)
  2. Space in the instruction **window**
  3. Register operands are available (**dataflow**)
How does the memory system work?

- Executing a load that access DRAM requires:
  1. Execution reaches load instruction (fetch)
  2. Space in the instruction window
  3. Register operands are available (dataflow)
  4. Memory bandwidth is available
How does the memory system work?

- Executing a load that access DRAM requires:
  1. Execution reaches load instruction (**fetch**)
  2. Space in the instruction **window**
  3. Register operands are available (**dataflow**)
  4. Memory **bandwidth** is available

- Bandwidth ~ # outstanding requests
How does the memory system work?

- Executing a load that access DRAM requires:
  
  **1.** Execution reaches load instruction (fetch)
  
  **2.** Space in the instruction window
  
  **3.** Register operands are available (dataflow)
  
  **4.** Memory bandwidth is available

- Memory bandwidth (#4) matters only if (#1-3) satisfied

- Bandwidth \( \sim \) # outstanding requests
Little’s Law
Little’s Law

effective MLP

(MLP = memory level parallelism)
Little’s Law

\[ \text{effective MLP} = \frac{\text{average memory bandwidth}}{\text{average memory latency}} \]

(MLP = memory level parallelism)
Little’s Law

effective MLP = \frac{\text{average memory bandwidth}}{\text{average memory latency}} \leq \text{application MLP}

(MLP = \text{memory level parallelism})
Single-Core Memory Bandwidth

1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases
Single-Core Memory Bandwidth

1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases
Single-Core Memory Bandwidth

1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases
Single-Core Memory Bandwidth

Pointer Chasing Microbenchmark with varying number of parallel pointer chases

1 core
Single-Core Memory Bandwidth

1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases
Single-Core Memory Bandwidth

Pointer Chasing Microbenchmark with varying number of parallel pointer chases

1 core

1 thread

fetch

window
Single-Core Memory Bandwidth

Pointer Chasing Microbenchmark with varying number of parallel pointer chases
Single-Core Memory Bandwidth

1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases
Single-Core Memory Bandwidth

Pointer Chasing Microbenchmark with varying number of parallel pointer chases
Outline

- Methodology
- Platform Memory Bandwidth Availability
- **Single-core Results**
- Parallel Results
- GAP Benchmark Suite
- Conclusion
Importance of Window Size

Application MLP:
- 12
- 8
- 4
- 2
- 1

1 core w/ 1 thread

Instructions per Miss (IPM)
Importance of Window Size

1 core w/ 1 thread

Effective MLP vs. Instructions per Miss (IPM)

Application MLP:
- 12: Solid blue circles
- 8: Blue diamonds
- 4: Green squares
- 2: Orange triangles
- 1: Purple triangles

18
Importance of Window Size

1 core w/ 1 thread

Application MLP
- Blue circles: 12
- Blue diamonds: 8
- Green squares: 4
- Yellow triangles: 2
- Purple triangles: 1

Effective MLP vs. Instructions per Miss (IPM)

1 fetch
Importance of Window Size

1 core w/ 1 thread

1 fetch

Effective MLP vs. Instructions per Miss (IPM)

Application MLP:
- 12
- 8
- 4
- 2
- 1

1 core w/ 1 thread

Instructions per Miss (IPM)
Importance of Window Size

1 core w/ 1 thread

1 core w/ 1 thread

Instructions per Miss (IPM)

Effective MLP

Application MLP

12
8
4
2
1

1 fetch

2 window
Importance of Window Size

1 core w/ 1 thread

1 core w/ 1 thread

Instructions per Miss (IPM)

Effective MLP

Application MLP

- 12
- 8
- 4
- 2
- 1

1. fetch

2. window

3. dataflow
Importance of Window Size

1 core w/ 1 thread

1. fetch
2. window
3. dataflow
4. bandwidth

Application MLP
- 12
- 8
- 4
- 2
- 1

Effective MLP vs. Instructions per Miss (IPM)
Importance of Window Size

1 core w/ 1 thread

Instructions per Miss (IPM)

Effective MLP

Application MLP
- 12
- 8
- 4
- 2
- 1

Model
Importance of Window Size

![Diagram showing the relationship between Effective MLP, window size, IPM, and Instructions per Miss (IPM). The graph demonstrates how the Effective MLP changes with varying window sizes and IPM values. The legend indicates different symbols for 12, 8, 4, 2, and 1 applications, with a model line for comparison. The x-axis represents Instructions per Miss (IPM), and the y-axis shows Effective MLP. The text "1 core w/ 1 thread" is also mentioned.]
Importance of Window Size

Instruction window size limits memory bandwidth if misses rare
Biggest Influence on Single-Core?

1 core w/ 1 thread
Biggest Influence on Single-Core?

1 core w/ 1 thread
Biggest Influence on Single-Core?

IPC vs. Effective MLP for different codebases:
- Galois
- GAPBS
- Ligra

1 core w/ 1 thread
Biggest Influence on Single-Core?

1 core w/ 1 thread
Biggest Influence on Single-Core?

1 core w/ 1 thread

Need suite, no single representative workload
Biggest Influence on Single-Core?

Need suite, no single representative workload

Only few workloads near memory bandwidth limit

1 core w/ 1 thread
1 core w/ 1 thread
1 core w/ 1 thread
Instruction Window Limits BW

1 core w/ 1 thread
Instruction Window Limits BW

1 core w/ 1 thread

1 window
Instruction Window Limits BW

Instruction window limits memory bandwidth

1 core w/ 1 thread
Outline

- Methodology
- Platform Memory Bandwidth Availability
- Single-core Results
- Parallel Results
- GAP Benchmark Suite
- Conclusion
Memory Bandwidth ~ Performance

- 32 threads vs. 1 thread

[Graph showing runtime speedup (x) vs. memory bandwidth increase (x) for various benchmarks: kron, road, web, twitter, uniform.]
Increasing memory bandwidth utilization increases performance.

Memory Bandwidth ~ Performance

32 threads vs. 1 thread

Memory Bandwidth Increase (x)

Runtime Speedup (x)

kron
road
web
Parallel Utilization

16 cores w/ 32 threads
At system level, memory bandwidth under-utilized

16 cores w/ 32 threads
Multithreading Opportunity

1. fetch
2. window
3. dataflow
4. bandwidth
Multithreading Opportunity

1. fetch
2. dataflow
3. bandwidth

same hardware (shared)
Multithreading Opportunity

1. fetch

2. window

3. dataflow

4. bandwidth

same hardware (shared)
Multithreading Opportunity

1. fetch + fewer instructions in flight
2. window  same hardware (shared)
3. dataflow
4. bandwidth same hardware (shared)
Multithreading Opportunity

1. fetch + fewer instructions in flight
2. window same hardware (shared)
3. dataflow ++ more application MLP
4. bandwidth same hardware (shared)
Multithreading Increases Bandwidth

1 core w/ 1-2 threads

Runtime Speedup (x)

Memory Bandwidth Increase (x)
Multithreading Increases Bandwidth

Speedups imply workloads probably bottlenecked by dataflow
Conclusions
Conclusions

- Most graph workloads do not utilize a large fraction of memory bandwidth
Conclusions

- Most graph workloads do not utilize a large fraction of memory bandwidth
  - Many graph workloads have decent locality
Conclusions

- Most graph workloads do not utilize a large fraction of memory bandwidth
  - Many graph workloads have decent locality
  - Cache misses too infrequent to fit in window
Conclusions

- Most graph workloads do not utilize a large fraction of memory bandwidth
  - Many graph workloads have decent locality
  - Cache misses too infrequent to fit in window
  - Changing processor alone could help
Conclusions

- Most graph workloads do not utilize a large fraction of memory bandwidth
  - Many graph workloads have decent locality
  - Cache misses too infrequent to fit in window
  - Changing processor alone could help

- Sub-linear parallel speedups cast doubt on gains from multithreading on OoO core
Outline

• Performance Analysis for Graph Applications

• Milk / Propagation Blocking

• Frequency based Clustering

• CSR Segmenting

• Summary
Milk / Propagation Blocking

• Is changing the architecture really the only way to improve the performance of graph applications running in memory?
  • Boosting MLP

• Other approaches
  • Reducing the amount of communication
Optimizing Indirect Memory References with milk

Vladimir Kiriansky, Yunming Zhang, Saman Amarasinghe

MIT

PACT ’16
September 13, 2016, Haifa, Israel
Indirect Accesses

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

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

![Speedup Chart]

uniform [0..100M]
8 threads, 8MB L3
Indirect Accesses with \textit{milk}

```
01  #pragma omp parallel for \textcolor{magenta}{milk}
02  for\textcolor{green}{(int \ i=0; \ i<N; \ i++)}
03     #pragma omp atomic \textcolor{red}{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

- Terabyte Working Sets
  - AWS 2TB VM
- In-memory Databases, Key-value stores
- Machine Learning
- Graph Analytics
Infinite Cache Locality in Graph Applications

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

Road (d=2.4)
Twitter (d=24)
Web (d=39)

[GAPBS]
Milk Execution Model

- Collection
- Distribution
- Delivery

Propagation Blocking:
  Binning
  (Collection + Distribution),
  Deliver
  (Accumulation)
```c
#pragma omp parallel for
for(int i=0; i<N; i++)
   #pragma omp atomic
   count[d[i]] += f(i);
```

```
```

```
```

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

![Image](image_url)
```c
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
    count[d[i]] += f(i);
```
```c
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
    count[d[i]] += f(i);
```
```c
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
    count[d[i]] += f(i);
```
```c
#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
  - \texttt{tag}(i) — address to group by
  - \texttt{pack}(v) — additional state
pack Combiners

pack (v[:all])

pack (v:+|*|min|max|any)
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;
        }
    }
}
PageRank with \textit{milk}

\begin{verbatim}
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;
    }
}
\end{verbatim}
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;
    }
}
```
MILK compiler and runtime

- Collection — loop transformation
- Distribution — runtime library
- Delivery — continuation
PageRank with \textit{milk}

\begin{verbatim}
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;
    }
}
\end{verbatim}
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

L2 pails

17 0.1
Tag Distribution

L2

pails

17 0.1 7 0.5
Distribution: Pail Overflow

L2
pails

...  

DRAM

tubs

17 0.1 7 0.5 17 0.2
Milk Delivery
Milk Delivery

```
#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 **milk**

![Graph showing speedup for different algorithms]

- **BC**: Betweenness Centrality
- **BFS**: Breadth-First Search
- **CC**: Connected Components
- **PR**: PageRank
- **SSSP**: Single-Source Shortest Paths

V=32M
8 MB L3
Overall Speedup with **milk**

<table>
<thead>
<tr>
<th>Speedup</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>1x</td>
<td>1.5x</td>
<td>2x</td>
</tr>
<tr>
<td>BFSp</td>
<td>0.5x</td>
<td>1x</td>
<td>1.5x</td>
</tr>
<tr>
<td>CC</td>
<td>0.5x</td>
<td>1x</td>
<td>1.5x</td>
</tr>
<tr>
<td>PR</td>
<td>2x</td>
<td>2.5x</td>
<td>3x</td>
</tr>
<tr>
<td>SSSP</td>
<td>1x</td>
<td>2x</td>
<td>2.5x</td>
</tr>
</tbody>
</table>

8 MB L3
Stall Cycle Reduction

- L2 miss stalls: 256 KB L2
- L3 miss stalls: 8 MB L3

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

V=32M
8 MB L3
256KB L2
Higher Degree $\rightarrow$ Higher Locality

![Bar chart showing speedup with varying degrees and average degrees.](image-url)
Related Works

• How is Milk different from BigSparse and Propagation Blocking?
Related Work

• Big Sparse
  • Big Sparse can work on both graphs with good and bad locality (Milk and Propagation Blocking both work on low locality graphs)
  • Can afford to do global sort instead of bucketing

• Propagation Blocking
  • Milk doesn’t have two separate phases for Binning and Accumulate (Collection, Distribution, Delivery are all fused together using coroutines)
  • PB reuses the tags to save memory bandwidth assuming the application is iterative
  • Milk has a more general programming model for various applications
Outline

- Performance Analysis for Graph Applications
- Milk / Propagation Blocking
- Frequency based Clustering
- CSR Segmenting
- Summary
Making Caches Work for Graph Analytics

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

MIT CSAIL and *Stanford InfoLab
Outline

• PageRank

• 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
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

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

Cache

#misses: 0
#hits: 0

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

Focus on the random memory accesses on ranks array

Cache
holds one cache line

#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

stored in two cache lines
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: 0
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

#misses: 1
#hits: 0

0 1 2 3
PageRank

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

Cache

\begin{array}{ccc}
  2 & 3 \\
\end{array}

#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

| 2 | 3 |

#hits: 0
#misses: 3
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

Cache

#hits: 0
#misses: 4

0 1

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

#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

0 1 2 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: 1
#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: 1
#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: 1
#misses: 6
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 ...

```java
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

Working set larger than cache
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

#hits: 1
#misses: 6

0 1 2 3

0 1 2 3
Performance Bottleneck

- Working set much larger than cache size
- Access pattern is random
  - Often uses part of the cache line
  - Can not benefit from hardware prefetching
- TLB miss, DRAM row miss (hundreds of cycles)
Performance Bottleneck

- Working set much larger than cache size
- Access pattern is random
  - Often uses part of the cache line
  - Can not benefit from hardware prefetching
  - TLB miss, DRAM row miss (hundreds of cycles)
Performance Bottleneck

• Working set much larger than cache size

• Access pattern is random

  • Often uses part of the cache line
  
  • Can not benefit from hardware prefetching

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

• Working set much larger than cache size
• Access pattern is random
  • Often uses part of the cache line
  • Can not benefit from hardware prefetching
• TLB miss, DRAM row miss (hundreds of cycles)
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 ...

```java
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 slow random memory accesses

on RMAT27 graph
while ...

for node : graph\texttt{.vertices}
  for ngh : graph\texttt{.getInNeighbors}(node)
    newRanks[node] += \texttt{ranks[0]/outDegree[0];}

for node : graph\texttt{.vertices}
  newRanks[node] = baseScore + damping*newRanks[node];

\texttt{swap ranks and newRanks}

---

**PageRank**

<table>
<thead>
<tr>
<th>Normalized Total Cycles</th>
<th>BaseLine</th>
<th>No Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Removing Random Accesses (Incorrect)

2.8x speedup if we can eliminate random memory accesses

on RMAT27 graph
while ...

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

**PageRank**

---

**Normalized Total Cycles**

- **BaseLine**: Within 2x of no random accesses
- **Cache Optimized**
- **No Random**

**on RMAT27 graph**
Outline

• PageRank

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

Group together high outdegree nodes
Frequency based Vertex Reordering

0  1  2  3

Group together high outdegree nodes

outdegree: 3

outdegree: 0

outdegree: 1

outdegree: 3
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

Relabel corresponding edges
Frequency based Vertex Reordering

Reorganize nodes’ data
Frequency based Vertex Reordering

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

Reorganize nodes’ data

Frequency based Vertex Reordering
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
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

0 1 2 3
PageRank

while ...

```python
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
```

Cache

#hits: 0
#misses: 0

0 1 2 3
PageRank

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

Cache

#hits: 0
#misses: 0

0 1 2 3

0 1 2 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: 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

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

Cache

#hits: 1
#misses: 1
PageRank

while ...

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

for node : graph.\texttt{vertices}
  newRanks[node] = baseScore + damping*newRanks[node];

swap \texttt{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: 2
#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
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: 3
#misses: 1

0 1 2 3

124
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: 3
#misses: 1

0 1 2 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: 3
#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

| 2 | 3 |
---|---|

#hits: 3
#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: 3
#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: 3
#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
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
Much better than
#hits: 1  #misses: 6
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

#misses: 3
#hits: 4
#misses: 6
#hits: 1
Outline

• PageRank

• 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

#hits: 0
#misses: 0
Graph Processing

Cache

#hits: 0
#misses: 0

![Graph Diagram]
Graph Processing

Cache

#hits: 0
#misses: 1
Graph Processing

Cache

#hits: 0
#misses: 1

#misses: 1
Graph Processing

#misses: 1
#hits: 1

Cache

0 1

0 1 2 3

0 1 2 3

0 1

2 3
Graph Processing

#hits: 2
#misses: 1
Graph Processing

Cache

#hits: 2
#misses: 1

0 1

3 0 1

0 1

2 3

0 1

1 2

2 3

0 1

1 2

Graph Processing

Cache

#hits: 2
#misses: 1

0 1

3 0 1

0 1

2 3

0 1

1 2

2 3

142
Graph Processing

Cache

#hits: 2
#misses: 1
Graph Processing

Cache

#hits: 2
#misses: 1
Graph Processing

#misses: 1
#hits: 2
Graph Processing

Cache

#hits: 2
#misses: 2
Graph Processing

#hit: 3
#misses: 2
Graph Processing

#misses: 2

#hits: 4

Cache

0 1
Graph Processing

Cache

#hits: 5
#misses: 2
Graph Processing

Only have 2 misses

Cache

#hits: 5
#misses: 2
Graph Processing

Better than Frequency based Reordering

#hits: 4
#misses: 3

Cache

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

Diagram showing the process of cache-aware merge with nodes and connections.
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge

![Graph showing the process of cache-aware merge]
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge
Cache-aware Merge

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

Diagram of data nodes and connections.
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

\[
\begin{align*}
0 + 1 &= 1 \\
0 + 1 &= 1 \\
2 + 3 &= 5
\end{align*}
\]
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

0 + 1

0 1

2 3

2 3
Sum up the intermediate updates from the two subgraphs.
while ...

```python
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
```
while ...

```python
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
```

35% cycle reduction

on RMAT27 graph
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

while ...

```python
    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
```

---

**Normalized Total Cycles on RMAT27 graph:**

- **BaseLine:** 100%
- **Reordering:** 75%
- **Segmenting:** 50%
- **Reordering + Segmenting:** 25%

*60% cycle reduction*
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, XStream)
Outline

• Motivation

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

<table>
<thead>
<tr>
<th>Dataset</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

<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>

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)

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>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>

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

Slowdown to Ours

- Ours
- HandOptC++
- GraphMat
- Ligra
- GridGraph
PageRank

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

The diagram shows the slowdown of different methods compared to our approach. The methods include:
- Ours
- HandOptC++
- Ligra

The x-axis represents different datasets: Twitter, RMAT25, RMAT27, and SD. The y-axis represents the slowdown compared to our approach. The results indicate that our method significantly outperforms the others in terms of speed.
Label Propagation

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

![Bar chart showing cycles stalled on memory/edge for different platforms. The chart compares Hand Optimized C++ and Ours for LiveJournal, RMAT25, Twitter, SD, and RMAT27.]
Evaluation

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

• Performance Analysis for Graph Applications
• Milk / Propagation Blocking
• Frequency based Clustering
• CSR Segmenting
• Summary
Improving Cache Performance for Graph Computations

- Reordering the Graph
- Partitioning the Graph for Locality
- Runtime Reordering the Memory Accesses
Improving Cache Performance for Graph Computations

- Reordering the Graph
- Partitioning the Graph for Locality
- Runtime Reordering the Memory Accesses

What are the tradeoffs?
Improving Cache Performance for Graph Computations

• Reordering the Graph
  • Small preprocessing cost, modest performance improvement, dependent on graph structure

• Partitioning the Graph for Locality
  • Bigger preprocessing cost, small runtime overhead, bigger performance gains, suitable for applications with lots of random accesses.

• Runtime Reordering the Memory Accesses
  • No preprocessing cost, bigger runtime overhead
Outside of Graph Computing?

- Sparse Linear Algebra
  - Matrix Reordering, Preconditioning (Graph Reordering)
  - Cache Blocking (CSR segmenting)
  - Inspector-Executor (Runtime Access Reordering)

- These are Fundamental Communication Reductions Techniques, used in many other domains (sparse linear algebra, join optimization in databases)
Performance Engineering

• Understand your applications’ performance characteristics
  
  • Many papers worked on different ways to abandon cache and improve MLP with a large number of threads
  
• Understand the tradeoff space of the optimizations

• Pick the technique that best suit your hardware, application and data