Combining Duplication & Reordering to Accelerate Parallel Graph Processing

Vignesh Balaji
Brandon Lucia
Graph Processing Has Many Important Applications

- Path Planning
- Social network analysis
- Recommender systems
- Graph Convolutional Networks
- Data Mining
# Large Graphs Can Be Processed in a Single Node

<table>
<thead>
<tr>
<th></th>
<th>Cores</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 - 64</td>
<td>(100s GB - TBs)</td>
</tr>
</tbody>
</table>

- Cores: 8 - 64
- Memory: (100s GB - TBs)
Large Graphs Can Be Processed in a Single Node

Uncompressed Facebook Friend Graph is ~1.5TB

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cores</strong></td>
<td>8 - 64</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>(100s GB - TBs)</td>
</tr>
</tbody>
</table>
Large Graphs Can Be Processed in a Single Node

Uncompressed Facebook Friend Graph is ~1.5TB

<table>
<thead>
<tr>
<th>Cores</th>
<th>8 - 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>(100s GB - TBs)</td>
</tr>
</tbody>
</table>

For Graphs that fit in main memory, single node graph processing is more efficient

Outline

❖ Single-node Graph Processing is Sub-optimal
❖ Existing Optimizations have Overheads
❖ RADAR: Combining the Benefits of Duplication and Reordering
❖ Advantages of RADAR over Push-Pull Direction Switching
Outline

❖ Single-node Graph Processing is Sub-optimal ⇐
  ➢ Expensive Atomic Updates ⇒ Coherence & serialization overheads
  ➢ Poor LLC locality ⇒ DRAM accesses dominate runtime

❖ Existing Optimizations have Overheads

❖ RADAR: Combining the Benefits of Duplication and Reordering

❖ Advantages of RADAR over Push-Pull Direction Switching
Shared-memory Graph Processing Overview

Offsets Array (OA)

Coordinates Array (CA)

Out-neighbors
Shared-memory Graph Processing Overview
Shared-memory Graph Processing Overview

Vertex Centric Graph Processing (Push)

```python
for src in Frontier:
    for dst in out_neigh(src):
        vtxData[dst] += auxData[src]
```
The Need for Atomic Updates

**Vertex Centric Graph Processing (Push)**

for src in Frontier:
    for dst in out_neigh(src):
        vtxData[dst] += auxData[src]

**parallel_for** src in Frontier:
    for dst in out_neigh(src):
        atomic {vtxData[dst] += auxData[src]}
The Need for Atomic Updates

Vertex Centric Graph Processing (Push)

parallel_for src in Frontier:
  for dst in out_neigh(src):
    atomic {vtxData[dst] += auxData[src]}

```python
for src in Frontier:
  for dst in out_neigh(src):
    vtxData[dst] += auxData[src]
```
Bottleneck #1: Atomic Updates Hurt Performance

Slowdown from Atomic Updates

- PR-Delta
- BC
- Radii
- BFS

- NO ATOMICS
- BASELINE (WITH ATOMICS)
Bottleneck #1: Atomic Updates Hurt Performance

Slowdown from Atomic Updates

- PR-Delta
- BC
- Radii
- BFS

NO ATOMICS vs BASELINE (WITH ATOMICS)
Source of Poor Locality ⇒ Irregular Memory Accesses

Vertex Centric Graph Processing (Push)

```python
for src in Frontier:
    for dst in out_neigh(src):
        vtxData[dst] += auxData[src]
```
Source of Poor Locality $\implies$ Irregular Memory Accesses

Vertex Centric Graph Processing (Push)

```
for src in Frontier:
    for dst in out_neigh(src):
        vtxData[dst] += auxData[src]
```
Source of Poor Locality $\Rightarrow$ Irregular Memory Accesses

Vertex Centric Graph Processing (Push)

for src in Frontier:
  for dst in out_neigh(src):
    vtxData[dst] += auxData[src]
Vertex Centric Graph Processing (Push)

```
for src in Frontier:
  for dst in out_neigh(src):
    vtxData[dst] += auxData[src]
```

Source of Poor Locality \(\Rightarrow\) Irregular Memory Accesses

Size of vtxData \(~ 200MB\)
Size of Typical LLCs \(~ 32MB\)
Bottleneck #2: Graph Applications are DRAM-latency bound

**Problem:** Poor LLC locality $\Rightarrow$ Many long-latency DRAM accesses

---

Figure from “Optimizing Cache Performance for Graph Analytics” ArXiv v1;
Outline

❖ Single-node Graph Processing is Sub-optimal ✔
   ➢ Expensive Atomic Updates ⇒ Coherence & serialization overheads
   ➢ Poor LLC locality ⇒ DRAM accesses dominate runtime

❖ Existing Optimizations incur Overheads

❖ RADAR: Combining the Benefits of Duplication and Reordering

❖ Advantages of RADAR over Push-Pull Direction Switching
Outline

❖ Single-node Graph Processing is Sub-optimal ✔

❖ Existing Optimizations incur Overheads ⇨
   ➢ Duplication incurs Hub Identification Overhead
   ➢ Reordering incurs False Sharing Overhead

❖ RADAR: Combining the Benefits of Duplication and Reordering

❖ Advantages of RADAR over Push-Pull Direction Switching
Optimizations for Eliminating Performance Bottlenecks

❖ **Data Duplication**: Improves Scalability

❖ **Graph Reordering**: Reduces DRAM accesses

**BUT**

❖ **Duplication** incurs Hub Identification Overhead

❖ **Reordering** incurs False Sharing Overhead
Optimization #1: Data Duplication

parallel_for src in Frontier:
   for dst in neigh(src):
      atomic {vtxData[dst] += auxData[src]}

vtxData
Optimization #1: Data Duplication

parallel_for src in Frontier:
    for dst in neigh(src):
        atomic {vtxData[dst] += auxData[src]}

parallel_for src in Frontier:
    for dst in neigh(src):
        vtxDataDup[tid][dst] += auxData[src]
Naive Duplication Imposes High Memory Overhead

For a graph with 64M vertices, 4B / vtx, 16 threads

Memory footprint after Duplication = 4GB!
Power-Law Graphs Allow Memory-Efficient Duplication

Air Traffic Network

Hubs
Power-Law Graphs Allow Memory-Efficient Duplication

Majority of Atomic Updates are to Hub Vertices

Duplicate only the Hub Vertex Data
HUBDUP: Duplication for Power-Law Graphs

Hubs

vtxData

Thread 0
LocalCopies[0]

... (Multiple threads)

Thread N
LocalCopies[N]
HUBDUP: Duplication for Power-Law Graphs

1. Find index into local copy
2. Update local copy \textbf{without} atomics

Update \(\text{vtxData}[dst]\)

\textbf{Is} \(dst\) a hub?

\textbf{Y} \quad \textbf{N}

Update \(\text{vtxData}[dst]\) \textbf{with} atomics
Update (vtxData[dst])

Is dst a hub?

Y

1. Find index into local copy
2. Update local copy without atomics

N

Update vtxData[dst] with atomics
**HUBDUP Incurs Overheads**

**Overhead #1: Identifying Hub Vertices**

- **Update** (vtxData[$dst$])
  - Is $dst$ a hub?
    - **Y**
      1. Find index into local copy
      2. Update local copy *without* atomics
    - **N**
      Update vtxData[$dst$] *with* atomics

**Vertex ID** ➔ **isHub?** ➔ **True/False**

**Hubs** ➔ **vtxData**
HUBDUP Incurs Overheads

Update (vtxData[dst])

Is dst a hub?

Y

1. Find index into local copy
2. Update local copy without atomics

N

Update vtxData[dst] with atomics

Overhead #2: Indexing into thread-local copies

(Hub) Vertex ID → Map → Local Index → LocalCopies[k]
Summary Of Overheads In HUBDUP

Overhead #1:  
*Hub Identification*
  
Vertex ID → **isHub?** → True/False

Overhead #2:  
*Local-Copy Indexing*
  
(Hub) Vertex ID → **Map** → Local Index
Summary Of Overheads In HUBDUP

Overhead #1:  
**Hub Identification**

- Vertex ID \(\rightarrow\) \text{isHub?} \(\rightarrow\) True/False

Overhead #2:  
**Local-Copy Indexing**

- (Hub) Vertex ID \(\rightarrow\) \text{Map} \(\rightarrow\) Local Index

**Problem:** Hubs can have *arbitrary* vertex IDs
Optimizations for eliminating performance bottlenecks

❖ **Data Duplication**: Improves Scalability ✔

❖ **Graph Reordering**: Reduces DRAM accesses

BUT

❖ **HUBDUP** incurs Hub Identification Overhead ✔

❖ **Reordering** incurs False Sharing Overhead
Optimization #2: Graph Reordering

Irregular accesses & Large working set

poor **spatial & temporal** locality
Optimization #2: Graph Reordering

Hubs

Degree Sorting
Re-assign vertex IDs based on degrees

vtxData

Hubs

vtxData
Optimization #2: Graph Reordering

Hubs

Re-assign vertex IDs based on degrees

Degree Sorting

vtxData

Hubs

vtxData

Hub 0  Hub 1  ...  Hub N

Cache Line

Spatial Locality

Temporal Locality

Temporal Locality
Degree Sorting Is Effective For Serial Graph Processing

Serial Executions

<table>
<thead>
<tr>
<th></th>
<th>Original Order</th>
<th>Degree Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR-Delta</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radii</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Speedup over Push

0 0.25 0.5 0.75 1 1.25 1.5

PR-Delta  BC  BFS  Radii
Degree Sorting Introduces False Sharing

Degree Sorting

Re-assign vertex IDs based on degrees

Cache Line
Degree Sorting Introduces False Sharing

Coherence operates on cache-line granularity

**Degree Sorting**
Re-assign vertex IDs based on degrees

Hub 0  Hub 1  ...  Hub N

Cache Line
Degree Sorting Introduces False Sharing

Degree Sorting

Re-assign vertex IDs based on degrees

Cache Line

Coherence operates on cache-line granularity
False Sharing Hurts Performance

Serial Executions

Parallel Executions (56 Threads)
Optimizations for eliminating performance bottlenecks

❖ Data Duplication : Improves Scalability ✔
❖ Graph Reordering : Reduces DRAM accesses ✔

BUT

❖ HUBDUP incurs Hub Identification Overhead ✔
❖ Reordering incurs False Sharing Overhead ✔
Summary of Duplication And Reordering

**HUBDUP**
- No Atomics for Hub Vertices
- Overhead of identifying Hubs

**Degree Sorting**
- Improves Cache Locality
- Introduces False Sharing
Outline

❖ Single-node Graph Processing is Sub-optimal ✓

❖ Existing Optimizations have Overheads ✓
  ➢ Duplication incurs Hub Identification Overhead
  ➢ Reordering incurs False Sharing Overhead

❖ RADAR: Combining the Benefits of Duplication and Reordering

❖ Advantages of RADAR over Push-Pull Direction Switching
Outline

❖ Single-node Graph Processing is Sub-optimal ✓

❖ Existing Optimizations have Overheads ✓

❖ RADAR: Combining the Benefits of Duplication and Reordering ⇅
  ➢ HUBDUP & Degree Sorting are Mutually Enabling
  ➢ RADAR = HUBDUP + Degree Sorting
  ➢ RADAR outperforms HUBDUP & Degree Sorting

❖ Advantages of RADAR over Push-Pull Direction Switching
HUBDUP And Degree Sorting Are Mutually Enabling
Degree Sorting Improves HUBDUP

**Ovhd #1:**  
*Hub Identification*

Vtx ID → **isHub?** → True/False

**Ovhd #2:**  
*Local-Copy Indexing*

(Hub) Vtx ID → **Map** → Local Index

-Hubs-

-vtxData-
Degree Sorting Improves HUBDUP

**Ovhd #1:** Hub Identification

- Vtx ID → isHub? → True/False
- Vtx ID → vtxID < thresh → True/False

**Ovhd #2:** Local-Copy Indexing

- (Hub) Vtx ID → Map → Local Index
- (Hub) Vtx ID → = → Local Index
HUBDUP Improves Degree Sorting

Hubs

vtxData

Tid 0

LocalCopies[0]

... 

LocalCopies[N]

Tid N
HUBDUP Improves Degree Sorting

Hubs

vhxData

LocalCopies[0] ...

LocalCopies[N]

Threads update a private copies of hubs ⇒ No False Sharing
Reordering Assisted Duplication/Duplication Assisted Reordering

HUBDUP

- No Atomics for Hub Vertices
- Costs incurred for detecting Hubs

Degree Sorting

- Improves Cache Locality
- Introduces False Sharing

RADAR

- No Atomics for Hub Vertices (with easy hub detection)
- Improves Cache Locality (without false-sharing for hubs)
Outline

❖ Single-node Graph Processing is Sub-optimal ✔

❖ Existing Optimizations have Overheads ✔

❖ RADAR: Combining the Benefits of Duplication and Reordering
  ➢ HUBDUP & Degree Sorting are Mutually Enabling
  ➢ RADAR = HUBDUP + Degree Sorting
  ➢ RADAR outperforms HUBDUP & Degree Sorting ⇀

❖ Advantages of RADAR over Push-Pull Direction Switching
Evaluation Space

Systems:
- Push (Baseline)
- + HUBDUP
- + Degree Sorting
- + RADAR

Applications:
- PageRank & PageRank-Delta
- Triangle Listing
- Betweenness Centrality
- BFS
- Graph Radius

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>E (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>
<td>1.93</td>
<td>1.937</td>
</tr>
</tbody>
</table>
Evaluation Platform

- 28 Cores / 56 Threads across 2 socket ⇒ All versions run with 56 threads
- 35MB LLC per socket ⇒ Hubs selected to fit LLC
- 64GB DRAM
RADAR Outperforms Both HUBDUP And Degree Sorting

Geo-Mean Speedup across all graphs

8.2x
RADAR Outperforms Both HUBDUP And Degree Sorting
RADAR Outperforms Both HUBDUP And Degree Sorting

Already Optimized with Test-and-Test-and-Set
Outline

❖ Single-node Graph Processing is Sub-optimal ✔

❖ Existing Optimizations have Overheads ✔

❖ RADAR: Combining the Benefits of Duplication and Reordering ✔
  ➢ HUBDUP & Degree Sorting are Mutually Enabling
  ➢ RADAR = HUBDUP + Degree Sorting
  ➢ RADAR outperforms HUBDUP & Degree Sorting

❖ Advantages of RADAR over Push-Pull Direction Switching
Outline

❖ Single-node Graph Processing is Sub-optimal ✔
❖ Existing Optimizations have Overheads ✔
❖ RADAR: Combining the Benefits of Duplication and Reordering ✔
❖ Advantages of RADAR over Push-Pull Direction Switching ⇆
  ➢ RADAR does not compromise work-efficiency
  ➢ RADAR can support larger input graphs
  ➢ RADAR incurs lower preprocessing overheads
State Of The Art: Push-Pull Direction Switching

**Push**

parallel_for src in Frontier:
for dst in out_neigh(src):
    atomic {vtxData[dst] += auxData[src]}

**Pull**

parallel_for dst in G:
for src in in_neigh(dst):
    if src in Frontier:
        vtxData[dst] += auxData[src]
State Of The Art: Push-Pull Direction Switching

**Push**

```
parallel_for src in Frontier:
  for dst in out_neigh(src):
    atomic {vtxData[dst] += auxData[src]}
```

**Pull**

```
parallel_for dst in G:
  for src in in_neigh(dst):
    if src in Frontier:
      vtxData[dst] += auxData[src]
```

Pull phase trade-off ⇒ work efficiency vs eliminating atomic updates
### Advantages of RADAR over Push-Pull

<table>
<thead>
<tr>
<th></th>
<th>Push-Pull Direction Switching</th>
<th>RADAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work-Efficiency</td>
<td>Work-inefficient</td>
<td>No change</td>
</tr>
<tr>
<td>Memory Footprint</td>
<td>2X ((\text{outCSR} + \text{inCSR}))</td>
<td>~1X ((\text{reordered outCSR}))</td>
</tr>
<tr>
<td>Preprocessing Cost</td>
<td>(O(E.\log E))</td>
<td>(O(V.\log V + E))</td>
</tr>
</tbody>
</table>
Performance Of RADAR Compared To Push-Pull

Geo-Mean Speedup across all graphs

8.2x

Speedup over Push

PR PR-Delta TriCnt BC BFS Radii

Push Push-Pull RADAR
Performance Of RADAR Compared To Push-Pull

![Bar chart showing speedup over push for PR, PR-Delta, TriCnt, BC, BFS, and Radii with RADAR showing 8.2x speedup compared to Push-Pull.]
Performance Of RADAR Compared To Push-Pull

Algorithms favoring Push-Pull

8.2x
Performance Of RADAR Compared To Push-Pull

8.2x Optimized with T&T&S

Speedup over Push

- PR
- PR-Delta
- TriCnt
- BC
- BFS
- Radii

Push Push-Pull RADAR
RADAR Can Support Larger Input Graphs

Push-Pull increases memory footprint by 2x
RADAR Can Support Larger Input Graphs

Push-Pull increases memory footprint by 2x

Speedups for the SDH Graph (102M vertices, 2B edges)
Outline

❖ Single-node Graph Processing is Sub-optimal ✔
❖ Existing Optimizations have Overheads ✔
❖ RADAR: Combining the Benefits of Duplication and Reordering ✔
❖ Advantages of RADAR over Push-Pull Direction Switching ✔
  ➢ RADAR does not compromise work-efficiency
  ➢ RADAR can support larger input graphs
  ➢ RADAR incurs lower preprocessing overheads
Conclusions

- Single-node Graph Processing Performance is Sub-optimal
- HUBDUP and Degree Sorting optimizations incur overheads
- RADAR offers the best of HUBDUP & Degree Sorting *without their overheads*
- RADAR has many advantages over the state-of-the-art Push-Pull optimization
Conclusions

❖ Single-node Graph Processing Performance is Sub-optimal
❖ HUBDUP and Degree Sorting optimizations incur overheads
❖ RADAR offers the best of HUBDUP & Degree Sorting \textit{without their overheads}
❖ RADAR has many advantages over the state-of-the-art Push-Pull optimization

\url{https://github.com/CMUAbstract/RADAR-Graph}
Thank You!
Combining Duplication & Reordering to Accelerate Parallel Graph Processing

Vignesh Balaji
vigneshb@andrew.cmu.edu

Brandon Lucia
blucia@andrew.cmu.edu

https://github.com/CMUAbstract/RADAR-Graph
Conclusions

❖ Single-node Graph Processing Performance is Sub-optimal
❖ HUBDUP and Degree Sorting optimizations incur overheads
❖ RADAR offers the best of HUBDUP & Degree Sorting *without their overheads*
❖ RADAR has many advantages over the state-of-the-art Push-Pull optimization

https://github.com/CMUAbstract/RADAR-Graph
Backup Slides
Importance of sizing hubs to LLC size
All Results for RADAR v/s HUBDUP & DegSort
All Results for RADAR v/s Push-Pull
RADAR is invariant to Graph Order
Push-Pull is better for BC

**Algorithm 1** Typical graph processing kernel

1: par_for src in Frontier do 
2: for dst in out_neigh(src) do 
3: AtomicUpd (vtxData[dst]), auxData[src])

**Algorithm 3** Pseudocode for push-phase of BC

1: par_for src in Frontier do 
2: for dst in out_neigh(src) do 
3: if Visited[dst] is True then 
4: AtomicUpd (vtxData[dst]), auxData[src])
RADAR Imposes Lower Preprocessing Overhead

All Systems require some form of preprocessing over input file (outCSR):

- **HUBDUP**: Populate bitvector, map, and inv_map
- **Degree-Sorting/RADAR**: Reorder input graph by decreasing degrees
- **Push-Pull**: Construct the inCSR of the graph

<table>
<thead>
<tr>
<th>System</th>
<th>Complexity</th>
<th>DBP</th>
<th>GPL</th>
<th>PLD</th>
<th>TWIT</th>
<th>MPI</th>
<th>KRON</th>
<th>WEB</th>
<th>SD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUBDUP</td>
<td>$O(V)$</td>
<td>0.06s</td>
<td>0.11s</td>
<td>0.14s</td>
<td>0.24s</td>
<td>0.22s</td>
<td>0.23s</td>
<td>0.19s</td>
<td>0.37s</td>
</tr>
<tr>
<td>DegSort/RADAR</td>
<td>$O(V \log V + E)$</td>
<td>0.88s</td>
<td>2.37s</td>
<td>2.29s</td>
<td>8.26s</td>
<td>19.06s</td>
<td>2.94s</td>
<td>3.49s</td>
<td>7.42s</td>
</tr>
<tr>
<td>Push-Pull</td>
<td>$O(E \log E)$</td>
<td>2.96s</td>
<td>7.03s</td>
<td>3.91s</td>
<td>9.68s</td>
<td>47.71s</td>
<td>0s</td>
<td>10.86s</td>
<td>12.51s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>HUBDUP</th>
<th>RADAR</th>
<th>Push-Pull</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.17s</td>
<td>3.93s</td>
<td>9.08</td>
</tr>
</tbody>
</table>