When Is Graph Reordering An Optimization?

A Cross Application and Input Graph Study on the Effectiveness of Lightweight Graph Reordering

Vignesh Balaji  Brandon Lucia
Graph Processing Has Many Applications

- Path Planning
- Social network analysis
- Recommender systems
Graph Processing Has Many Applications

- Path Planning
- Social network analysis
- Recommender systems
Graph Applications Are Memory Bound

Cycles stalled on DRAM / Total Cycles

LLC Miss Rate

Figure from “Optimizing Cache Performance for Graph Analytics” ArXiv v1;
Graph Applications Are Memory Bound

Problem: Poor LLC locality ⇒ Many long-latency DRAM accesses
Reason For Poor Locality - Irregular Memory Accesses
Reason For Poor Locality - Irregular Memory Accesses

```python
for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)
```

Typical graph processing kernel
Reason For Poor Locality - Irregular Memory Accesses

for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)

Typical graph processing kernel

Input Graph
Reason For Poor Locality - Irregular Memory Accesses

for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)

Typical graph processing kernel

Input Graph

Compressed Sparse Row (CSR) Representation
Reason For Poor Locality - Irregular Memory Accesses

for v in G:
    for u in neigh(v):
        process(..., vtxData[u], ...)

Irregular accesses to vtxData array

Typical graph processing kernel

Input Graph

Compressed Sparse Row (CSR) Representation

Offsets

Coordinates
Irregular Accesses Have Poor Temporal And Spatial Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)

\[
\begin{array}{ccccccccc}
0 & 1 & 4 & 8 & 8 & 11 & \ldots \\
3 & 0 & 7 & 3 & 1 & 3 & 7 & 9 & 0 & 3 & 5 & \ldots
\end{array}
\]

\[
2 \text{ lines}
\]

\[
2 \text{ words/line}
\]

LLC

Time
Irregular Accesses Have Poor Temporal And Spatial Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u], ...)

vtxData[3]
for v in G:
    for u in neigh(v):
        process(..., vtxData[u], ...)

vtxData[3]

Line granular transfers. 2 words / line

Miss

Offsets
0 1 4 8 8 11 ...

Coordinates
3 0 7 3 1 3 7 9 0 3 5 ...

Time

LLC
Irregular Accesses Have Poor Temporal And Spatial Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)

vtxData[3]  vtxData[0]

Miss  Miss

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LLC</td>
<td>LLC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offsets: 0 1 4 8 8 11 ...
Coordinates: 3 0 7 3 1 3 7 9 0 3 5 ...

Time
Irregular Accesses Have Poor Temporal And Spatial Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u], ...)

![Diagram showing temporal and spatial locality](image)

- **Omits**
  - **VtxData[3]**
  - **VtxData[0]**
  - **VtxData[7]**

**Eviction due to capacity miss**
Irregular Accesses Have Poor Temporal And Spatial Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)


Miss  Miss  Miss  Miss

0 1 4 8 8 11 ...
3 0 7 3 1 3 7 9 0 3 5 ...

Time
Irregular Accesses Have Poor Temporal And Spatial Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u], ...)


Miss Miss Miss Miss

2 3 2 3 6 7 6 7
0 1 0 1 2 3

LLC LLC LLC LLC

Working set size >> LLC capacity

↓ Poor Temporal Locality
Irregular Accesses Have Poor Temporal And Spatial Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)


Miss        Miss        Miss        Miss

2 3          2 3          6 7          6 7
0 1          0 1          2 3

Time

Working set size >> LLC capacity

Poor Temporal Locality

Line Size > Access granularity

Poor Spatial Locality
Outline

❖ Poor Locality of Graph Processing Applications ✔
❖ Improving locality through Graph Reordering
❖ Graph Reordering Challenge - *Application and Input-dependent Speedups*
❖ When is Graph Reordering an Optimization?
❖ Selective Graph Reordering
Outline

❖ Poor Locality of Graph Processing Applications  ✔
❖ **Improving locality through Graph Reordering**
❖ Graph Reordering Challenge - *Application and Input-dependent Speedups*
❖ When is Graph Reordering an Optimization?
❖ Selective Graph Reordering
Real-world Graphs Offer Opportunities To Improve Locality

Air Traffic Network

Power-law Degree Distribution
Real-world Graphs Offer Opportunities To Improve Locality

Power-law Degree Distribution

Air Traffic Network

Hubs
Real-world Graphs Offer Opportunities To Improve Locality

Power-law Degree Distribution

Community Structure

Hubs

Air Traffic Network

Facebook friend Graph

Right figure from “Rabbit Order: Just-in-time Parallel Reordering for Fast Graph Analysis” IPDPS 2016
Real-world Graphs Offer Opportunities To Improve Locality

Power-law Degree Distribution

Air Traffic Network

Hubs

Community Structure

Communities

Facebook friend Graph
Real-world Graphs Offer Opportunities To Improve Locality

Observation: Subset of vertices are accessed together
Reordering To Improve Locality of Graph Applications

**Key Insight:** Store commonly accessed vertices contiguously in memory
Reordering To Improve Locality of Graph Applications

**Key Insight:** Store commonly accessed vertices contiguously in memory

*Power-law graph*

```
0 1 2 3 4 5 6 7 8 9
```

*Offsets*

```
0 1 4 8 8 11 ...
```

*Coordinates*

```
3 0 7 3 1 3 7 9 0 3 5 ...
```
Reordering To Improve Locality of Graph Applications

Key Insight: Store commonly accessed vertices contiguously in memory

Power-law graph

Offsets
0 1 4 8 8 11 ...

Coordinates
3 0 7 3 1 3 7 9 0 3 5 ...

Reorder

Offsets
0 0 2 3 4 7 ...

Coordinates
0 2 1 0 0 3 5 0 0 1 0 ...

Carnegie Mellon
Reordering Improves Spatial & Temporal Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)

Reordered CSR

Time

2 lines
2 words/line

LLC
for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)

vtxData[0]

Miss

LLC

Offsets

0 0 2 3 4 7 ...

Coordinates

0 2 1 0 0 3 5 0 0 1 0 ...

Time
Reordering Improves Spatial & Temporal Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)

vtxData[0]  vtxData[2]

Miss  Miss

0 1
2 3

0 1
2 3

LLC  LLC

Offsets
0 0 2 3 4 7 ...

Coordinates
0 2 1 0 0 3 5 0 0 1 0 ...

Time
for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)

Reordering Improves Spatial & Temporal Locality

Offsets
0 0 2 3 4 7 ...

Coordinates
0 2 1 0 0 3 5 0 0 1 0 ...

vtxData[0] vtxData[2] vtxData[1]

Miss Miss Hit

0 1 0 1 0 1

2 3 2 3 2 3

LLC LLC LLC
Reordering Improves Spatial & Temporal Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)

vtxData[0]  vtxData[2]  vtxData[1]  vtxData[0]

Miss  Miss  Hit  Hit

0 1  0 1  0 1  0 1

0 1  2 3  2 3  2 3

LLC  LLC  LLC  LLC

Offsets
0 0 2 3 4 7 ...

Coordinates
0 2 1 0 3 5 0 0 1 0 ...

Time
Reordering Improves Spatial & Temporal Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)

<table>
<thead>
<tr>
<th>Time</th>
<th>vtxData[0]</th>
<th>vtxData[2]</th>
<th>vtxData[1]</th>
<th>vtxData[0]</th>
<th>vtxData[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss</td>
<td>Miss</td>
<td>Hit</td>
<td>Hit</td>
<td>Hit</td>
<td>Hit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LLC</th>
<th>LLC</th>
<th>LLC</th>
<th>LLC</th>
<th>LLC</th>
<th>LLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>2 3</td>
<td>2 3</td>
<td>2 3</td>
<td>2 3</td>
<td>2 3</td>
<td>2 3</td>
</tr>
</tbody>
</table>

Offsets: 0 0 2 3 4 7 ...
Coordinates: 0 2 1 0 0 3 5 0 0 1 0 ...

Reordering Improves Spatial & Temporal Locality
Reordering Improves Spatial & Temporal Locality

for v in G:
    for u in neigh(v):
        process(..., vtxData[u],...)

Graph Reordering improved **Spatial** and **Temporal** locality of vtxData accesses
Outline

❖ Poor Locality of Graph Processing Applications ✔
❖ Improving locality through Graph Reordering ✔
❖ Graph Reordering Challenge - Application and Input-dependent Speedups
❖ When is Graph Reordering an Optimization?
❖ Selective Graph Reordering
Outline

- Poor Locality of Graph Processing Applications ✔
- Improving locality through Graph Reordering ✔
- **Graph Reordering Challenge - Application and Input-dependent Speedups**
- When is Graph Reordering an Optimization?
- Selective Graph Reordering
Graph Reordering Is Not A Panacea

\[
\text{Speedup} = \frac{T_{\text{Original}}}{T_{\text{Reordered}} + \text{Reordering Time}}
\]
Graph Reordering Is Not A Panacea

![Bar Chart]

Net Speedup

\[ \text{Speedup} = \frac{T_{\text{Original}}}{T_{\text{Reordered}} + \text{Reordering Time}} \]
Graph Reordering Is Not A Panacea

\[ \text{Speedup} = \frac{T_{\text{Original}}}{T_{\text{Reordered}} + \text{Reordering Time}} \]
Net speedup from Reordering depends on the **Application** and **Input Graph**
Question: What are the properties of Applications and Input Graphs that benefit from Reordering?
Outline

❖ Poor Locality of Graph Processing Applications ✔
❖ Improving locality through Graph Reordering ✔
❖ Graph Reordering Challenge - Application and Input-dependent Speedups ✔
❖ When is Graph Reordering an Optimization?
❖ Selective Graph Reordering
Outline

❖ Poor Locality of Graph Processing Applications ✔
❖ Improving locality through Graph Reordering ✔
❖ Graph Reordering Challenge - Application and Input-dependent Speedups ✔
❖ When is Graph Reordering an Optimization?
   ➢ Characterization Space
   ➢ Which Applications benefit from Reordering?
   ➢ Which Input Graphs benefit from Reordering?
❖ Selective Graph Reordering
Outline

❖ Poor Locality of Graph Processing Applications ✔
❖ Improving locality through Graph Reordering ✔
❖ Graph Reordering Challenge - Application and Input-dependent Speedups ✔
❖ When is Graph Reordering an Optimization?
  ➢ Characterization Space
  ➢ Which Applications benefit from Reordering?
  ➢ Which Input Graphs benefit from Reordering?
❖ Selective Graph Reordering
Characterization Space

3 Graph Reordering Techniques

15 Applications (Ligra, GAP)

8 Input Graphs (M vertices, B edges)

Server-class Processor
(dual-socket, 28 cores, 35MB LLC, 64GB DRAM)
Lightweight Reordering (LWR) Techniques

Selection Criteria: Low reordering overhead
(Require very few runs/iterations to amortize overheads)

➢ Rabbit Ordering [Arai et. al., IPDPS 2016]
➢ Frequency-based Clustering (or “Hub-Sorting”) [Zhang et. al., Big Data 2017]
➢ Hub-Clustering (Our Variation of Hub Sorting)
LWR 1 - Rabbit Ordering

(a) Randomly ordered graph

(c) Adjacency matrix of graph (a)

Figure from “Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis” IPDPS 2016
LWR 1 - Rabbit Ordering

(a) Randomly ordered graph
(b) Reordered graph
(c) Adjacency matrix of graph (a)
(d) Adjacency matrix of graph (b)

Figure from “Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis” IPDPS 2016
Fast community detection using incremental aggregation

Complexity - $O(|E| - ck|V|)$

where $c = \text{clustering coeff.}$

$k = \text{avg. degree}$
LWR 2 & 3 - HubSorting & HubClustering

Vertex Degrees (Original)

<table>
<thead>
<tr>
<th>v0</th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
<th>v5</th>
<th>v6</th>
<th>v7</th>
<th>v8</th>
<th>v9</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
<td>4</td>
<td>21</td>
<td>25</td>
<td>99</td>
<td>6</td>
<td>49</td>
<td>64</td>
<td>4</td>
</tr>
</tbody>
</table>

degrees

Vtx IDs
### Vertex Degrees (Original)

<table>
<thead>
<tr>
<th>Vtx IDs</th>
<th>4</th>
<th>20</th>
<th>4</th>
<th>21</th>
<th>25</th>
<th>99</th>
<th>6</th>
<th>49</th>
<th>64</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
<td>v8</td>
<td>v9</td>
<td></td>
</tr>
</tbody>
</table>

### Vertex Degrees (Hub Sorted)

<table>
<thead>
<tr>
<th>Vtx IDs</th>
<th>99</th>
<th>64</th>
<th>49</th>
<th>21</th>
<th>25</th>
<th>4</th>
<th>6</th>
<th>4</th>
<th>20</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
<td>v8</td>
<td>v9</td>
<td></td>
</tr>
</tbody>
</table>

**HubSorting**

**Vtx IDs**

**Degrees**
LWR 2 & 3 - HubSorting & HubClustering

Vertex Degrees (Original)

<table>
<thead>
<tr>
<th>vertex</th>
<th>degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>4</td>
</tr>
<tr>
<td>v1</td>
<td>20</td>
</tr>
<tr>
<td>v2</td>
<td>4</td>
</tr>
<tr>
<td>v3</td>
<td>21</td>
</tr>
<tr>
<td>v4</td>
<td>25</td>
</tr>
<tr>
<td>v5</td>
<td>99</td>
</tr>
<tr>
<td>v6</td>
<td>6</td>
</tr>
<tr>
<td>v7</td>
<td>49</td>
</tr>
<tr>
<td>v8</td>
<td>64</td>
</tr>
<tr>
<td>v9</td>
<td>4</td>
</tr>
</tbody>
</table>

Vertex Degrees (Hub Sorted)

<table>
<thead>
<tr>
<th>vertex</th>
<th>degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>99</td>
</tr>
<tr>
<td>v1</td>
<td>64</td>
</tr>
<tr>
<td>v2</td>
<td>49</td>
</tr>
<tr>
<td>v3</td>
<td>21</td>
</tr>
<tr>
<td>v4</td>
<td>25</td>
</tr>
<tr>
<td>v5</td>
<td>4</td>
</tr>
<tr>
<td>v6</td>
<td>6</td>
</tr>
<tr>
<td>v7</td>
<td>20</td>
</tr>
<tr>
<td>v8</td>
<td>4</td>
</tr>
<tr>
<td>v9</td>
<td>4</td>
</tr>
</tbody>
</table>

Vertex Degrees (Hub Clustered)

<table>
<thead>
<tr>
<th>vertex</th>
<th>degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>99</td>
</tr>
<tr>
<td>v1</td>
<td>49</td>
</tr>
<tr>
<td>v2</td>
<td>64</td>
</tr>
<tr>
<td>v3</td>
<td>21</td>
</tr>
<tr>
<td>v4</td>
<td>25</td>
</tr>
<tr>
<td>v5</td>
<td>4</td>
</tr>
<tr>
<td>v6</td>
<td>6</td>
</tr>
<tr>
<td>v7</td>
<td>20</td>
</tr>
<tr>
<td>v8</td>
<td>4</td>
</tr>
<tr>
<td>v9</td>
<td>4</td>
</tr>
</tbody>
</table>
LWR 2 & 3 - HubSorting & HubClustering

### Vertex Degrees (Original)

<table>
<thead>
<tr>
<th>Vtx IDs</th>
<th>4</th>
<th>20</th>
<th>4</th>
<th>21</th>
<th>25</th>
<th>99</th>
<th>6</th>
<th>49</th>
<th>64</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
<td>v8</td>
<td>v9</td>
<td></td>
</tr>
</tbody>
</table>

#### HubSorting

- **Locality Benefits**: Temporal AND Spatial ↑↑
- **Complexity**: $O(|V|.\log V)$ ↓↓

#### HubClustering

- **Locality Benefits**: Temporal ↑
- **Complexity**: $O(|V|)$ ↓
Outline

❖ Poor Locality of Graph Processing Applications ✔

❖ Improving locality through Graph Reordering ✔

❖ Graph Reordering Challenge - *Application and Input-dependent Speedups* ✔

❖ **When is Graph Reordering an Optimization?**
  ➢ Characterization Space ✔
  ➢ Which Applications benefit from Reordering?
  ➢ Which Input Graphs benefit from Reordering?

❖ Selective Graph Reordering
Outline

❖ Poor Locality of Graph Processing Applications ✔

❖ Improving locality through Graph Reordering ✔

❖ Graph Reordering Challenge - Application and Input-dependent Speedups ✔

❖ When is Graph Reordering an Optimization?
  ➢ Characterization Space ✔
  ➢ *Which Applications benefit from Reordering?*
  ➢ Which Input Graphs benefit from Reordering?

❖ Selective Graph Reordering
Legend for Results

- **Application**
- **Reordering Technique**
- **Input Graph**

Speedup with respect to original ordering of graph

*Graph showing speedup with respect to original ordering for different applications and reordering techniques.*

Legend:
- **Rabbit**
- **HubSort**
- **HubCluster**

*Bar chart with applications DBP, GPL, PLD, KRON, TWIT, MPI, WEB, SD1 showing speedup values.*
Legend for Results

Speedup excluding the overhead of reordering

(Max Speedup)
Legend for Results

Speedup including the overhead of reordering

(Net Speedup)
Legend for Results

Reordering overhead

Speedup including the overhead of reordering

(Net Speedup)
15 Applications → 5 Categories

• **Category 1:** Applications processing Large Frontiers are *good candidates*

• **Category 2:** Symmetric bipartite graphs require *bi-partiteness aware reordering*

• **Category 3:** Applications processing small frontiers *offer limited opportunity*

• **Category 4:** Reordering for Push-style applications introduces *false-sharing*

• **Category 5:** Reordering *affects convergence* for applications with ID-dependent computations
15 Applications $\rightarrow$ 5 Categories

- **Category 1**: Applications processing Large Frontiers are *good candidates*
- **Category 2**: Symmetric bipartite graphs require *bi-partiteness aware reordering*
- **Category 3**: Applications processing small frontiers *offer limited opportunity*
- **Category 4**: Reordering for Push-style applications introduces *false-sharing*
- **Category 5**: Reordering *affects convergence* for applications with ID-dependent computations
Category I - Applications Processing a Large Fraction Of Edges

- PageRank (Ligra & Gap)
- Graph Radii Estimation (Ligra)
Category I - Applications Processing a Large Fraction Of Edges

PR-L

PR-G

Radii-L

Speedup

0.0

0.5

1.0

1.5

2.0

DBP  GPL  PLD  KRON  TWIT  MPI  WEB  SD1

Speedup

0.0

0.5

1.0

1.5

2.0

DBP  GPL  PLD  KRON  TWIT  MPI  WEB  SD1

2.7x

Speedup

0.0

0.5

1.0

1.5

2.0

DBP  GPL  PLD  KRON  TWIT  MPI  WEB  SD1

Rabbit

HubSort

HubCluster
Observation 1: LWR provides *end-to-end* speedups in some cases.
Observation 1: LWR provides *end-to-end* speedups in some cases

Observation 2: Maximum speedups from HubSort > HubCluster
Category I - Applications Processing a Large Fraction Of Edges

Observation 1: LWR provides *end-to-end* speedups in some cases

Observation 2: Maximum speedups from HubSort > HubCluster

Observation 3: Reordering Overhead is HubSort > HubCluster
Observation 1: LWR provides *end-to-end* speedups in some cases

Observation 2: Maximum speedups from HubSort > HubCluster

Observation 3: Reordering Overhead is HubSort > HubCluster

Observation 4: HubSort strikes a balance between effectiveness and overhead
Category II - Executions On Symmetric Bipartite Graphs

- Collaborative Filtering (Ligra)
Category II - Executions On Symmetric Bipartite Graphs

![Graph showing speedup comparison between Rabbit, HubSort, and HubCluster on various graphs: DBLP, DISC, WIKI, EDIT, LIVEJ, ORK, DELI, TRACK. The x-axis represents different datasets, and the y-axis represents speedup. The graph illustrates performance variation across these datasets.]
Category II - Executions On Symmetric Bipartite Graphs

Surprising trend: HubSort causes net slowdowns
Category II - Reason For Slowdown With HubSort
Category II - Reason For Slowdown With HubSort

Heatmap showing the part that each vertex belongs to
Category II - Reason For Slowdown With HubSort

for $v$ in $G$:
    for $u$ in $\text{neigh}(v)$:
        process(..., vtxData[u], ...)

Heatmap showing the part that each vertex belongs to

[0, V)
Category II - Reason For Slowdown With HubSort

Range of Irregular accesses to vtxData = #nodes in a part

for v in G:
    for u in neigh(v):
        process(..., vtxData[u], ...)

[0, V)
Category II - Reason For Slowdown With HubSort

Vertices from different parts assigned consecutive IDs

\[ \text{Original Ordering} \]

Part 1 IDs

Part 2 IDs

Hub Sorting

\[ [0, V) \]

for \( v \) in \( G \):
  for \( u \) in neigh(\( v \)):
    process(\( ... \), vtxData[\( u \]), ...)
Category II - Reason For Slowdown With HubSort

for \( v \) in \( G \):
   for \( u \) in \( \text{neigh}(v) \):
      \( \text{process}(..., \text{vtxData}[u], ...) \)

Opportunity: Bipartiteness-aware Graph Reordering Techniques
Category III - Applications Processing a Small Fraction of Edges

- Betweenness Centrality
- BFS
- K-Core Decomposition
Category III - Applications Processing a Small Fraction of Edges

Low speedup even without Reordering overheads
Category III - Applications Processing a Small Fraction of Edges

- BC-L
- BFS-L
- KCore-L

Avg. Percentage of edges processed each iteration:

- Page Rank: 100%
- Radii: 75%
- BC-L: 50%
- BFS-L: 25%
- K-Core-L: 0%
Category III - Applications Processing a Small Fraction of Edges

For BC-L, BFS-L, and KCore-L, we see graphs that illustrate the speedup of various applications (DBP, GPL, PLD, KRON, TWIT, MPI, WEB, SD1) across different algorithms and datasets.

The right-hand graph shows the average percentage of edges processed each iteration for Page Rank, Radii, BC-L, BFS-L, and K-Core-L.
Category III - Applications Processing a Small Fraction of Edges

- **BC-L**
- **BFS-L**
- **KCore-L**

Average Percentage of edges processed each iteration:

- Page Rank: 100%
- Radii: 75%
- BC-L: 25%
- BFS-L: 0%
Category III - Applications Processing a Small Fraction of Edges

- Limited reuse in vtxData accesses
- Lower headroom for reordering
Outline

❖ Poor Locality of Graph Processing Applications ✓

❖ Improving locality through Graph Reordering ✓

❖ Graph Reordering Challenge - Application and Input-dependent Speedups ✓

❖ When is Graph Reordering an Optimization?
  ➢ Characterization Space ✓
  ➢ Which Applications benefit from Reordering? ✓
  ➢ Which Input Graphs benefit from Reordering?

❖ Selective Graph Reordering
Outline

❖ Poor Locality of Graph Processing Applications ✔

❖ Improving locality through Graph Reordering ✔

❖ Graph Reordering Challenge - Application and Input-dependent Speedups ✔

❖ When is Graph Reordering an Optimization?
  ➢ Characterization Space ✔
  ➢ Which Applications benefit from Reordering? ✔
  ➢ Which Input Graphs benefit from Reordering?

❖ Selective Graph Reordering
Speedup From HubSorting Varies Across Inputs

Category I Application

HubSort

Input Graphs

Net Speedup (App - PR)

PLD  KRON  TWIT  MPI  SD1  DBP  GPL  WEB
Speedup From HubSorting Varies Across Inputs

Need to **predict speedup** from HubSorting
AND
**selectively** perform HubSorting
Understanding Performance Improvement From HubSorting

 vtxData

 Cache line
Understanding Performance Improvement From HubSorting

Hub Vertices

Cache line

Layout of hubs in original ordering
Understanding Performance Improvement From HubSorting

- Layout of hubs in original ordering
- Layout of hubs after reordering vertices

Hub Vertices

Cache line

HubSort
Understanding Performance Improvement From HubSorting

Hub Sorting is effective for Graphs with:

- **Property #1**: Skew in the degree-distribution (Presence of Hubs)
- **Property #2**: Sparsely distributed hub vertices (Quality of original ordering)
Packing Factor - A Measure of Hub Density

**Packing Factor** is a measure of how densely the hubs are packed after HubSorting.

![Diagram showing original and reordered layouts]

- **Layout of the original ordering of vertices**
- **Layout after reordering vertices by HubSorting**

**Packing Factor** can be calculated as follows:

\[
Packing \ Factor = \frac{5}{2} = 2.5
\]
Packing Factor Can Predict Speedup From HubSorting

Pearson Correlation = 0.92
Packing Factor Can Predict Speedup From HubSorting

Pearson Correlation = 0.92

Speedup from HubSorting for a given pair of (Application, Input Graph)
Packing Factor Can Predict Speedup From HubSorting

Pearson Correlation = 0.92

Packing Factor is a good predictor for Speedup
Outline

❖ Poor Locality of Graph Processing Applications ✔

❖ Improving locality through Graph Reordering ✔

❖ Graph Reordering Challenge - Application and Input-dependent Speedups ✔

❖ When is Graph Reordering an Optimization?
  ➢ Characterization Space ✔
  ➢ Which Applications benefit from Reordering? ✔
  ➢ Which Input Graphs benefit from Reordering? ✔

❖ Selective Graph Reordering
Outline

- Poor Locality of Graph Processing Applications ✔
- Improving locality through Graph Reordering ✔
- Graph Reordering Challenge - Application and Input-dependent Speedups ✔
- When is Graph Reordering an Optimization?
  ➢ Characterization Space ✔
  ➢ Which Applications benefit from Reordering? ✔
  ➢ Which Input Graphs benefit from Reordering? ✔
- Selective Graph Reordering
Selective Graph Reordering

\[ G' = \text{HubSort}(G) \]

\[ \text{Process}(G') \]
Selective Graph Reordering

Net Speedup from Unconditionally HubSorting (PR-G)

Increasing order of Packing Factor

G' = HubSort(G)
Process(G')
Selective Graph Reordering

Net Speedup from Unconditionally HubSorting (PR-G)

G' = HubSort(G)
Process(G')

PF = ComputePF(G)
if (PF > 4):
  G' = HubSort(G)
  Process(G')
else:
  Process(G)
Selective Graph Reordering

\[ G' = \text{HubSort}(G) \]
\[ \text{Process}(G') \]

\[ \text{PF} = \text{ComputePF}(G) \]
\[ \text{if } (\text{PF} > 4) : \]
\[ G' = \text{HubSort}(G) \]
\[ \text{Process}(G') \]
\[ \text{else} : \]
\[ \text{Process}(G) \]
Selective Graph Reordering

Selective Reordering avoids slowdowns
Selective Graph Reordering

Computing Packing Factor does not degrade performance
Selective Graph Reordering

Net Speedup from Unconditionally HubSorting (PR-G)

Net Speedup from Selective HubSorting (PR-G)

Selective Reordering is a viable Optimization
Conclusions

❖ Graph Reordering does not benefit all Application and Input Graphs

❖ Opportunity to design new Reordering techniques for specific applications

❖ Packing Factor enables Selective Graph Reordering
Source Code Available

- Includes code for:
  - Packing Factor
  - Lightweight Reordering Techniques
  - Selective HubSorting

- Open sourced at -
  - https://github.com/CMUAbstract/Graph-Reordering-IISWC18
Thank You!
When Is Graph Reordering An Optimization?

A Cross Application and Input Graph Study on the Effectiveness of Lightweight Graph Reordering

Vignesh Balaji  Brandon Lucia
Use-cases Where Reordering Overhead Cannot be Amortized

Sophisticated Reordering Techniques are impractical for cases where graph is processed only a few times

*Left fig. from “Chronos: A Graph Engine for Temporal Graph Analysis” EuroSys 2014; Right fig. from “Graph Evolution: Densification and Shrinking Diameters” TKDD 2007*
Sophisticated Reordering Techniques Impose *High* Overhead

<table>
<thead>
<tr>
<th></th>
<th>gplus</th>
<th>web</th>
<th>pld-arc</th>
<th>twitter</th>
<th>kron26</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Time (baseline)</td>
<td>6.40s</td>
<td>7.84s</td>
<td>12.40s</td>
<td>21.3s</td>
<td>12.88s</td>
</tr>
<tr>
<td>Run Time (Gorder)</td>
<td>4.48s</td>
<td>7.77s</td>
<td>6.54s</td>
<td>13.09s</td>
<td>5.01s</td>
</tr>
<tr>
<td>Overhead (Gorder)</td>
<td>1685.9s</td>
<td>459.8s</td>
<td>7255s</td>
<td>25200s</td>
<td>53234s</td>
</tr>
<tr>
<td>#Runs to amortize ovhd</td>
<td>873</td>
<td>6477</td>
<td>1237</td>
<td>3072</td>
<td>6771</td>
</tr>
</tbody>
</table>

**Assumption:** Reordered graph will be processed multiple times
Irregular Accesses Have Poor Temporal And Spatial Locality

for v in G:
    for u in neigh(v):
        process(..., vData[u],...)

```
    Miss
    vData[3]
    2 3
    0 1
    LLC

    Miss
    vData[0]
    2 3
    0 1
    LLC

    Miss
    vData[7]
    6 7
    0 1
    LLC

    Miss
    vData[3]
    6 7
    2 3
    LLC

    Miss
    vData[1]
    0 1
    2 3
    LLC
```

Offsets
0 1 4 8 8 11 ...

Coordinates
3 0 7 3 1 3 7 9 0 3 5 ...

Time
Irregular Accesses Have Poor Temporal And Spatial Locality

for v in G:
    for u in neigh(v):
        process(..., vData[u],...)

Offsets

Coordinates


Time

Miss  Miss  Miss  Miss  Miss  Hit

2 3 0 1 2 3 6 7 2 3 0 1 2 3

LLC  LLC  LLC  LLC  LLC  LLC
Irregular Accesses Have Poor Temporal And Spatial Locality

for v in G:
    for u in neigh(v):
        process(..., vData[u],...)
Graph Applications

Ligra
- Page Rank
- Page Rank-Delta
- SSSP – Bellman Ford
- Collaborative Filtering
- Radii
- Betweenness Centrality
- BFS
- Kcore
- Maximal Independent Set
- Connected Components

GAP
- Page Rank
- SSSP – Delta Stetting
- Betweenness Centrality
- BFS
- Connected Components

11 Distinct Algorithms
HW Platform

- Dual-Socket Intel Xeon E5-2660v4 processors
- 14 cores per Socket (2HT/core)
- 35 MB Last Level Cache per processor
- 64 GB of main memory

![Diagram of HW Platform showing dual socket with 14 cores each, 35 MB Last Level Cache, and 32 GB DRAM per core.](image-url)

Socket 1: Core 1, Core 2, ..., Core 14

35MB Last Level Cache

32GB DRAM

Socket 2: Core 1, Core 2, ..., Core 14

35MB Last Level Cache

32GB DRAM
## Input Graphs

<table>
<thead>
<tr>
<th></th>
<th>DBP</th>
<th>GPL</th>
<th>PLD</th>
<th>KRON</th>
<th>TWIT</th>
<th>MPI</th>
<th>WEB</th>
<th>SD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>V</td>
<td>$ (in M)</td>
<td>18.27</td>
<td>28.94</td>
<td>42.89</td>
<td>33.55</td>
<td>61.58</td>
<td>52.58</td>
</tr>
<tr>
<td>$</td>
<td>E</td>
<td>$ (in B)</td>
<td>0.172</td>
<td>0.462</td>
<td>0.623</td>
<td>1.047</td>
<td>1.468</td>
<td>1.963</td>
</tr>
<tr>
<td>vData Sz (MB)</td>
<td>146.16</td>
<td>231.52</td>
<td>343.12</td>
<td>268.4</td>
<td>498.64</td>
<td>420.64</td>
<td>405.12</td>
<td>759.6</td>
</tr>
<tr>
<td>CSR Sz (GB)</td>
<td>1.41</td>
<td>3.66</td>
<td>4.96</td>
<td>8.05</td>
<td>11.34</td>
<td>15.02</td>
<td>14.75</td>
<td>15.13</td>
</tr>
</tbody>
</table>
## Input Graphs

<table>
<thead>
<tr>
<th></th>
<th>DBP</th>
<th>GPL</th>
<th>PLD</th>
<th>KRON</th>
<th>TWIT</th>
<th>MPI</th>
<th>WEB</th>
<th>SD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>V</td>
<td>) (in M)</td>
<td>18.27</td>
<td>28.94</td>
<td>42.89</td>
<td>33.55</td>
<td>61.58</td>
<td>52.58</td>
</tr>
<tr>
<td>(</td>
<td>E</td>
<td>) (in B)</td>
<td>0.172</td>
<td>0.462</td>
<td>0.623</td>
<td>1.047</td>
<td>1.468</td>
<td>1.963</td>
</tr>
<tr>
<td>vData Sz (MB)</td>
<td>146.16</td>
<td>231.52</td>
<td>343.12</td>
<td>268.4</td>
<td>498.64</td>
<td>420.64</td>
<td>405.12</td>
<td>759.6</td>
</tr>
<tr>
<td>CSR Sz (GB)</td>
<td>1.41</td>
<td>3.66</td>
<td>4.96</td>
<td>8.05</td>
<td>11.34</td>
<td>15.02</td>
<td>14.75</td>
<td>15.13</td>
</tr>
</tbody>
</table>

Irregular working set size >> Aggregate LLC Capacity
### Input Graphs

We use the original ordering of Input Graphs.

<table>
<thead>
<tr>
<th>V (in M)</th>
<th>DBP</th>
<th>GPL</th>
<th>PLD</th>
<th>KRON</th>
<th>TWIT</th>
<th>MPI</th>
<th>WEB</th>
<th>SD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.27</td>
<td>28.94</td>
<td>42.89</td>
<td>33.55</td>
<td>61.58</td>
<td>52.58</td>
<td>50.64</td>
<td>94.95</td>
<td></td>
</tr>
<tr>
<td>0.172</td>
<td>0.462</td>
<td>0.623</td>
<td>1.047</td>
<td>1.468</td>
<td>1.963</td>
<td>1.93</td>
<td>1.937</td>
<td></td>
</tr>
<tr>
<td>vData Sz (MB)</td>
<td>146.16</td>
<td>231.52</td>
<td>343.12</td>
<td>268.4</td>
<td>498.64</td>
<td>420.64</td>
<td>405.12</td>
<td>759.6</td>
</tr>
<tr>
<td>CSR Sz (GB)</td>
<td>1.41</td>
<td>3.66</td>
<td>4.96</td>
<td>8.05</td>
<td>11.34</td>
<td>15.02</td>
<td>14.75</td>
<td>15.13</td>
</tr>
</tbody>
</table>

Irregular working set size >> Aggregate LLC Capacity
Lightweight Reordering Can Provide End-to-end Speedups

Figure from “Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis” IPDPS 2016
Lightweight Reordering Can Provide End-to-end Speedups

Reordering techniques exploiting power-law distributions and community structure can have low-overheads

Figure from “Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis” IPDPS 2016
Surprising trends:
- HubSort offers the least performance benefits
- HubSort causes slowdowns
Category II - Reason For Slowdown With HubSort

Assigning vertices from each part of the graph a contiguous range is good for temporal locality

```
for v in G:
    for u in neigh(v):
        process(..., vData[u], ...)
```

Assigning vertices from each part of the graph a contiguous range of IDs

Base Ordering

Hub Sorting

Hub Clustering

Need a simple mechanism to assign hub vertices from the same part a contiguous range of IDs
**Category IV - Push-based Graph Applications**

**Push-phase**

\[
\text{parallel for src in Frontier:} \\
\text{for dst in outNeigh(v):} \\
\quad \text{atomic}\{\text{parent}[\text{dst}] = \text{src}\}
\]

- **+ Work Efficient execution**
- **- Overhead of synchronization**

**Pull-phase**

\[
\text{parallel for dst in G:} \\
\text{for src in inNeigh(v):} \\
\quad \text{if src in Frontier:} \\
\quad \quad \text{parent}[\text{dst}] = \text{src}
\]

- **+ No synchronization required**
- **- Work-inefficient (iterate over all Vertices)**
Category IV - Push-based Graph Applications

Reordering causes an increase in false-sharing

<table>
<thead>
<tr>
<th>Category</th>
<th>Rabbit</th>
<th>GPL</th>
<th>PLD</th>
<th>KRON</th>
<th>TWIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR-δ-L</td>
<td>1.11x</td>
<td>1.53x</td>
<td>1.53x</td>
<td>0.92x</td>
<td>1.26x</td>
</tr>
<tr>
<td></td>
<td>0.94x</td>
<td>0.99x</td>
<td>1.43x</td>
<td>1.77x</td>
<td>1.77x</td>
</tr>
<tr>
<td></td>
<td>1.06x</td>
<td>1.01x</td>
<td>1.24x</td>
<td>1.46x</td>
<td>1.27x</td>
</tr>
<tr>
<td>SSSP-L</td>
<td>0.87x</td>
<td>1.36x</td>
<td>1.2x</td>
<td>0.95x</td>
<td>0.97x</td>
</tr>
<tr>
<td></td>
<td>1.02x</td>
<td>1.14x</td>
<td>1.58x</td>
<td>2.0x</td>
<td>1.4x</td>
</tr>
<tr>
<td></td>
<td>1.14x</td>
<td>1.07x</td>
<td>1.47x</td>
<td>1.58x</td>
<td>1.4x</td>
</tr>
</tbody>
</table>

LWR favors pull-style graph applications
Category V - LWR can affect convergence

inline bool update (uintE s, uintE d) {
    //if neighbor is in MIS, then we are out
    if (flags[d] == IN) {
        if (flags[s] != OUT) flags[s] = OUT;
    }
    //if neighbor has higher priority (lower ID) and is undecided, then so are we
    else if ((d < s) && flags[s] == CONDITIONALLY_IN && flags[d] < OUT)
        flags[s] = UNDECIDED;
    return 1;
}

Vertex IDs influence amount of work done each iteration
Category V - LWR can affect convergence

Increase in Iterations until convergence due to LWR

Opportunity to accelerate convergence by reordering vertices
The Need For Selective Lightweight Reordering

Unconditionally performing LWR causes net slowdowns on some input graphs.

Completely avoid LWR misses speedups up to 1.8x.

Need to predict speedup from LWR for an input graph and only selectively perform LWR.
Using Packing Factor for Selective Reordering

Selective Reordering avoids slowdowns for graphs with low Packing Factor
Using Packing Factor for Selective Reordering

The low overhead of Packing Factor computation does not sacrifice speedup for high Packing Factor graphs.
Speedups From HubSorting Are Due To Locality Improvements
Computing Packing Factor

Algorithm 2 Computing the Packing Factor of a graph

1: procedure COMPUTEPACKINGFACTOR(G)
2:   numHubs ← 0
3:   hubWSet_Original ← 0
4:   for CacheLine in vDataLines do
5:       containsHub ← False
6:       for vtx in CacheLine do
7:           if ISHUB(vtx) then
8:               numHubs += 1
9:               containsHub ← True
10:          if containsHub = True then
11:              hubWSet_Original += 1
12:      hubWSet_Sorted ← CEIL(numHubs/VtxPerLine)
13:  PackingFactor ← hubWSet_Original/hubWSet_Sorted
14: return PackingFactor