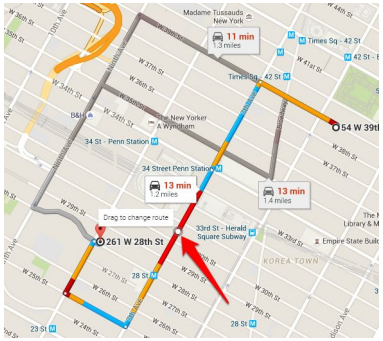
Combining Duplication & Reordering to Accelerate Parallel Graph Processing

Vignesh Balaji Brandon Lucia

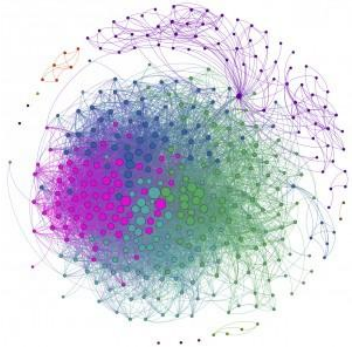
Electrical & Computer ENGINEERING

Carnegie Mellon

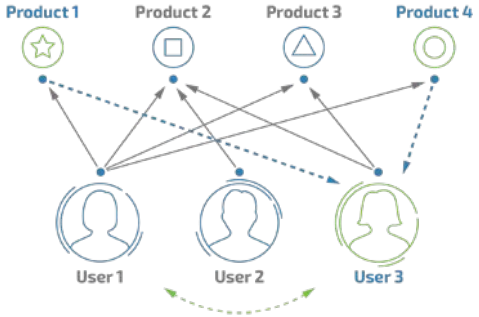
Graph Processing Has Many Important Applications



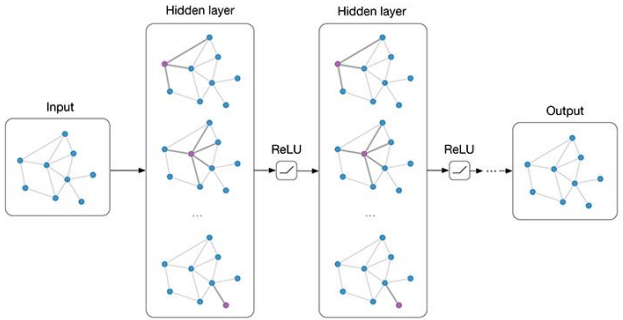
Path Planning



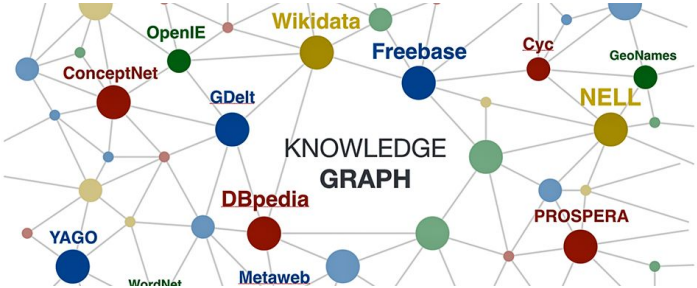
Social network analysis



Recommender systems



Graph Convolutional Networks



Data Mining

Large Graphs Can Be Processed in a Single Node

Cores	8 - 64 ⬆
Memory	(100s GB - TBs) ⬆

Large Graphs Can Be Processed in a Single Node

Cores	8 - 64
Memory	(100s GB - TBs)

Uncompressed Facebook Friend Graph is **~1.5TB**

Large Graphs Can Be Processed in a Single Node

Cores	8 - 64
Memory	(100s GB - TBs)

Uncompressed Facebook Friend Graph is **~1.5TB**

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

[1] "Scalability! But at what COST?" Usenix HOTOS'15

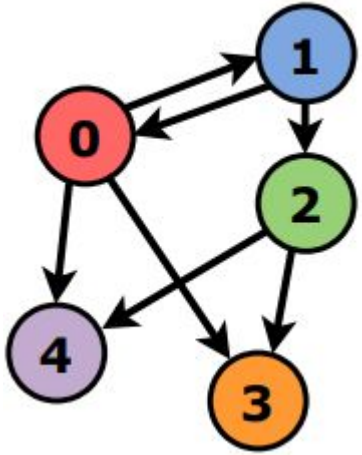
Outline

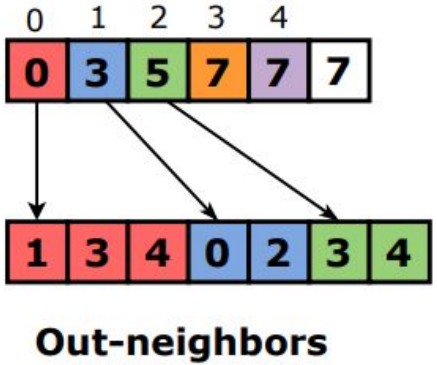
- **Single-node Graph Processing is Sub-optimal**
- Existing Optimizations have Overheads
- RADAR: Combining the Benefits of Duplication and Reordering
- Advantages of RADAR over Push-Pull Direction Switching

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

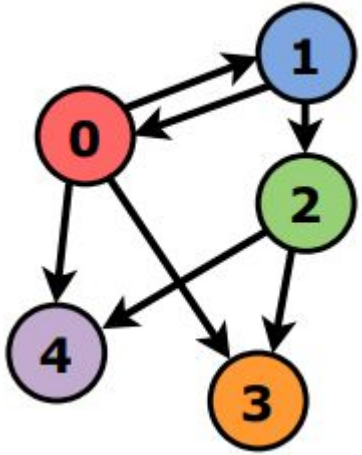


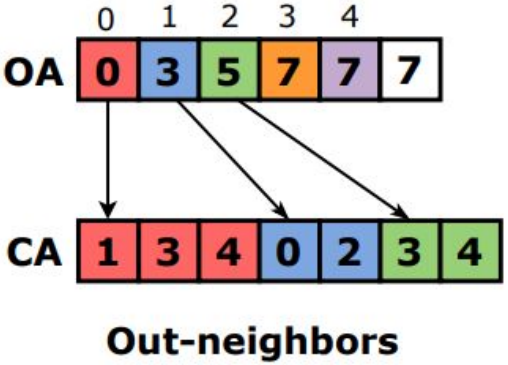


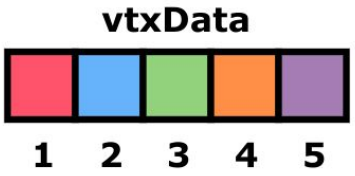
Offsets Array (OA)

Coordinates Array (CA)

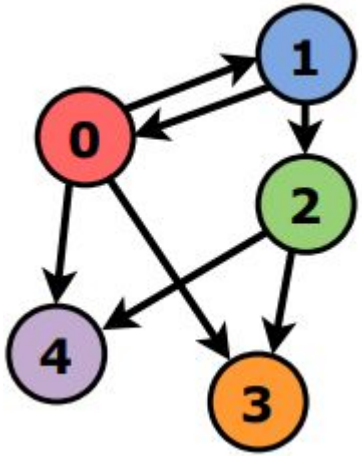
Shared-memory Graph Processing Overview





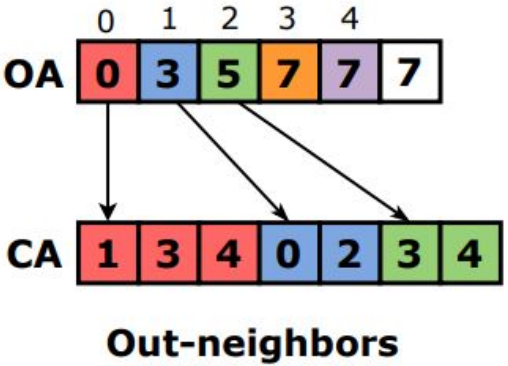


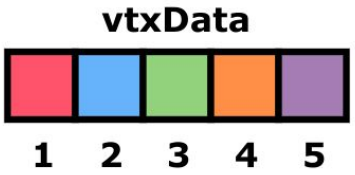
Shared-memory Graph Processing Overview



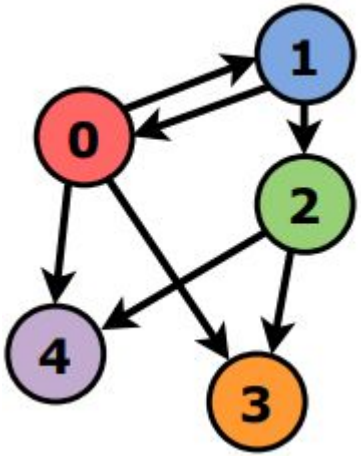
Vertex Centric Graph Processing (Push)

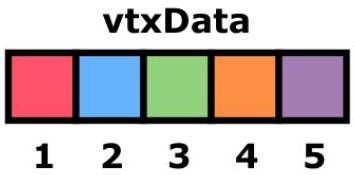
```
for src in Frontier:
    for dst in out_neigh(src):
        vtxData[dst] += auxData[src]
```





The Need for Atomic Updates



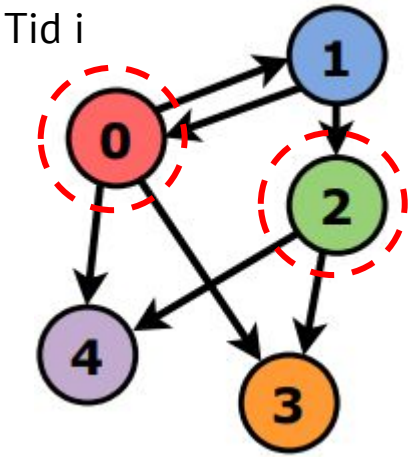


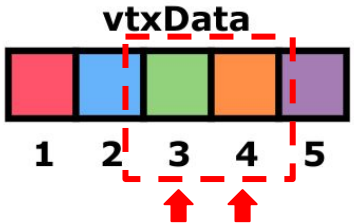
Vertex Centric Graph Processing (Push)

```
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



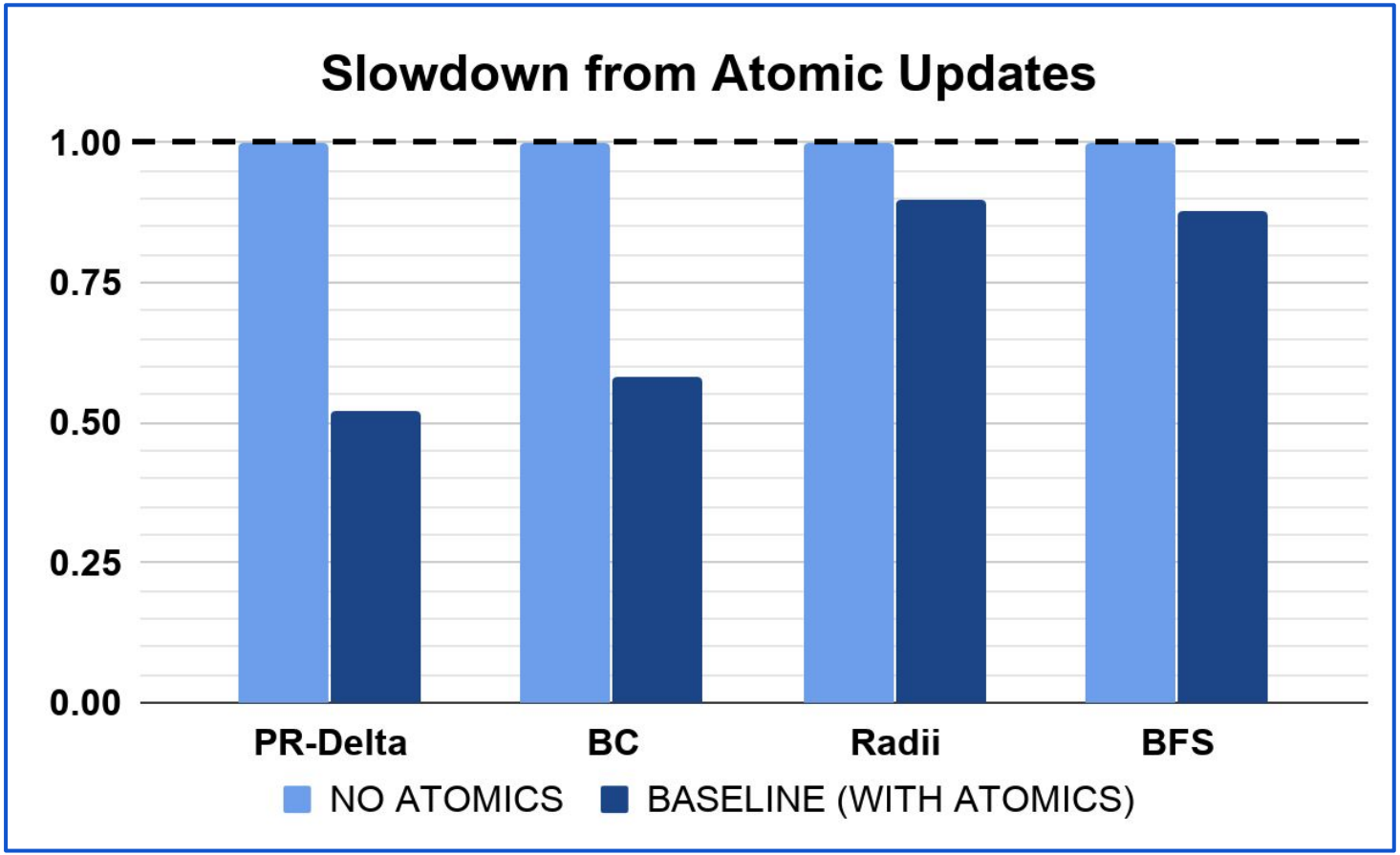


Vertex Centric Graph Processing (Push)

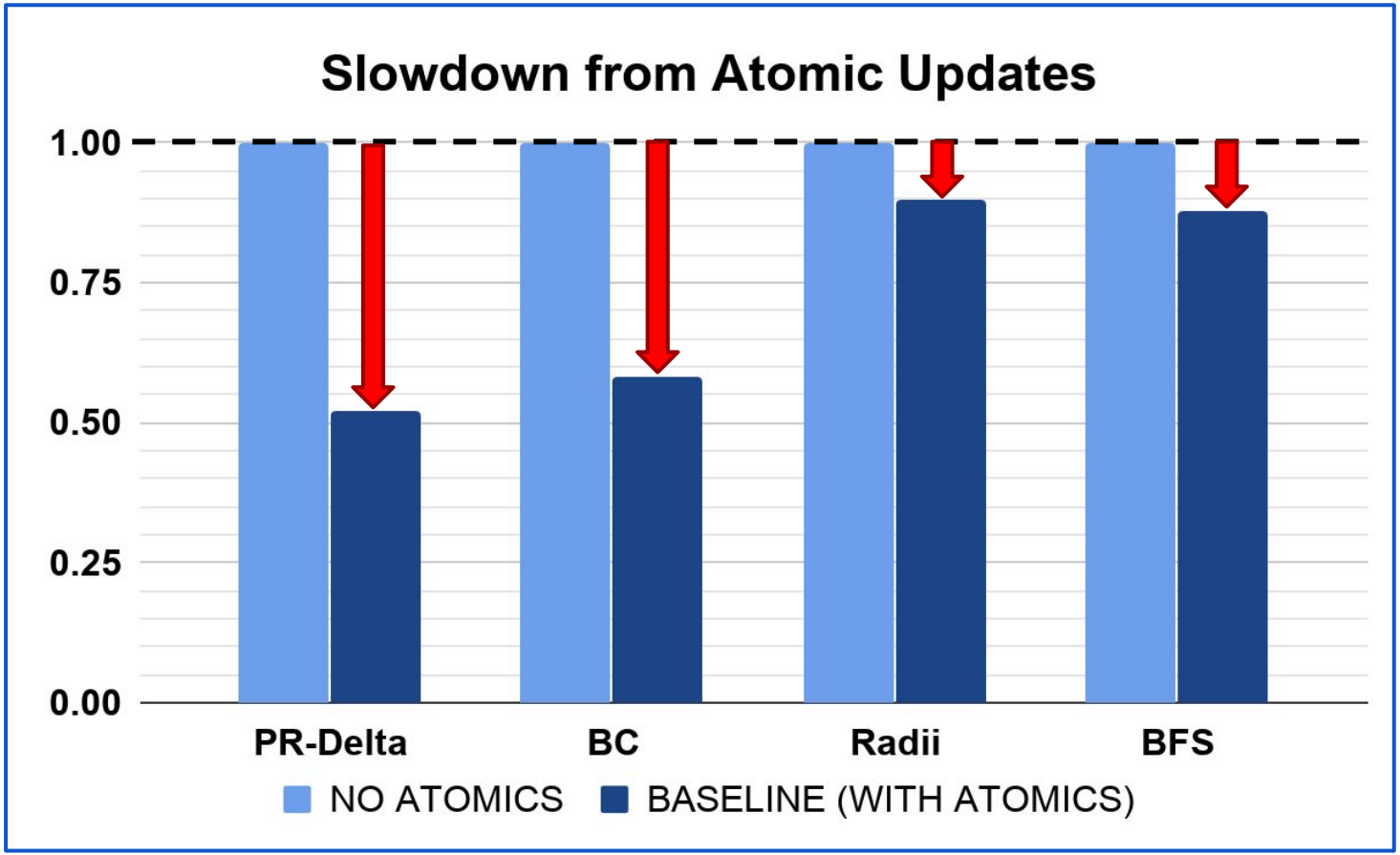
```
for src in Frontier:
    for dst in out_neigh(src):
        vtxData[dst] += auxData[src]
```

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

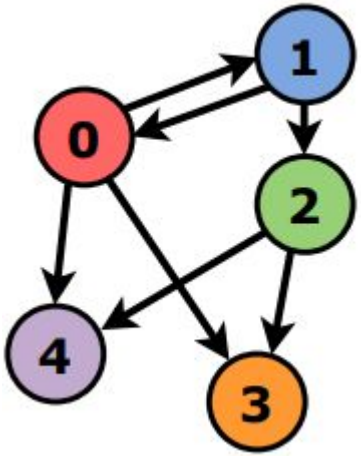
Bottleneck #1: Atomic Updates Hurt Performance



Bottleneck #1: Atomic Updates Hurt Performance

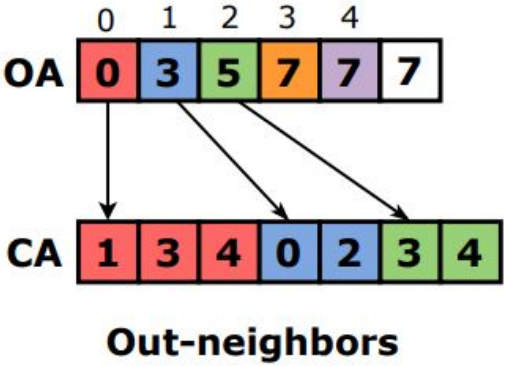


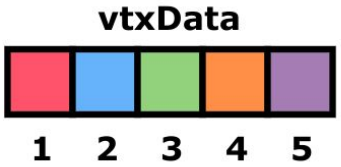
Source of Poor Locality ⇒ Irregular Memory Accesses



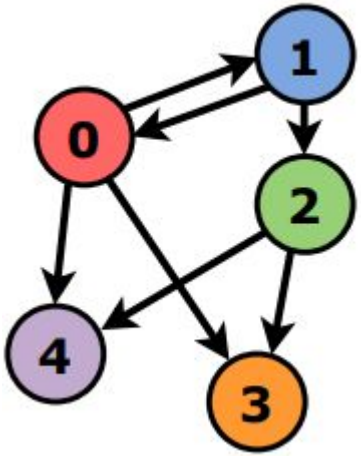
Vertex Centric Graph Processing (Push)

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



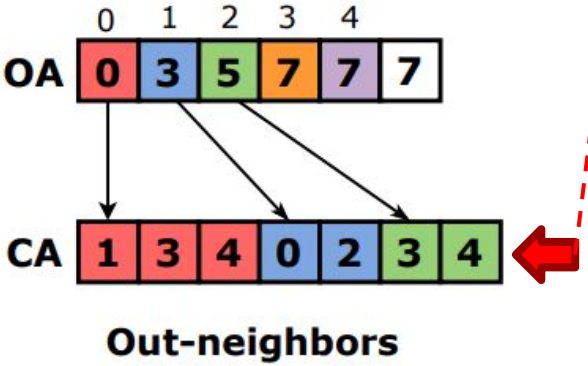


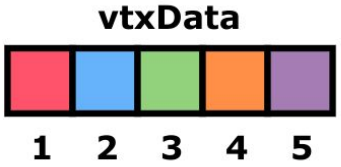
Source of Poor Locality ⇒ Irregular Memory Accesses



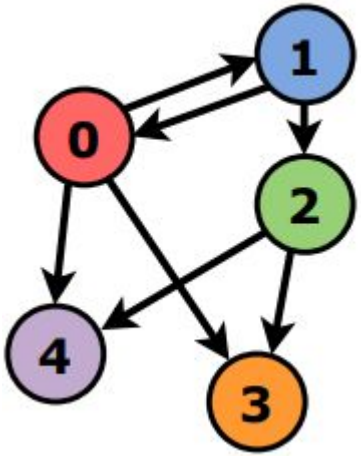
Vertex Centric Graph Processing (Push)

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





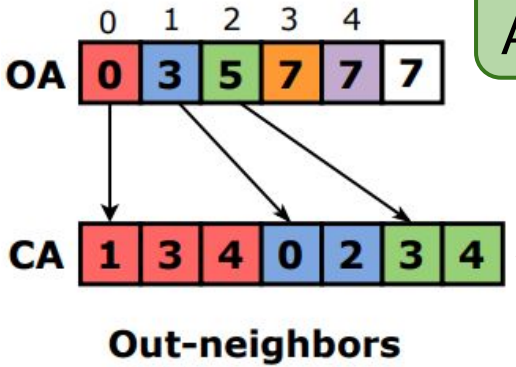
Source of Poor Locality ⇒ Irregular Memory Accesses

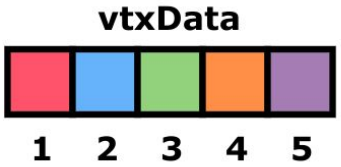


Vertex Centric Graph Processing (Push)

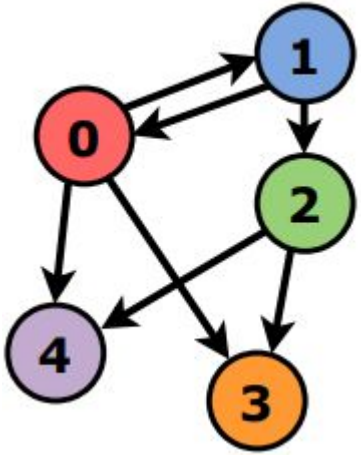
```
for src in Frontier:
    for dst in out_neigh(src):
        vtxData[dst] += auxData[src]
```

Irregular Memory Accesses





Source of Poor Locality ⇒ Irregular Memory Accesses



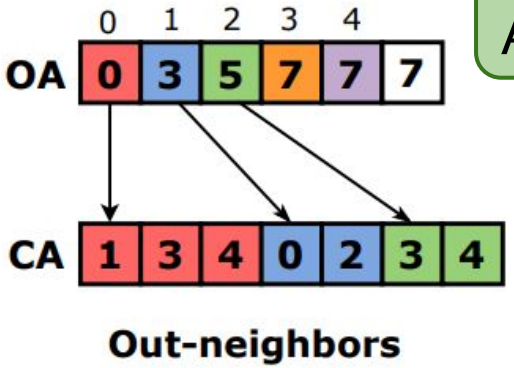
Vertex Centric Graph Processing (Push)

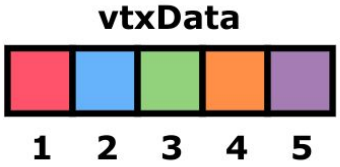
```
for src in Frontier:
    for dst in out_neigh(src):
        vtxData[dst] += auxData[src]
```

Irregular Memory Accesses

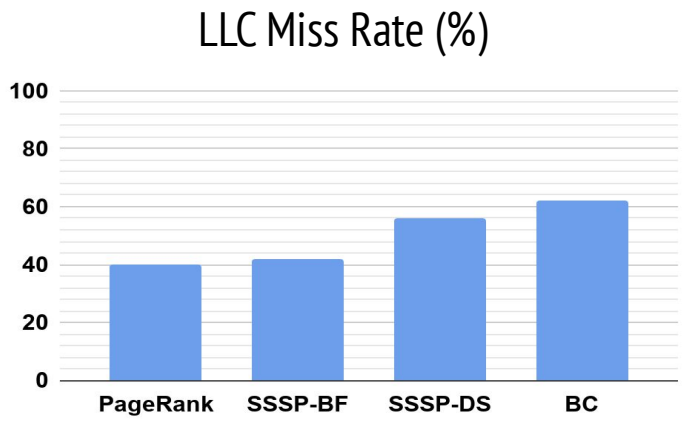
Size of vtxData ~ **200MB**

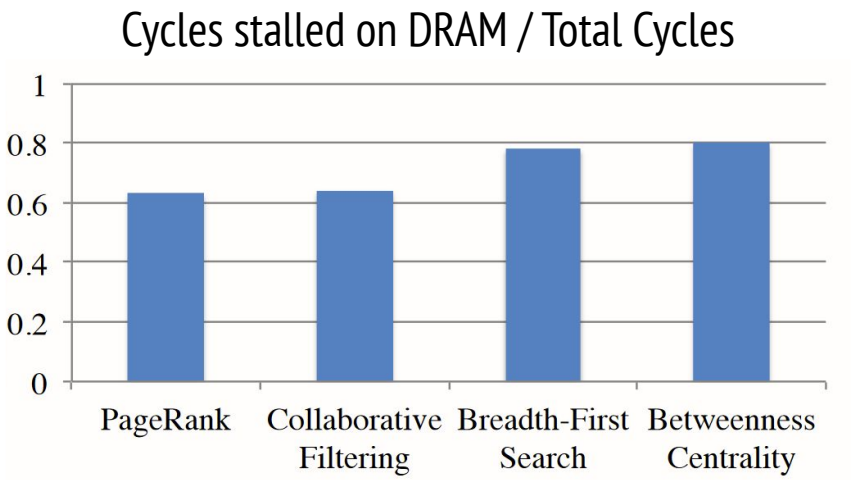
Size of Typical LLCs ~ **32MB**





Bottleneck #2: Graph Applications are DRAM-latency bound





Problem: Poor LLC locality ⇒ Many long-latency DRAM accesses

Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

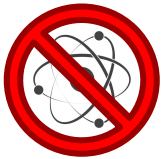
Outline

- **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

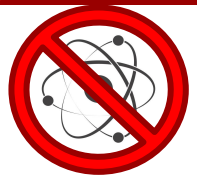


- **Data Duplication** : Improves Scalability
- **Graph Reordering** : Reduces DRAM accesses

BUT

- **Duplication** incurs Hub Identification Overhead
- **Reordering** incurs False Sharing Overhead

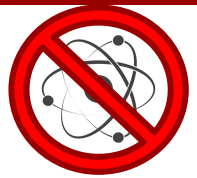
Optimization #1: Data Duplication



```
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]
```

vtxData

vtxData

Tid 0

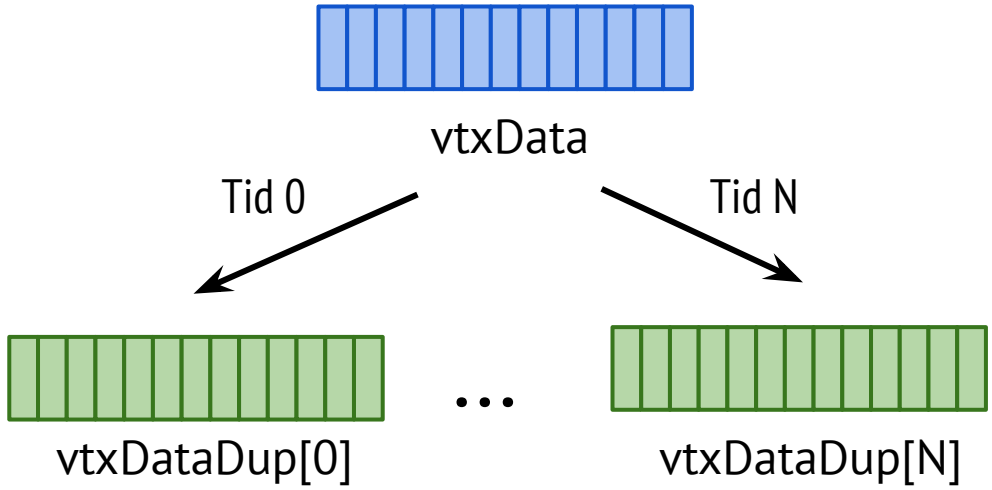
Tid N

vtxDataDup[0]

...

vtxDataDup[N]

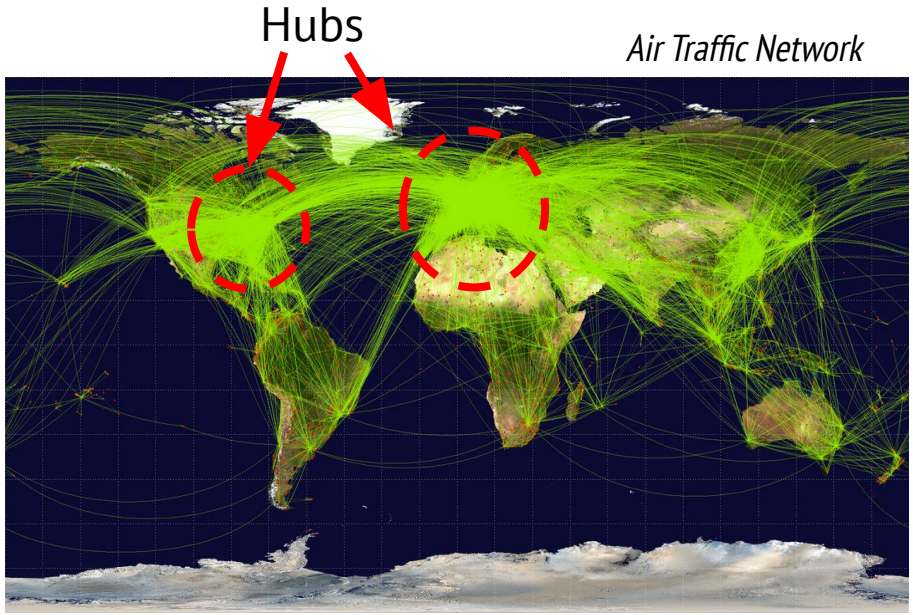
Naive Duplication Imposes High Memory Overhead



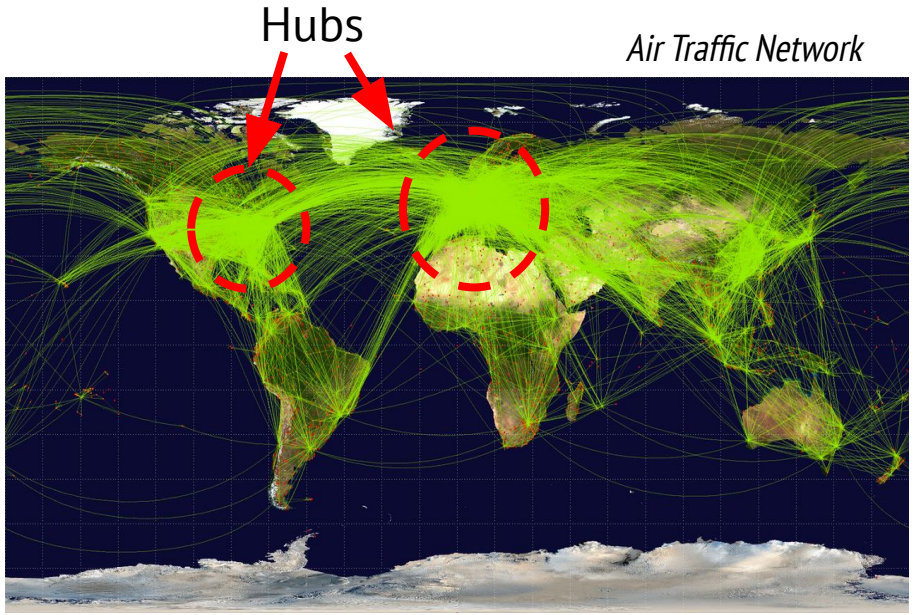
For a graph with 64M vertices, 4B / vtx, 16 threads

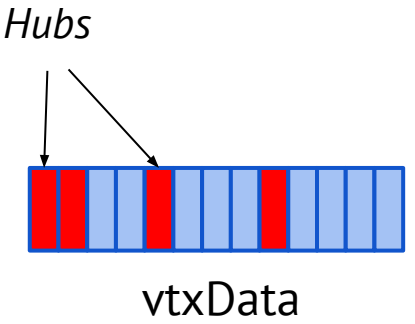
Memory footprint after Duplication = **4GB!**

Power-Law Graphs Allow Memory-Efficient Duplication



Power-Law Graphs Allow Memory-Efficient Duplication

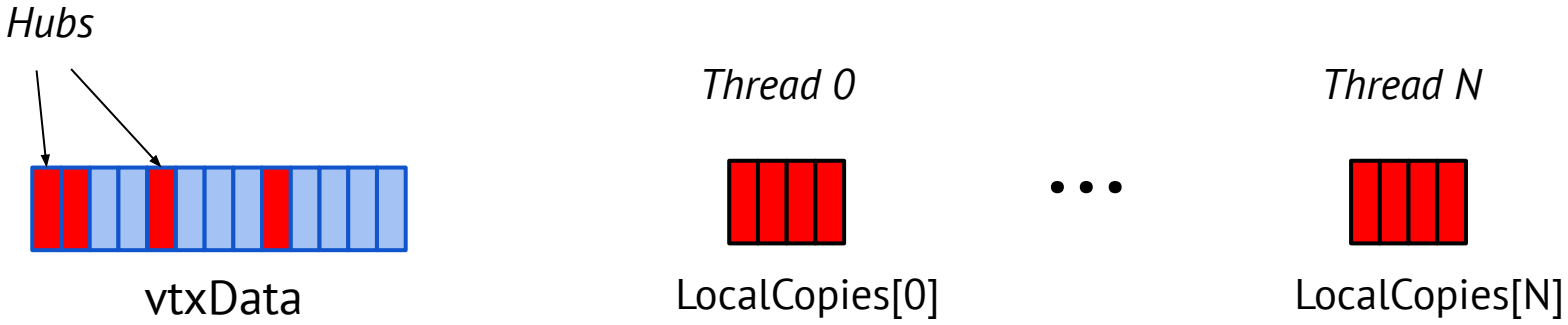




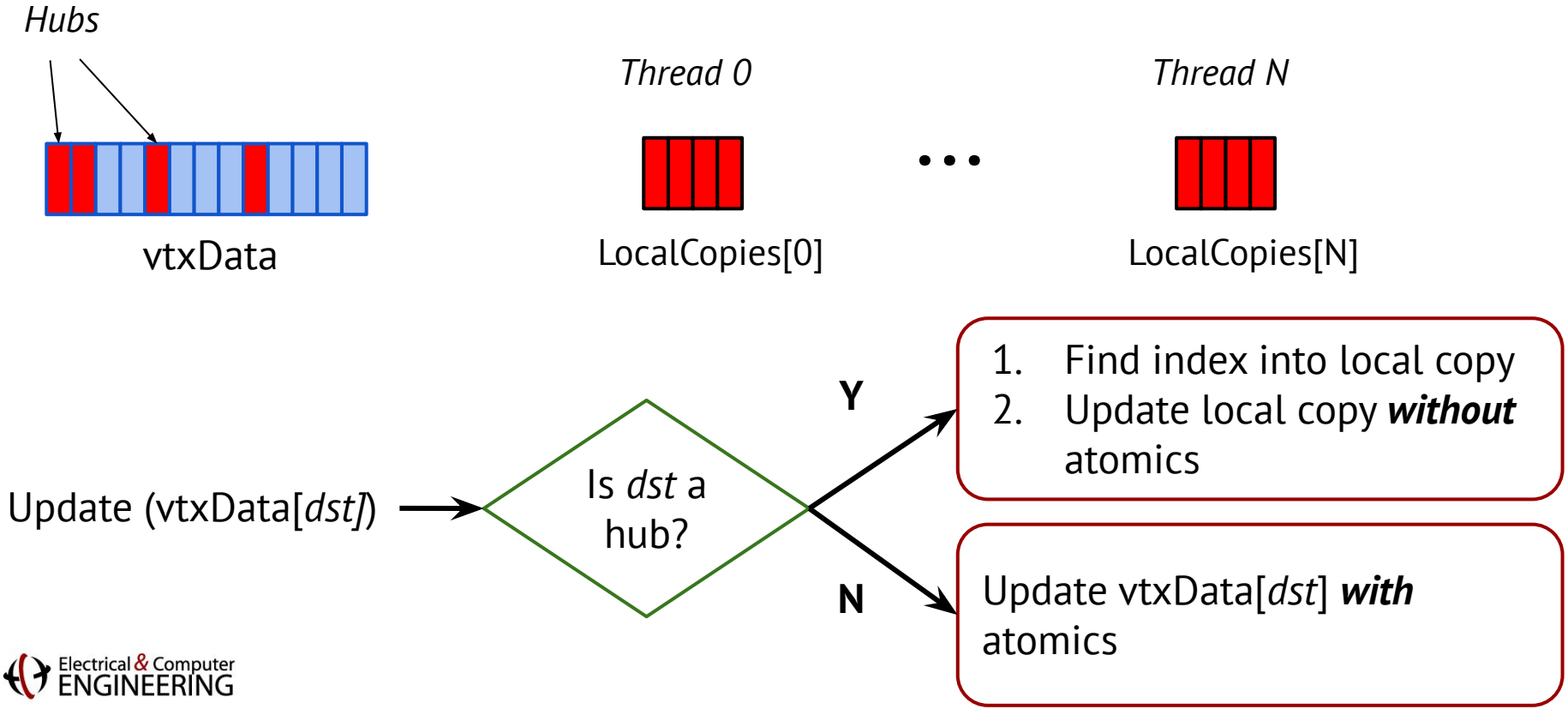
Majority of Atomic Updates are to Hub Vertices

Duplicate *only* the Hub Vertex Data

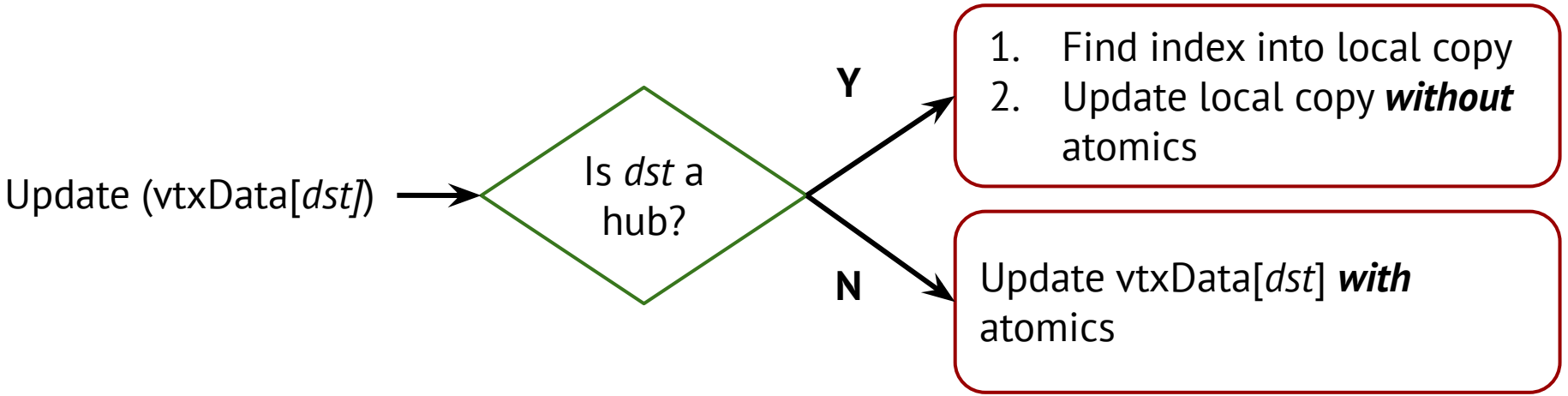
HUBDUP: Duplication for Power-Law Graphs



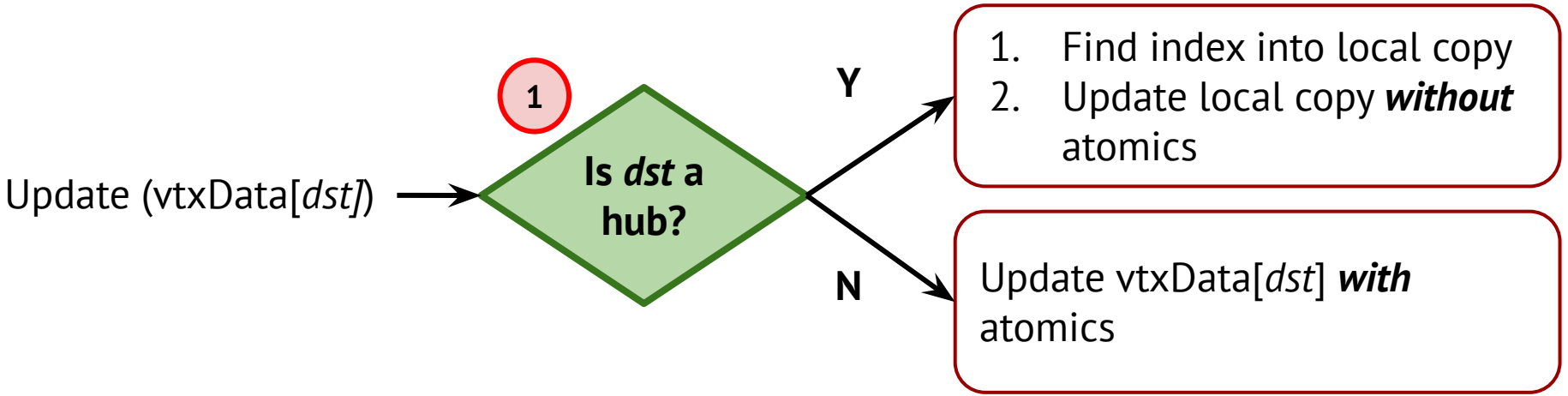
HUBDUP: Duplication for Power-Law Graphs



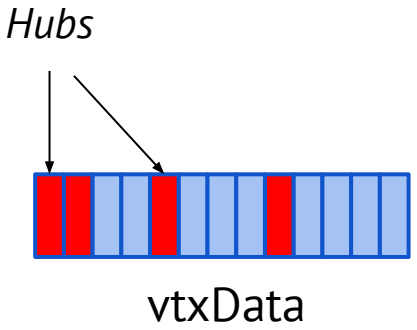
HUBDUP Incurs Overheads



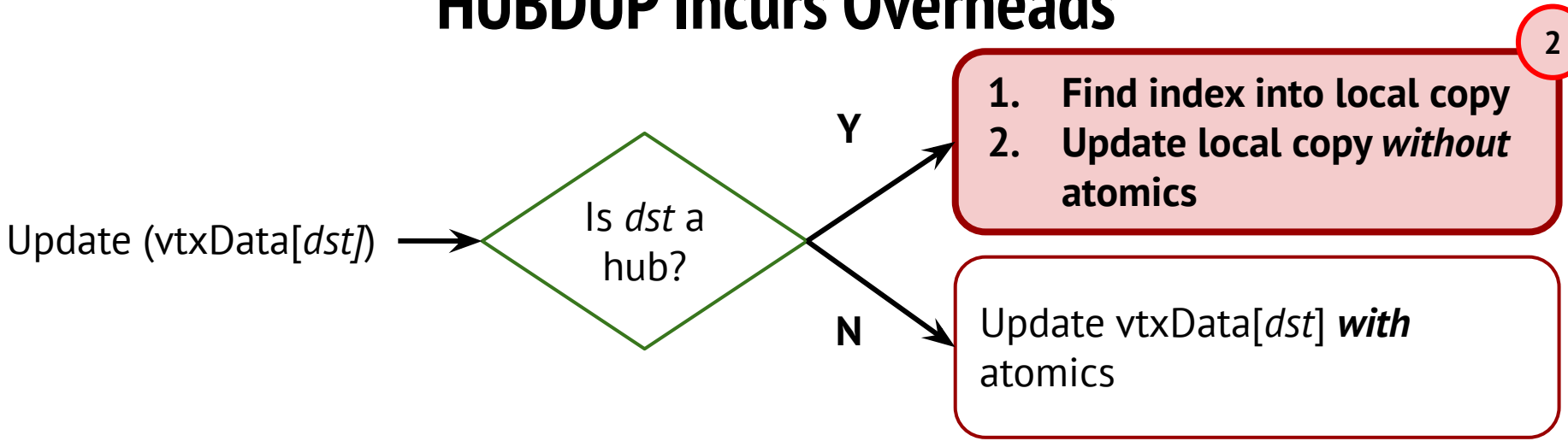
HUBDUP Incurs Overheads



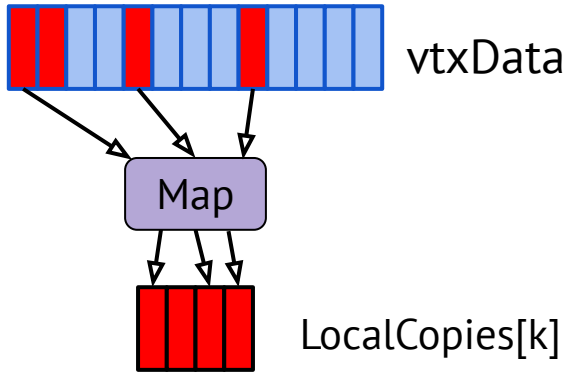
Overhead #1: *Identifying Hub Vertices*

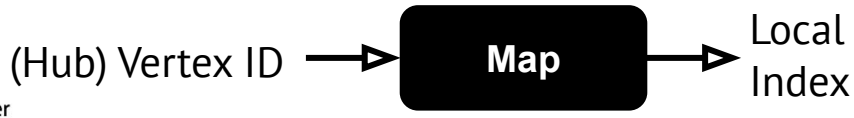


HUBDUP Incurs Overheads



Overhead #2: *Indexing into thread-local copies*

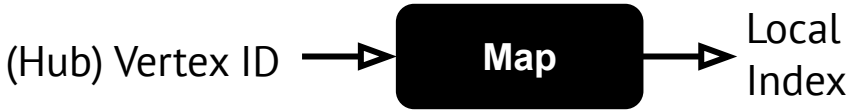




Summary Of Overheads In HUBDUP

Overhead #1:
Hub Identification

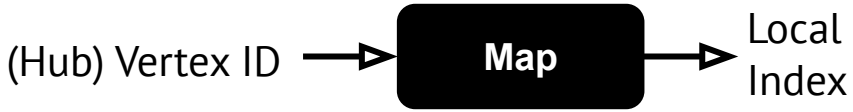
Overhead #2:
Local-Copy Indexing



Summary Of Overheads In HUBDUP

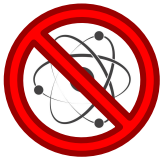
Overhead #1:
Hub Identification

Overhead #2:
Local-Copy Indexing



Problem: Hubs can have *arbitrary* vertex IDs

Optimizations for eliminating performance bottlenecks

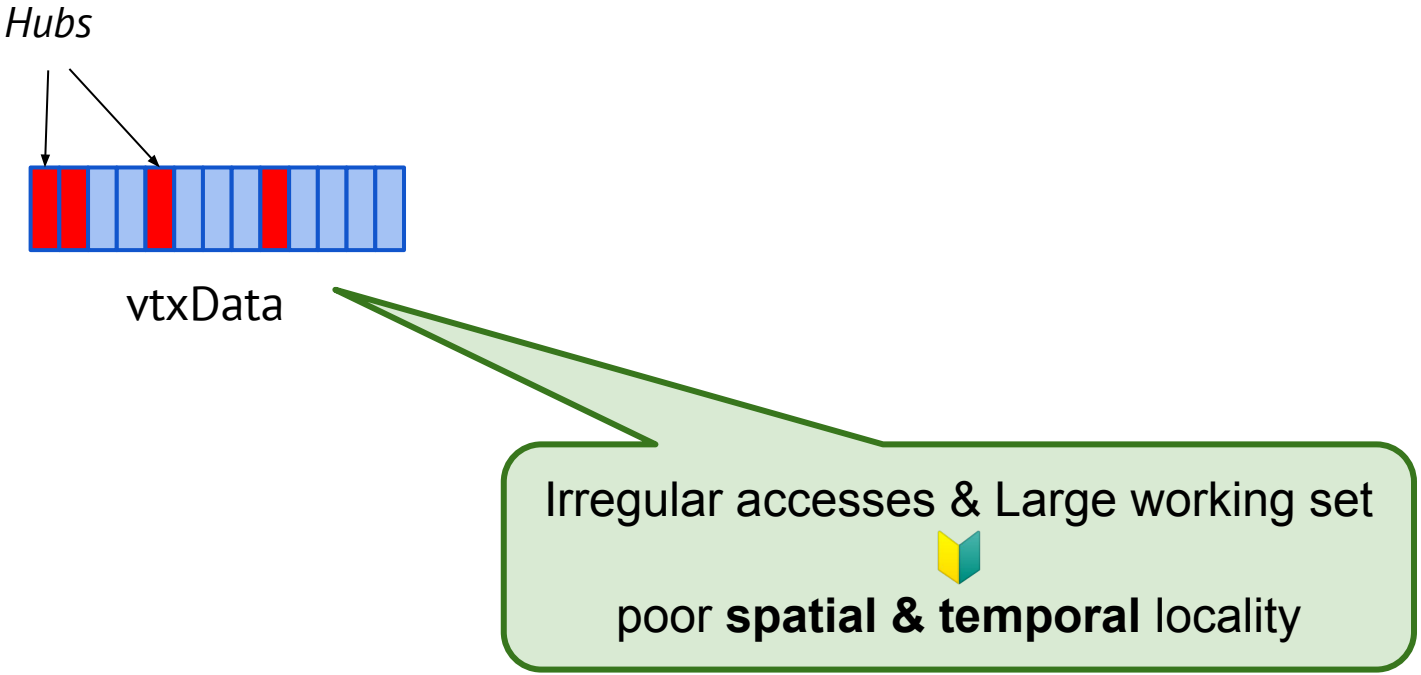


- **Data Duplication** : Improves Scalability ✔
- **Graph Reordering** : Reduces DRAM accesses

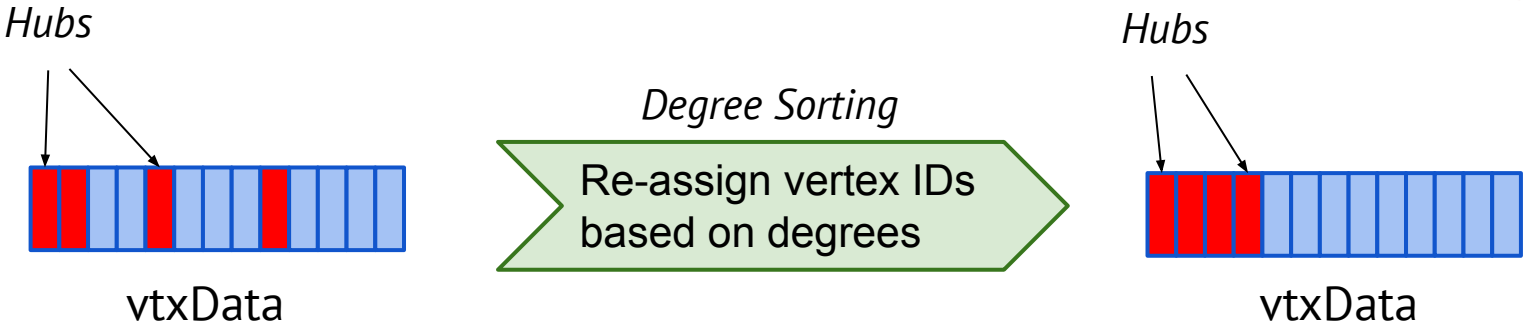
BUT

- **HUBDUP** incurs Hub Identification Overhead ✔
- **Reordering** incurs False Sharing Overhead

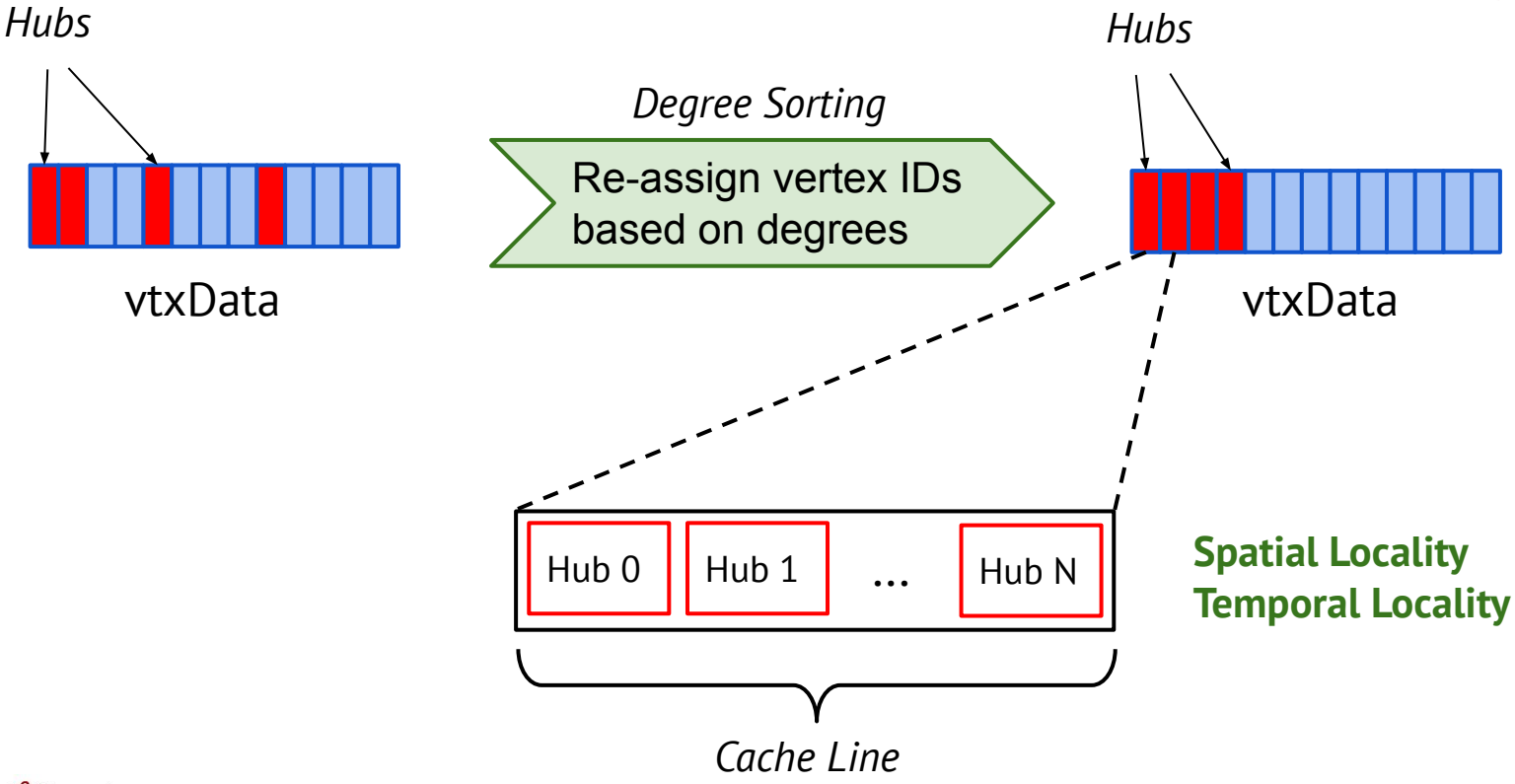
Optimization #2: Graph Reordering



Optimization #2: Graph Reordering

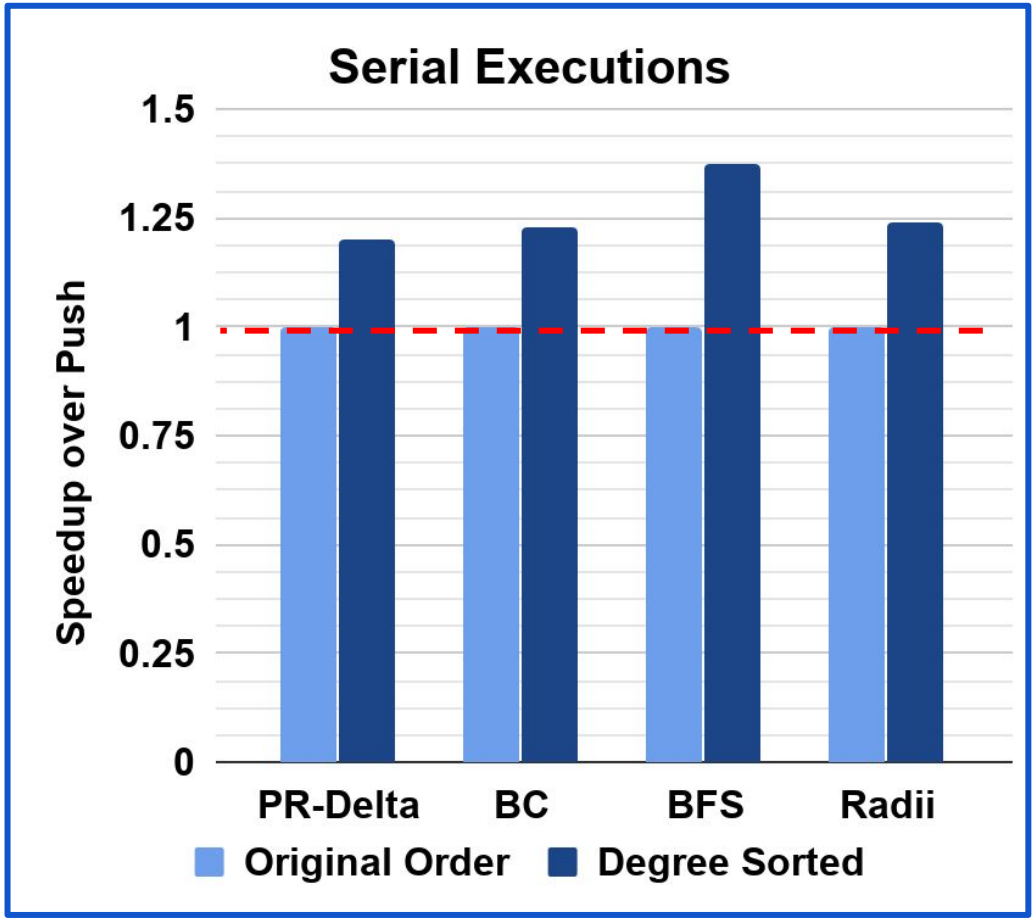


Optimization #2: Graph Reordering

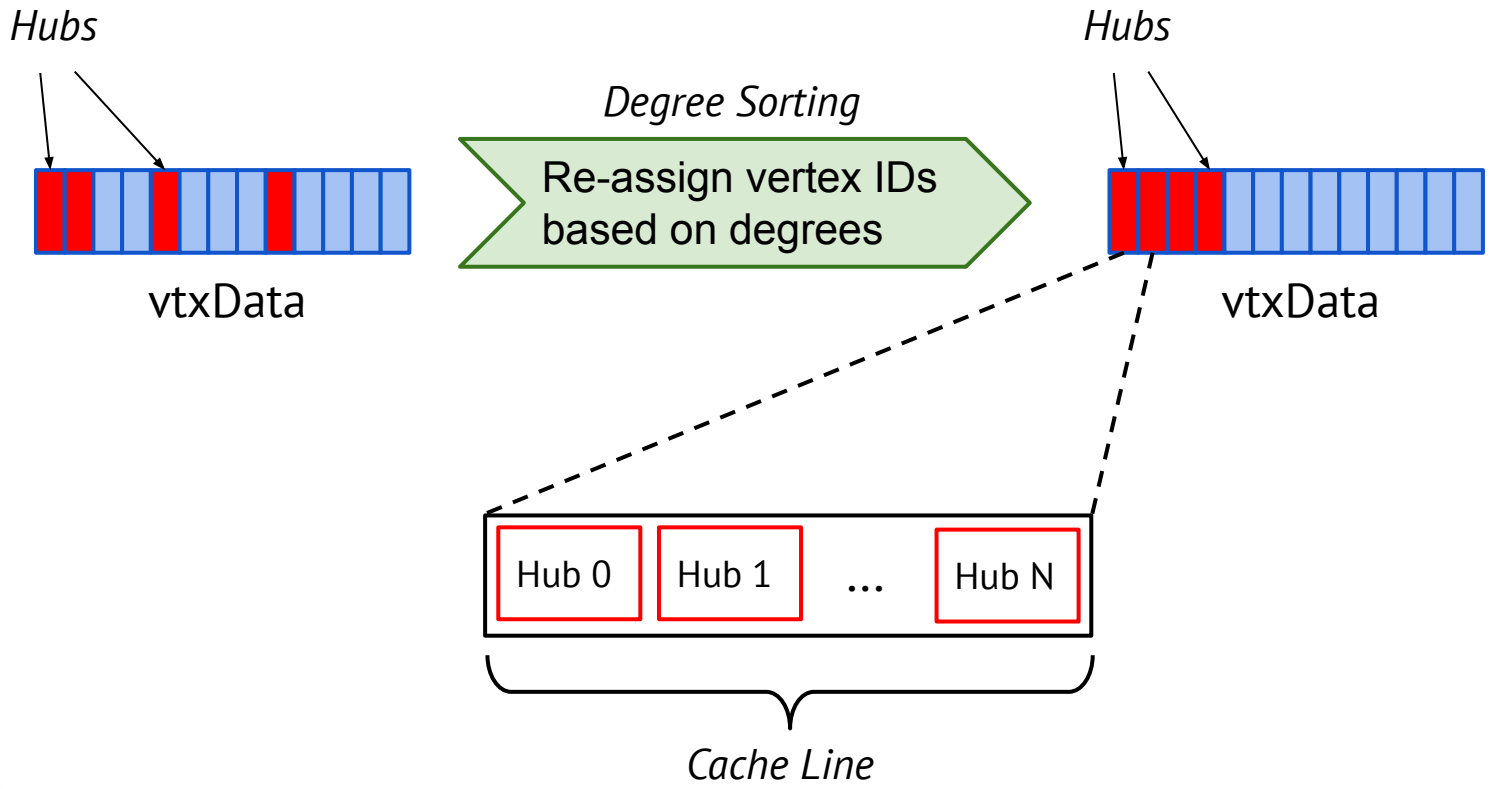


Spatial Locality
Temporal Locality

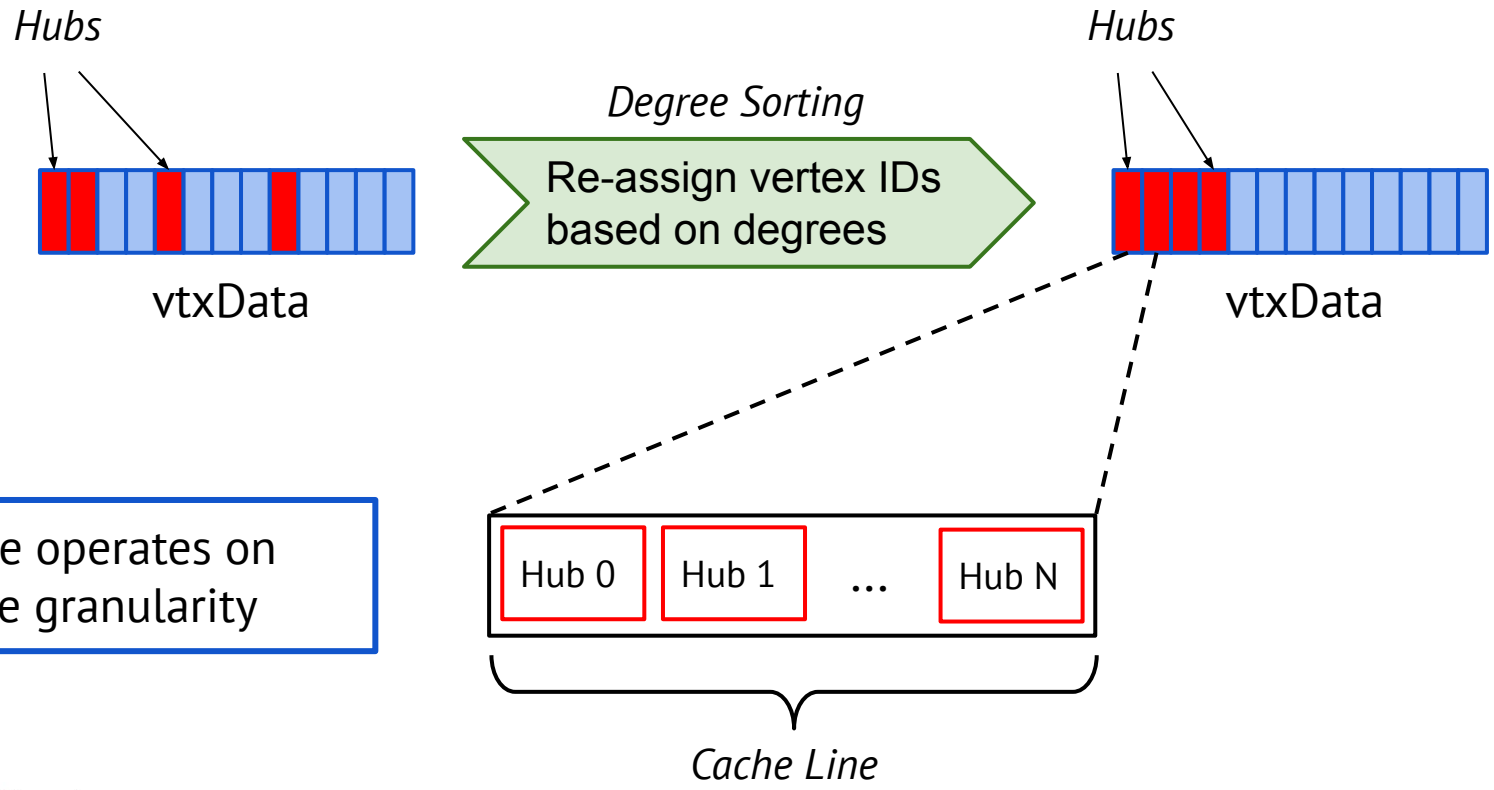
Degree Sorting Is Effective For Serial Graph Processing



Degree Sorting Introduces False Sharing

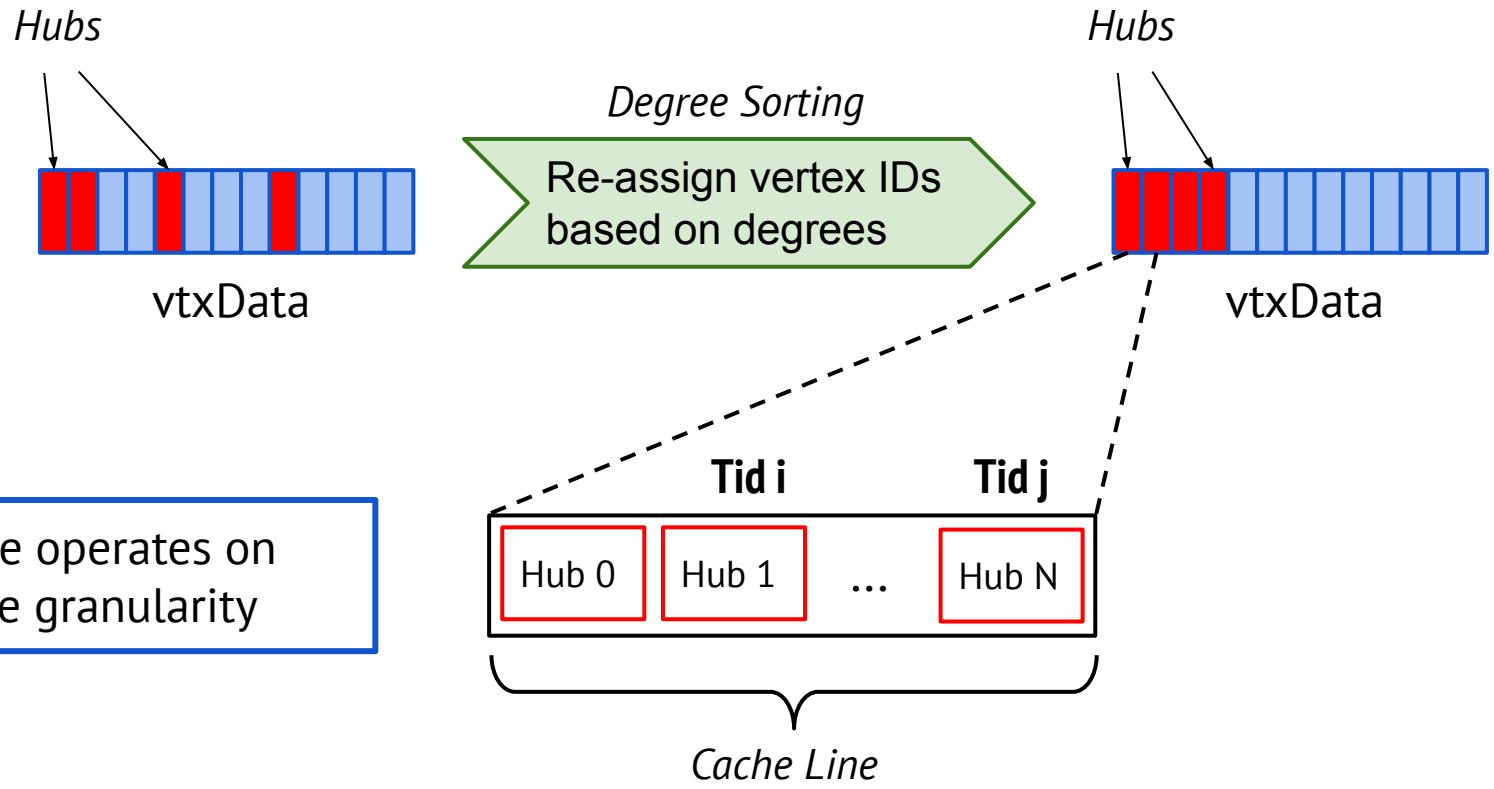


Degree Sorting Introduces False Sharing



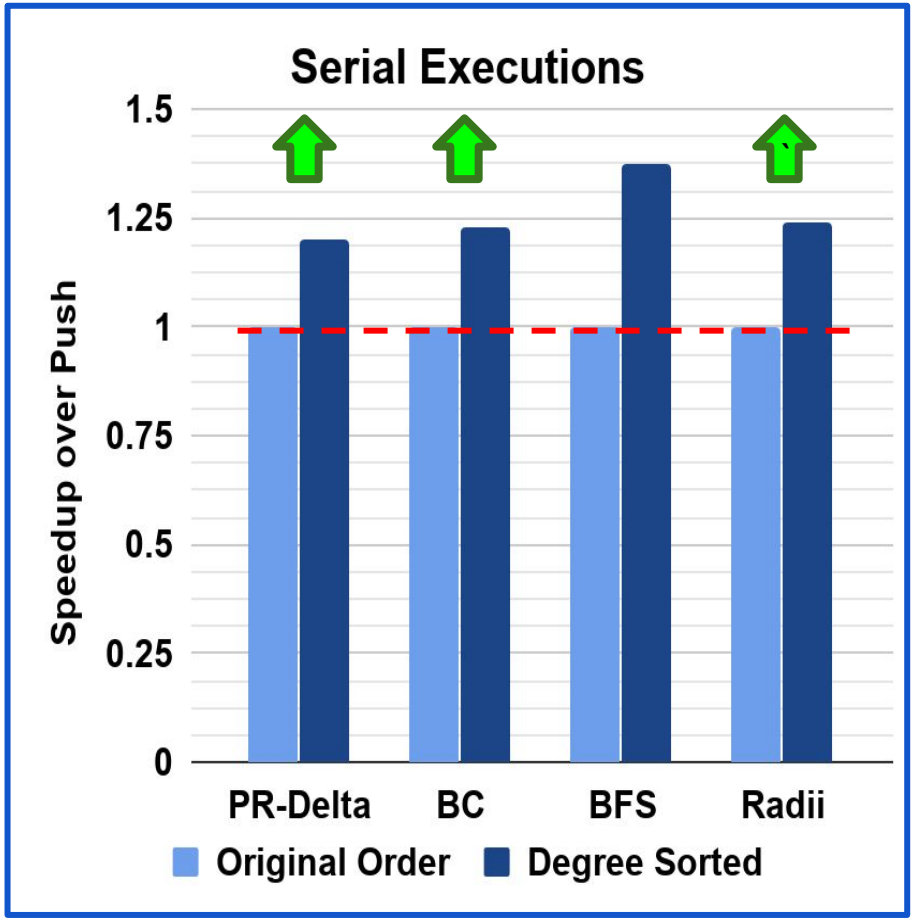
Coherence operates on cache-line granularity

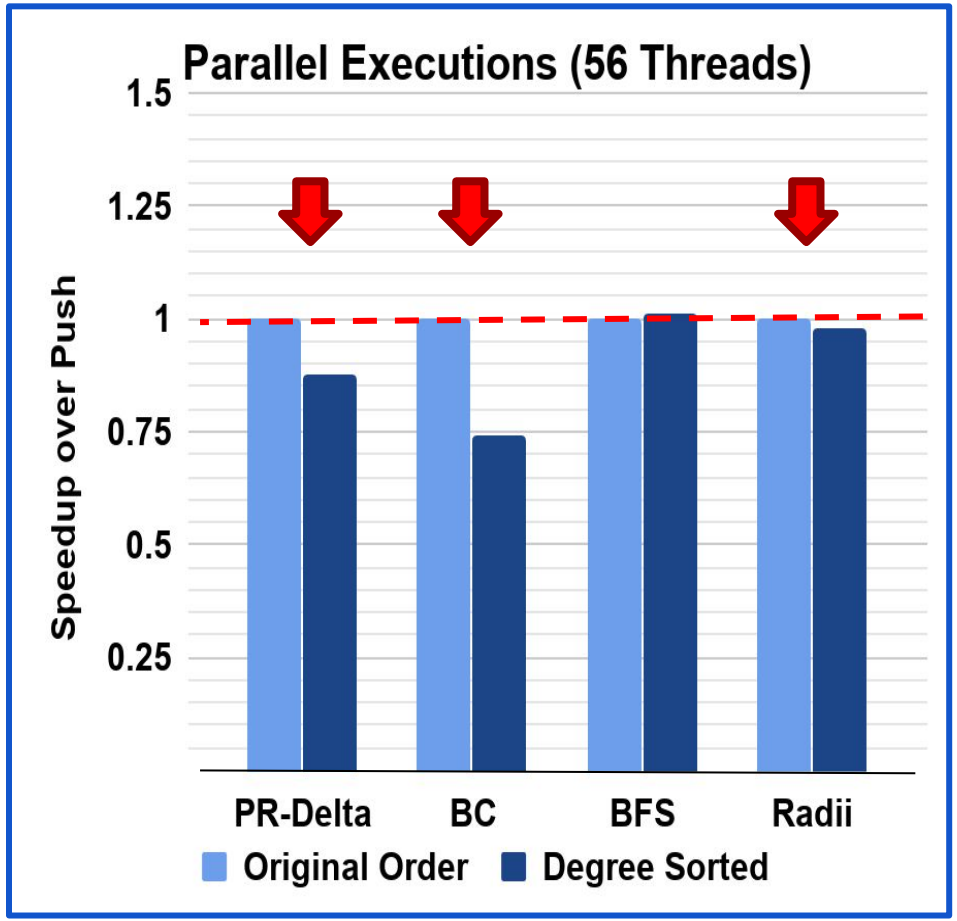
Degree Sorting Introduces False Sharing



Coherence operates on cache-line granularity

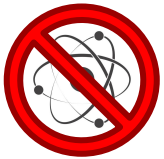
False Sharing Hurts Performance





Optimizations for eliminating performance bottlenecks

- **Data Duplication** : Improves Scalability ✔

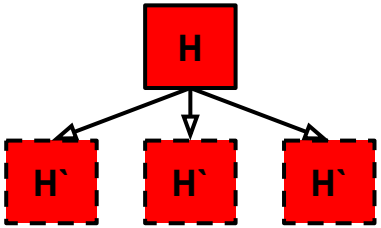


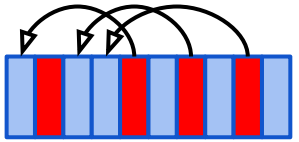
- **Graph Reordering** : Reduces DRAM accesses ✔

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

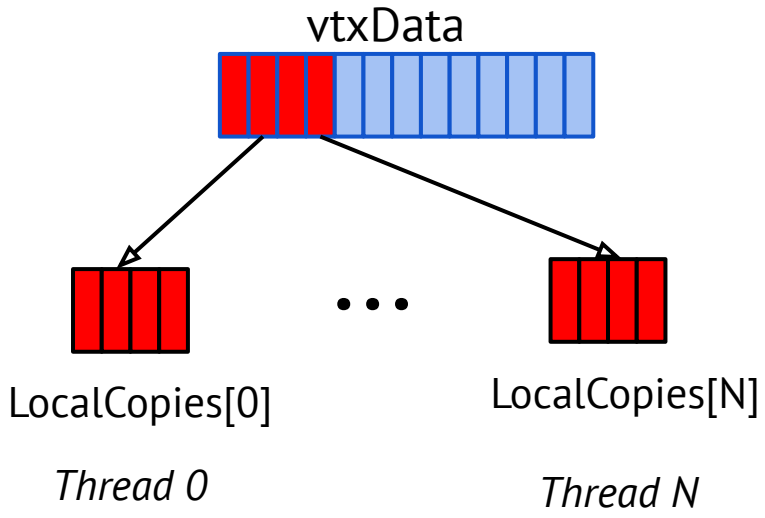
Outline

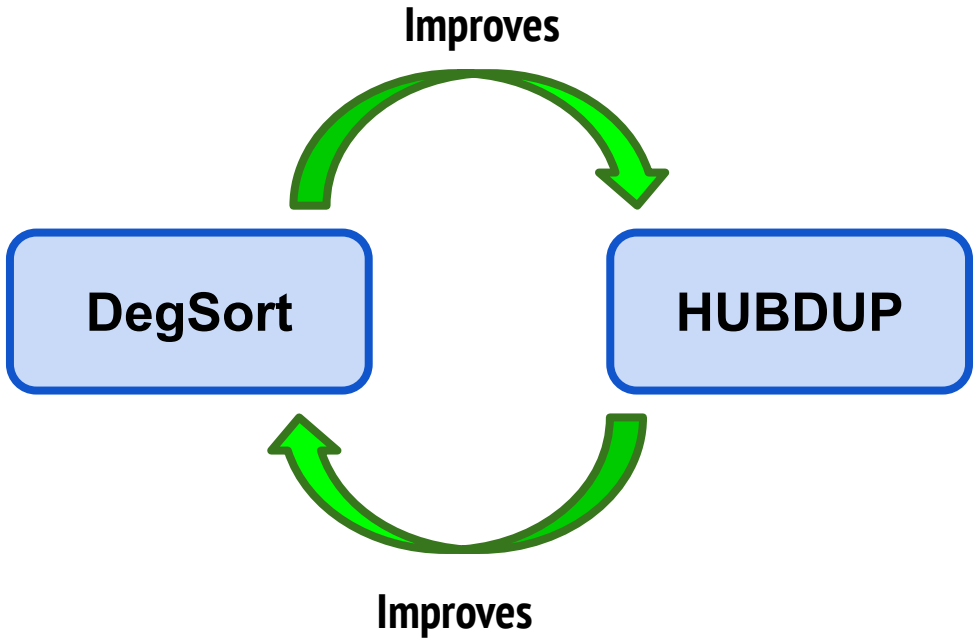
- **Single-node Graph Processing is Sub-optimal ✔**
- **Existing Optimizations have Overheads ✔**
  - **Duplication incurs Hub Identification Overhead**
  - **Reordering incurs False Sharing Overhead**
- RADAR: Combining the Benefits of Duplication and Reordering
- Advantages of RADAR over Push-Pull 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

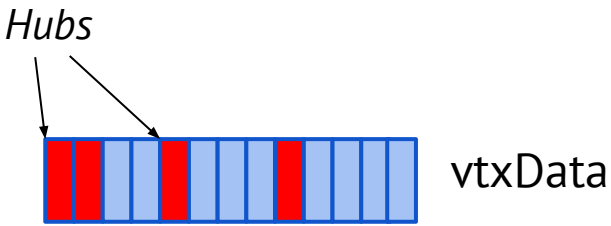




Degree Sorting Improves HUBDUP

Ovhd #1:
Hub Identification

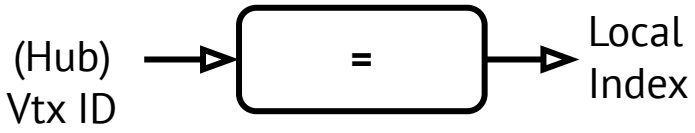
Ovhd #2:
Local-Copy Indexing

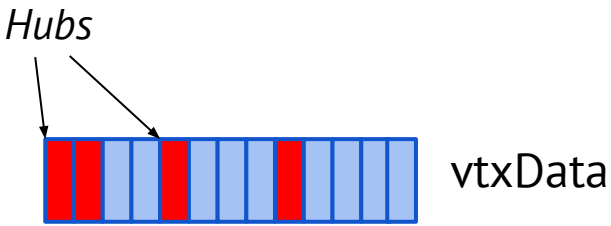


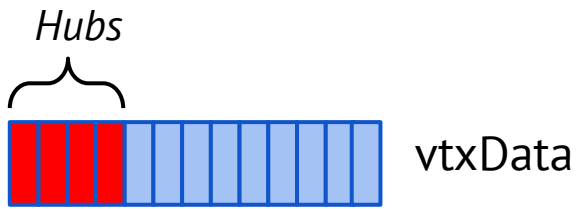
Degree Sorting Improves HUBDUP

Ovhd #1:
Hub Identification

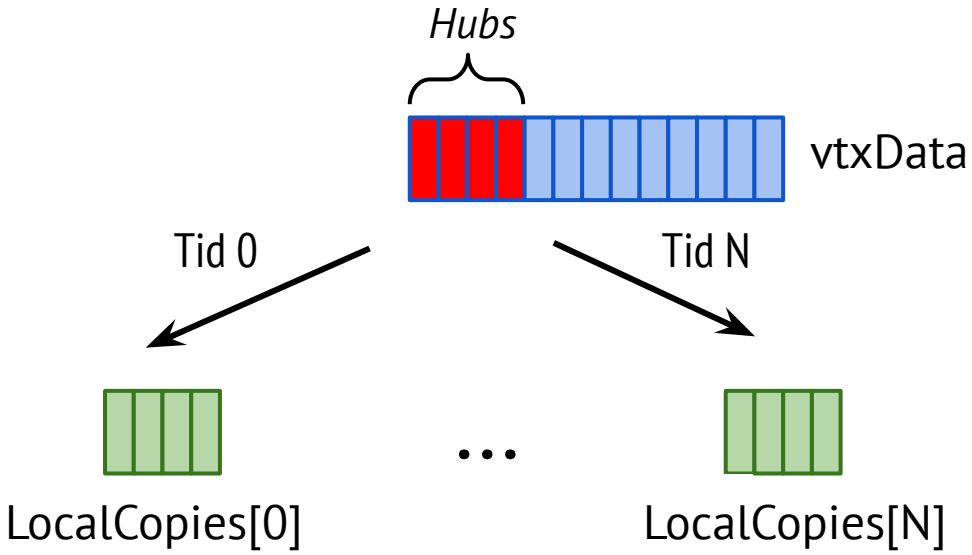
Ovhd #2:
Local-Copy Indexing



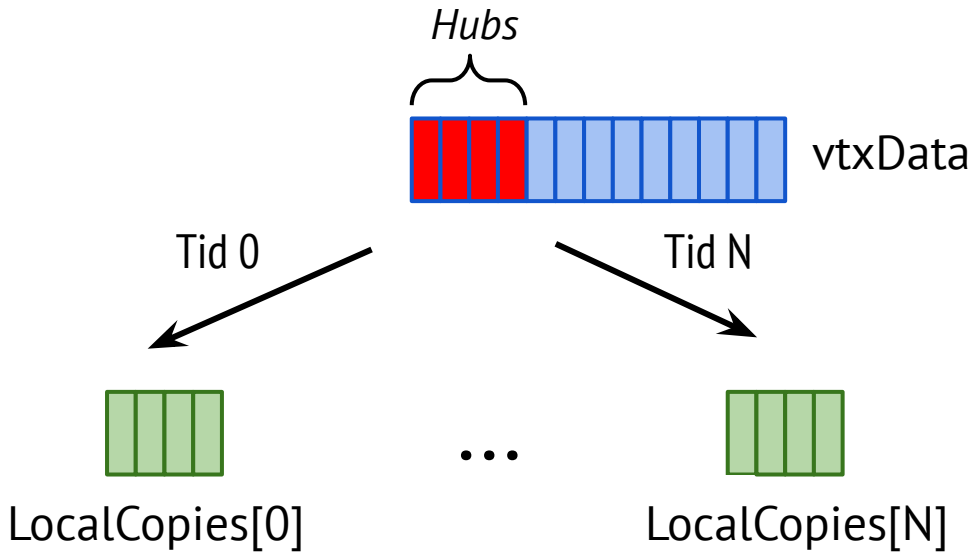




HUBDUP Improves Degree Sorting

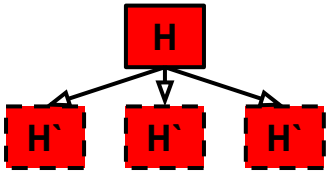


HUBDUP Improves Degree Sorting



Threads update a private copies of hubs ⇒ No False Sharing

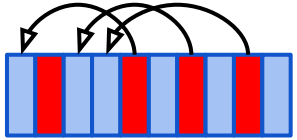
Reordering Assisted Duplication/Duplication Assisted Reordering



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)

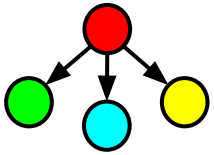
Improves Cache Locality (without false-sharing for hubs)

Outline

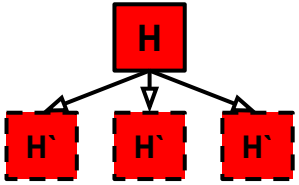
- Single-node Graph Processing is Sub-optimal ✅
- Existing Optimizations have Overheads ✅
- RADAR: Combining the Benefits of Duplication and Reordering
  - HUBDUP & Degree Sorting are Mutually Enabling
  - RADAR = HUBDUP + Degree Sorting
  - RADAR outperforms HUBDUP & Degree Sorting ⇦
- Advantages of RADAR over Push-Pull Direction Switching

Evaluation Space

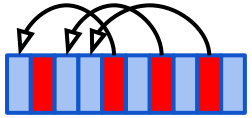
Systems:



Push (Baseline)



+ HUBDUP

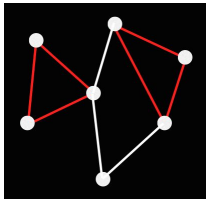


+ Degree Sorting

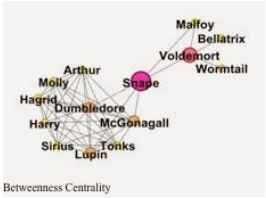
+ RADAR

Applications:

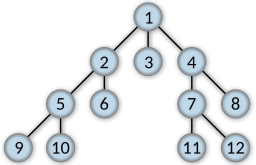
PageRank &
PageRank-Delta



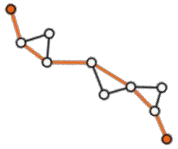
Triangle Listing



Betweenness
Centrality



BFS



Graph
Radius

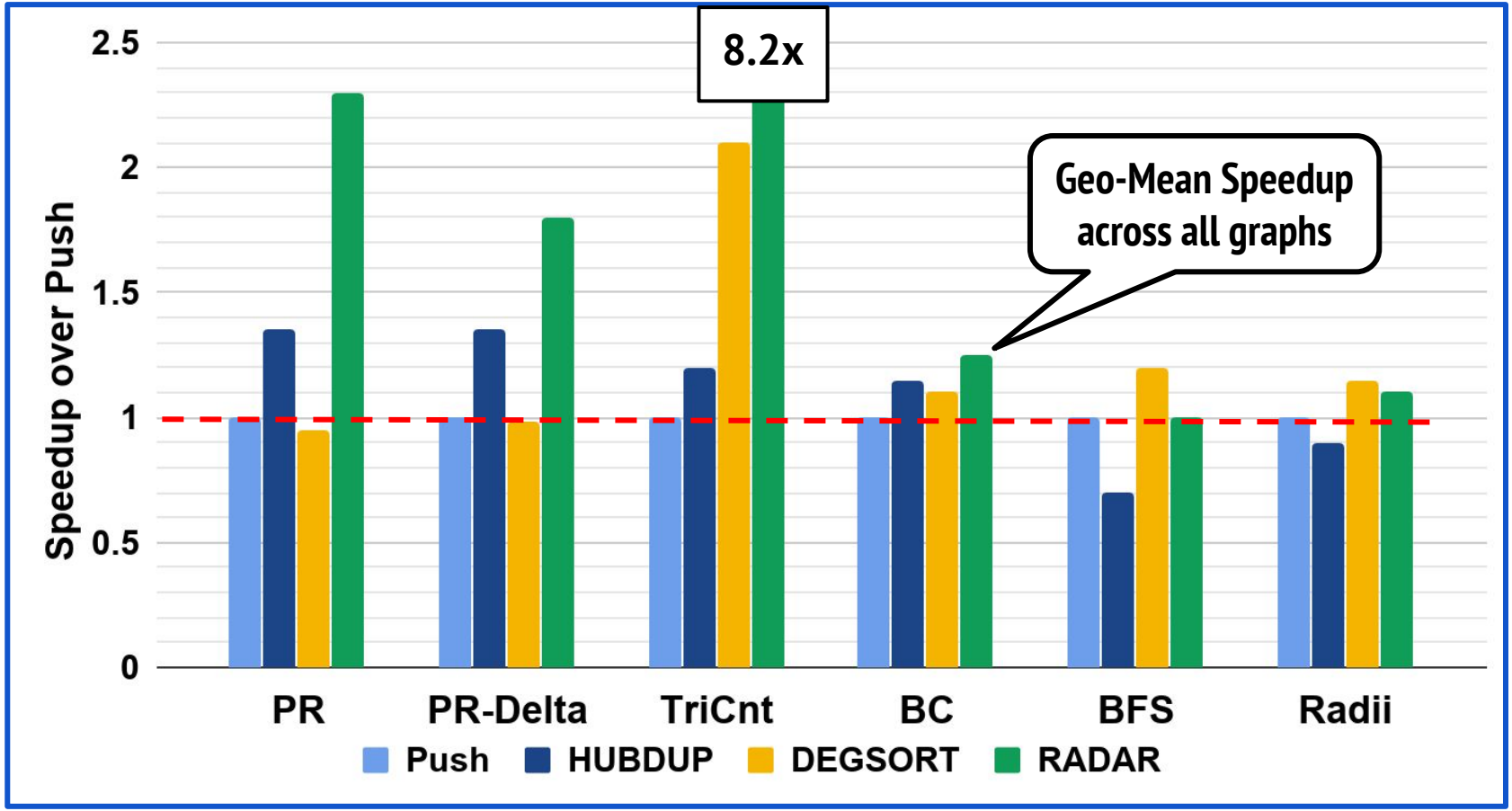
Input Graphs:

	DBP	GPL	PLD	KRON	TWIT	MPI	WEB	SD1
$\|V\|$ (in M)	18.27	28.94	42.89	33.55	61.58	52.58	50.64	94.95
$\|E\|$ (in B)	0.172	0.462	0.623	1.047	1.468	1.963	1.93	1.937

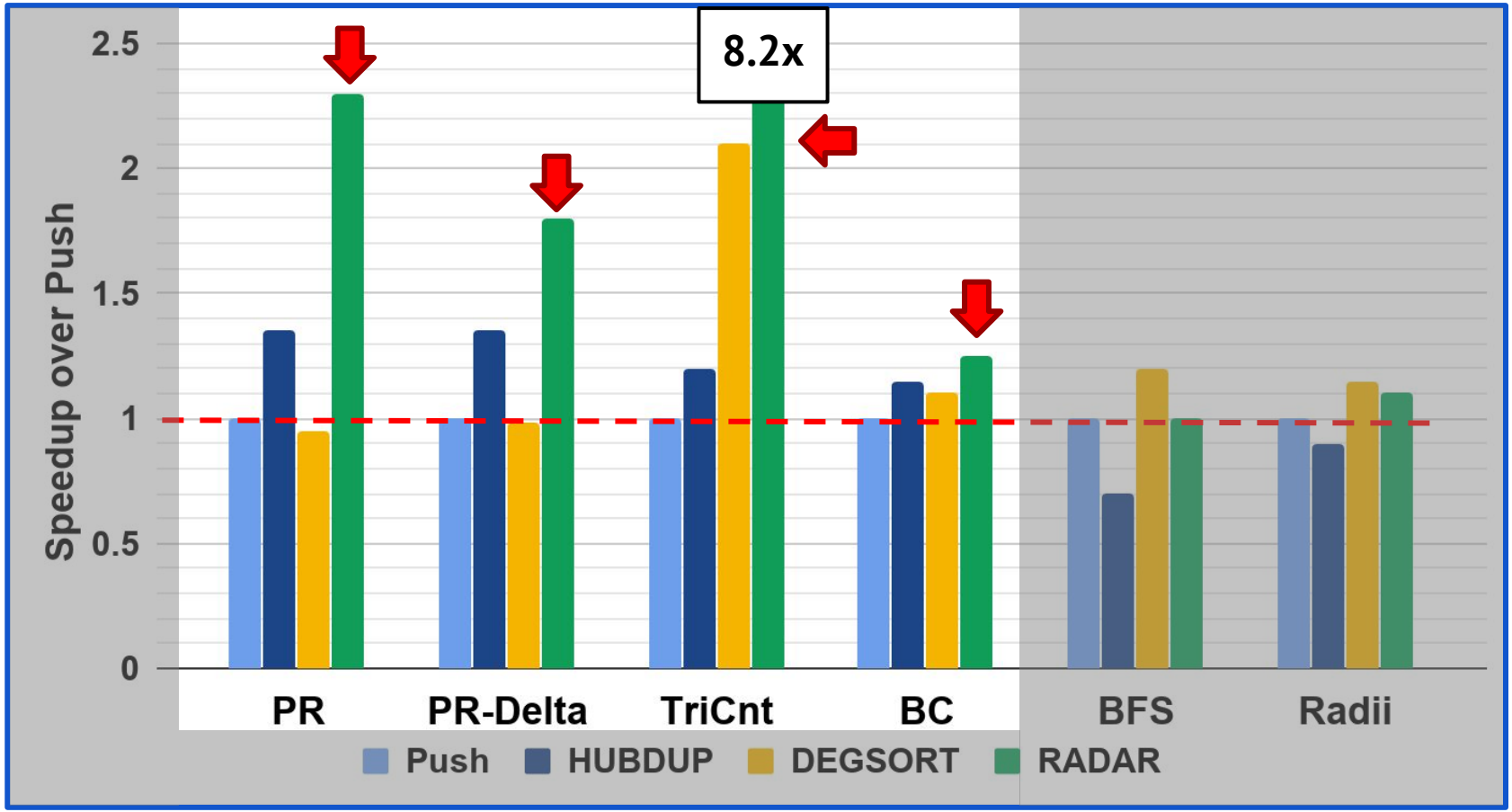
Evaluation Platform

- 28 Cores / 56 Threads across 2 socket ⇒ **All versions run with 56 threads**
- 35MB LLC per socket ⇒ **Hubs selected to fit LLC**
- 64GB DRAM

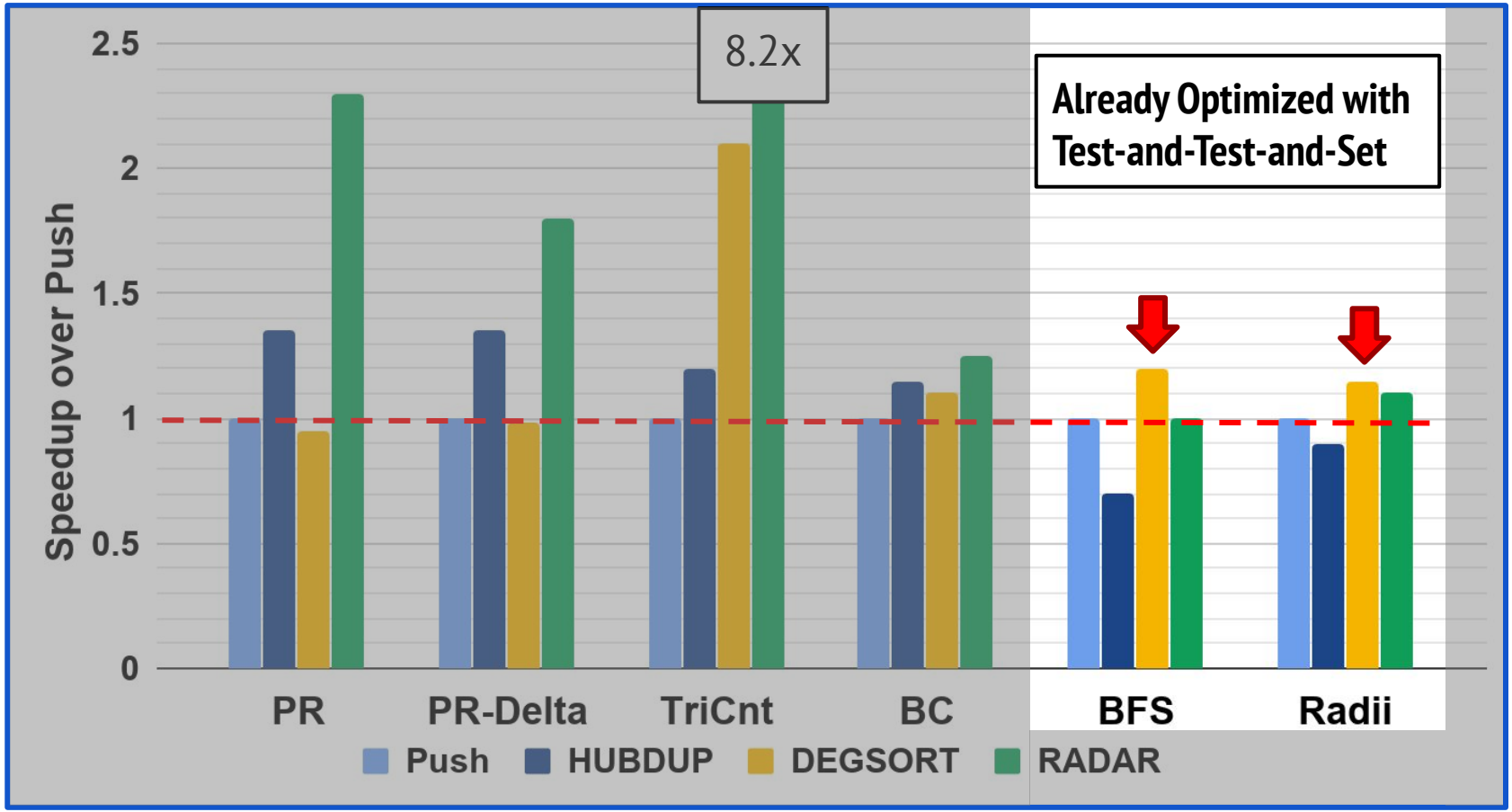
RADAR Outperforms Both HUBDUP And Degree Sorting



RADAR Outperforms Both HUBDUP And Degree Sorting



RADAR Outperforms Both HUBDUP And Degree Sorting



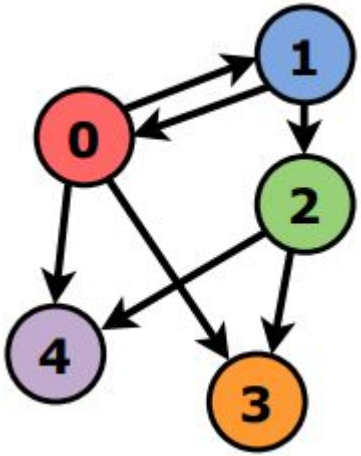
Outline

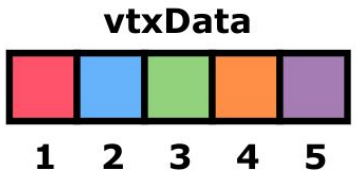
- Single-node Graph Processing is Sub-optimal ✔
- Existing Optimizations have Overheads ✔
- RADAR: Combining the Benefits of Duplication and Reordering ✔
  - HUBDUP & Degree Sorting are Mutually Enabling
  - RADAR = HUBDUP + Degree Sorting
  - RADAR outperforms HUBDUP & Degree Sorting
- Advantages of 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





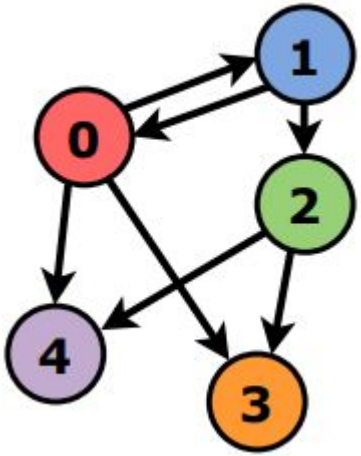
Push

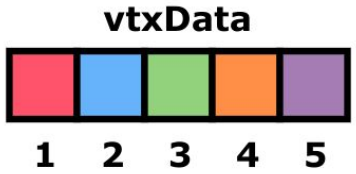
```
parallel_for src in Frontier:
  for dst in out_neigh(src):
    atomic {vtxData[dst] += auxData[src]}
```

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

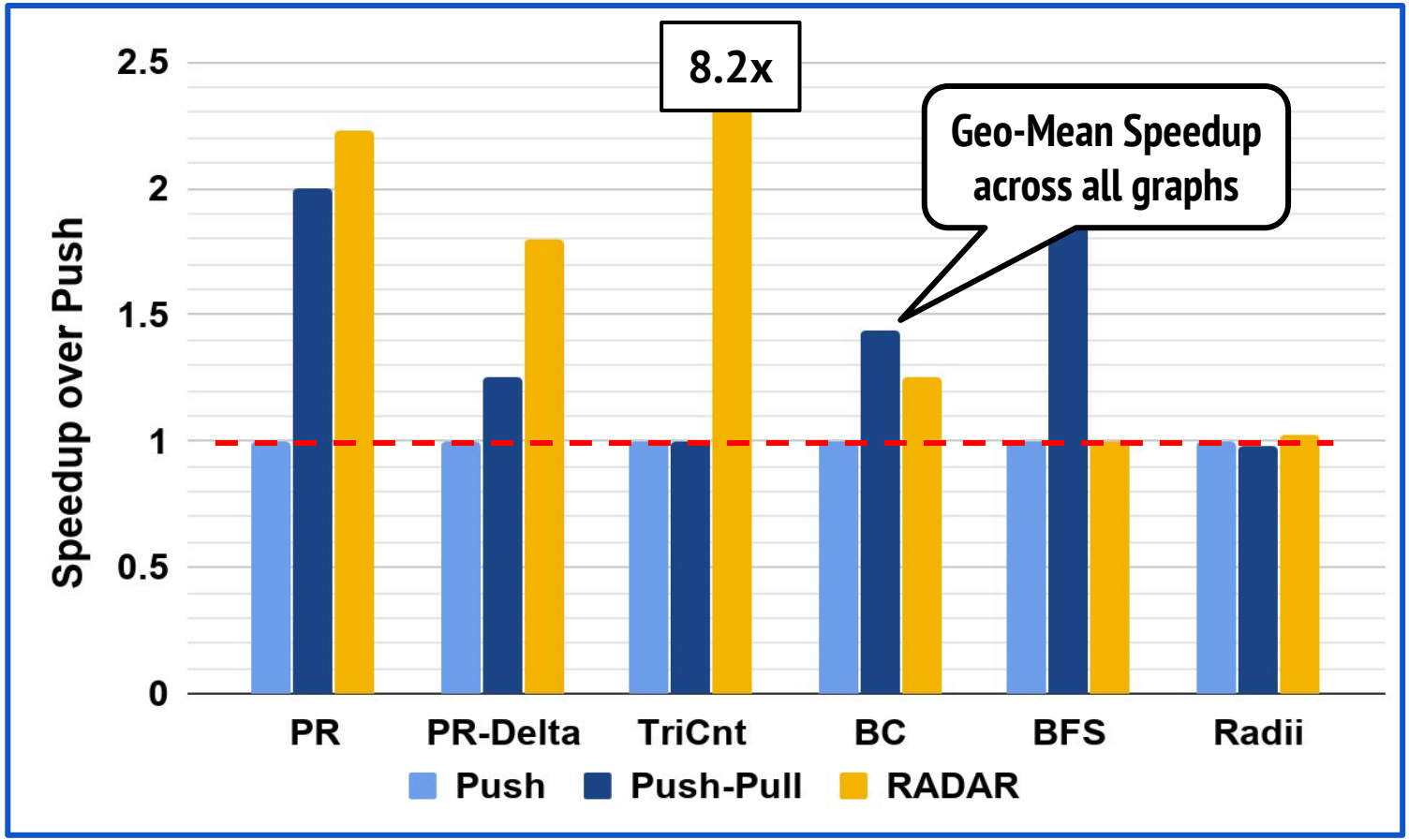
```
parallel_for dst in G:
  for src in in_neigh(dst):
    if src in Frontier:
      vtxData[dst] += auxData[src]
```

Pull phase trade-off ⇒ work efficiency *vs* eliminating atomic updates

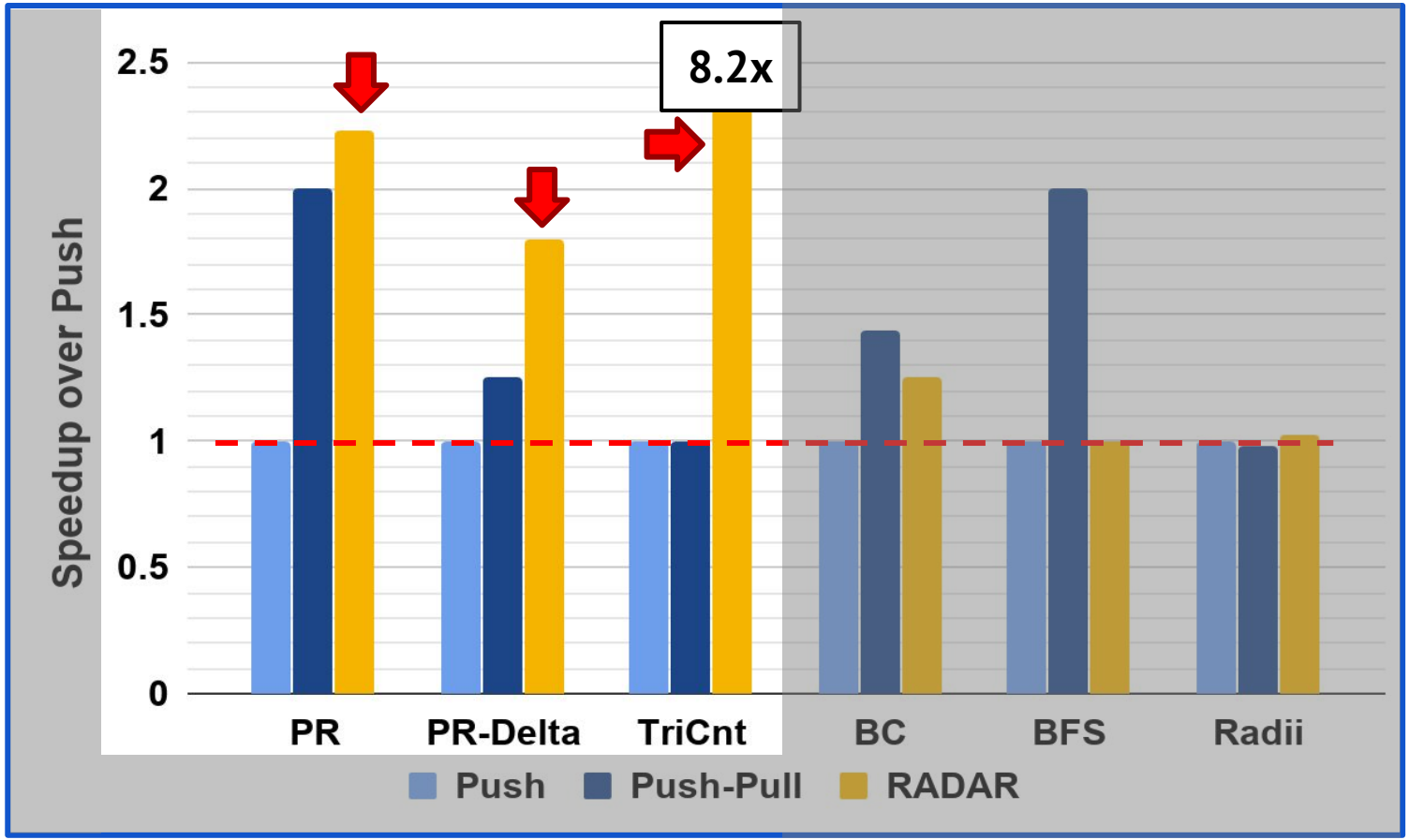
Advantages of RADAR over Push-Pull

	Push-Pull Direction Switching	RADAR
Work-Efficiency	Work-inefficient	No change
Memory Footprint	**2X** (outCSR + inCSR)	**~1X** (reordered outCSR)
Preprocessing Cost	O(E.logE)	O(V.logV + E)

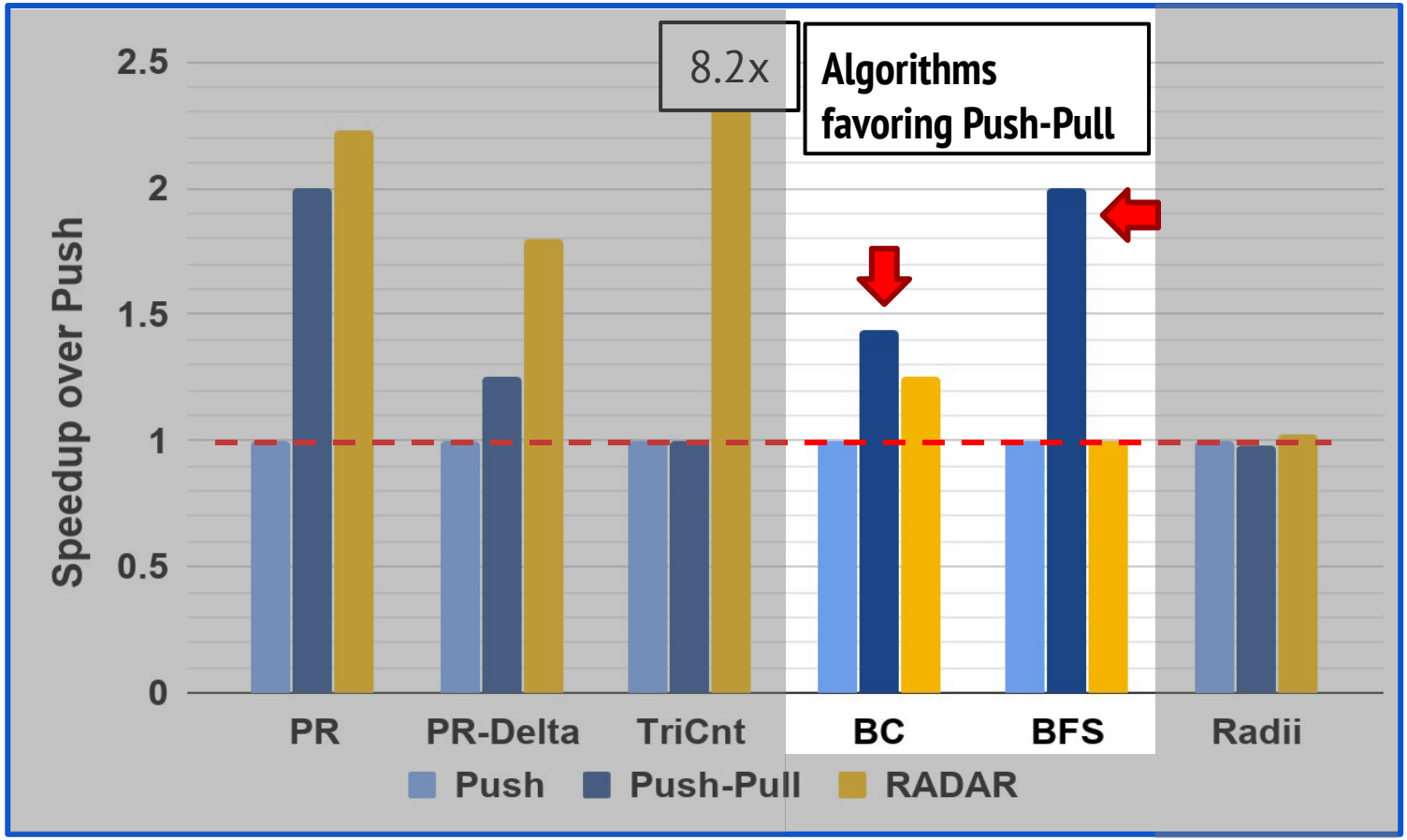
Performance Of RADAR Compared To Push-Pull



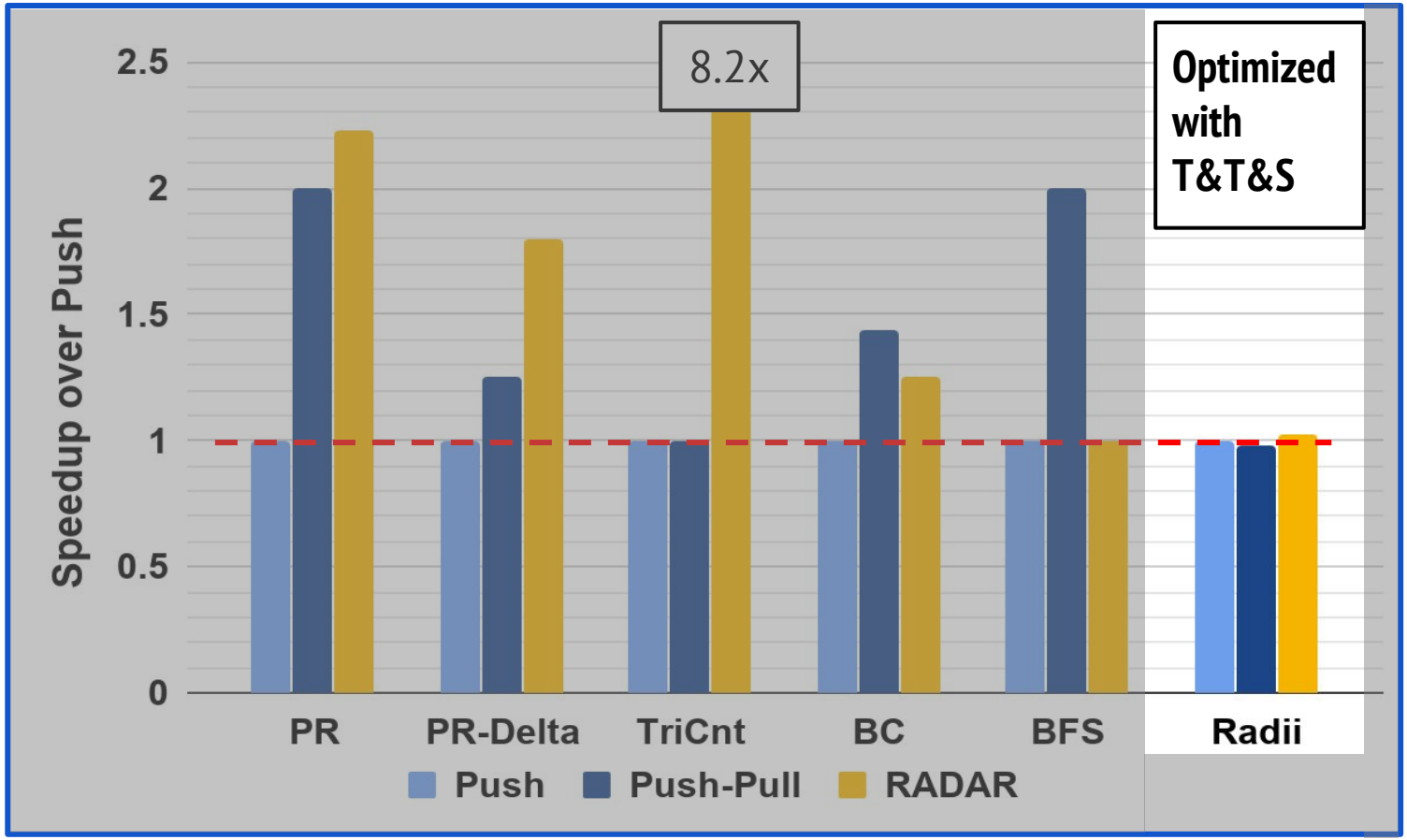
Performance Of RADAR Compared To Push-Pull



Performance Of RADAR Compared To Push-Pull



Performance Of RADAR Compared To Push-Pull

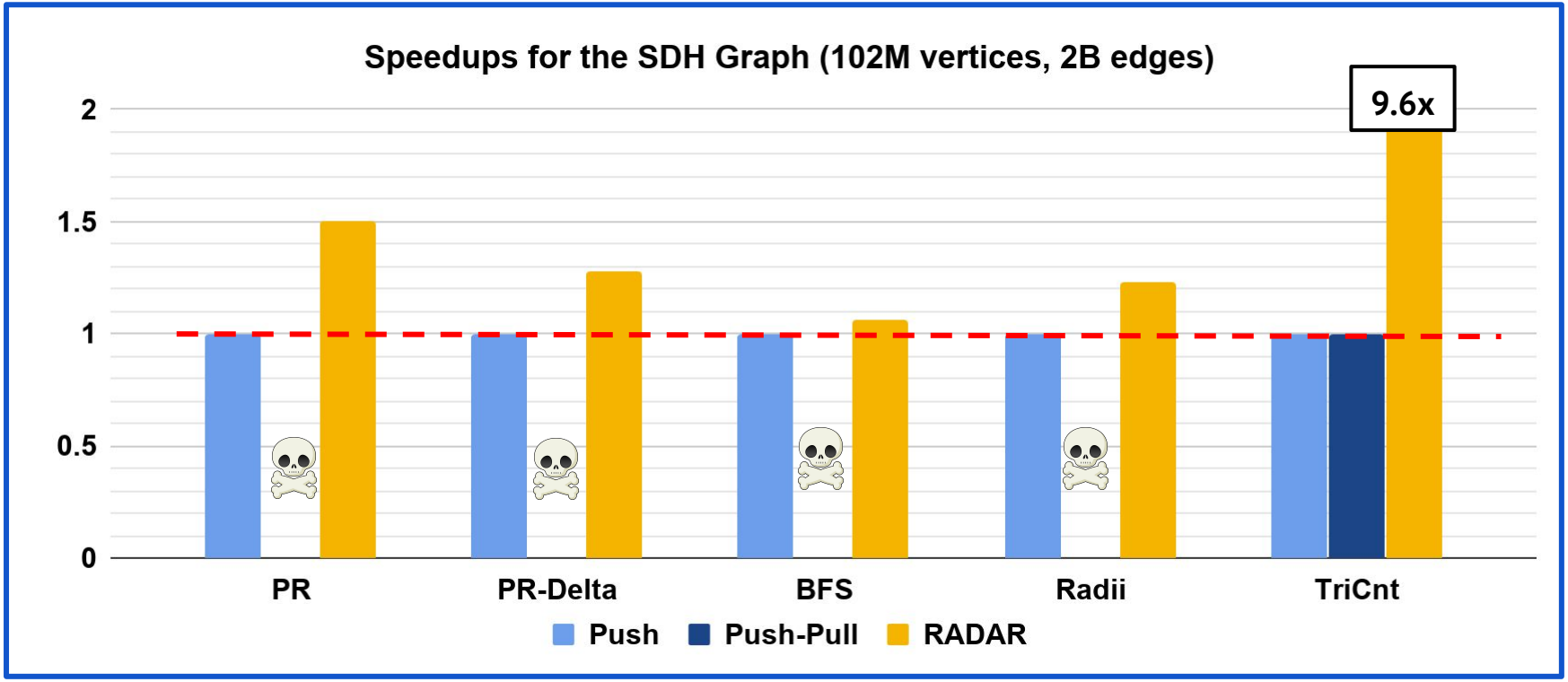


RADAR Can Support Larger Input Graphs

Push-Pull increases memory footprint by 2x

RADAR Can Support Larger Input Graphs

Push-Pull increases memory footprint by 2x



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 *without their overheads*
- RADAR has many advantages over the state-of-the-art Push-Pull optimization

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

Electrical & Computer ENGINEERING

Carnegie Mellon

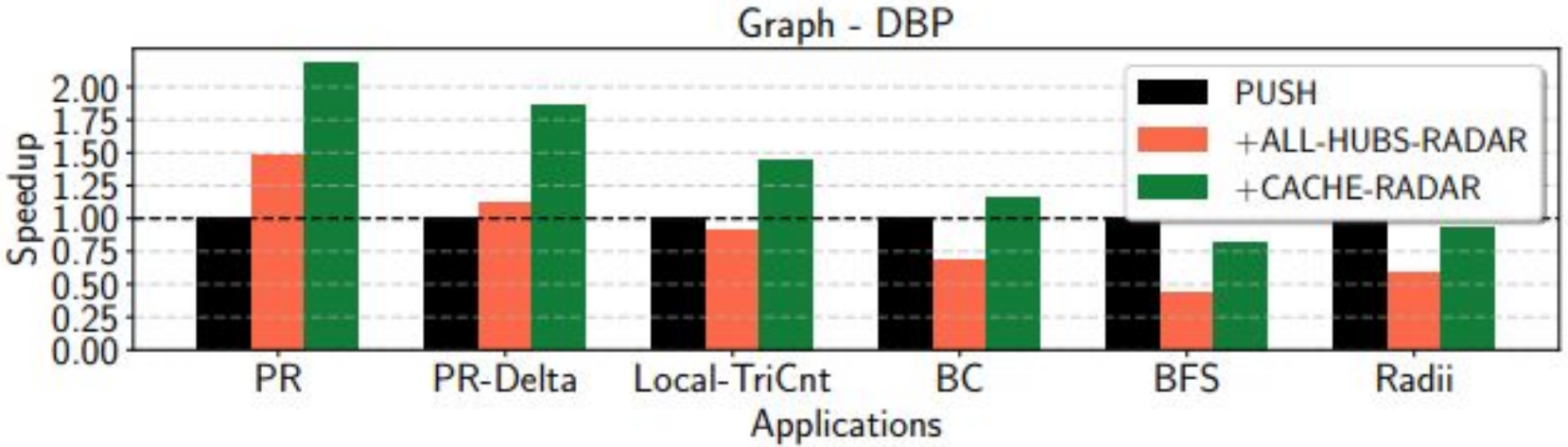
Conclusions

- 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

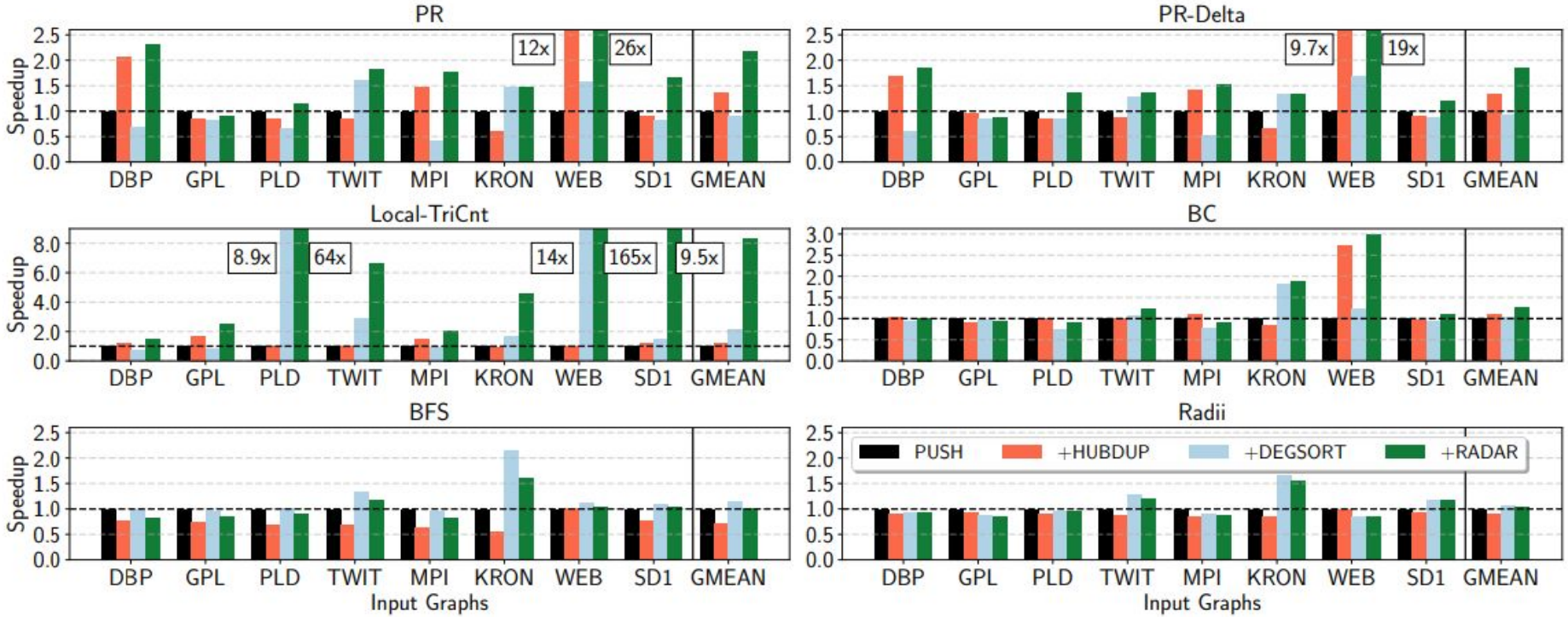
https://github.com/CMUAbstract/RADAR-Graph

Backup Slides

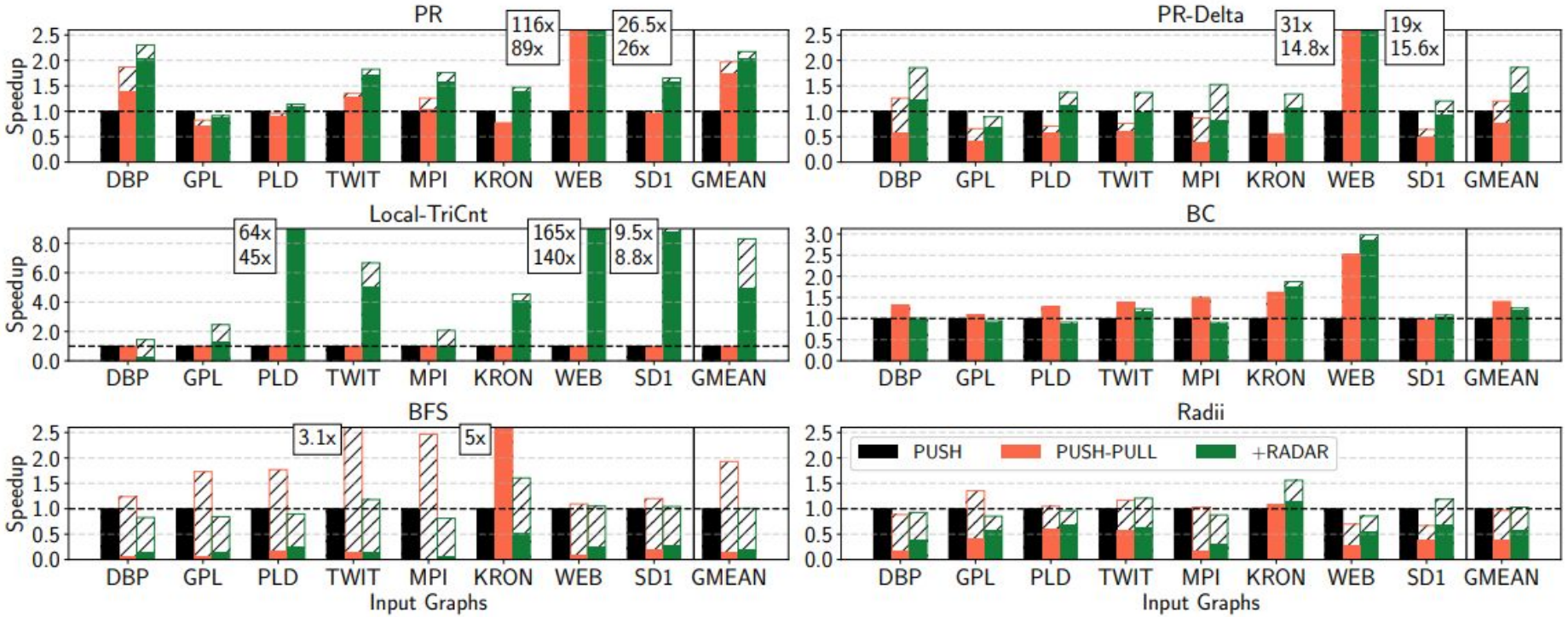
Importance of sizing hubs to LLC size



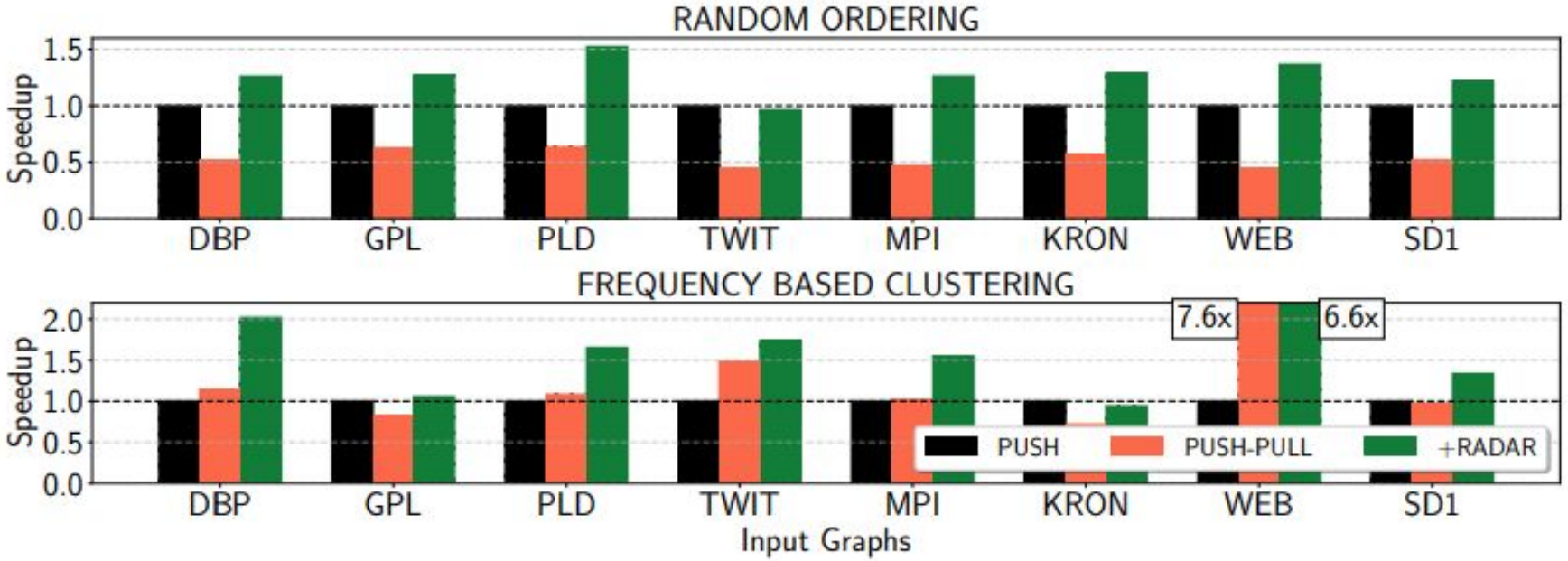
All Results for RADAR v/s HUBDUP & DegSort



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

	Complexity	DBP	GPL	PLD	TWIT	MPI	KRON	WEB	SD1
HUBDUP	$O(V)$	0.06s	0.11s	0.14s	0.24s	0.22s	0.23s	0.19s	0.37s
DegSort/RADAR	$O(VlogV + E)$	0.88s	2.37s	2.29s	8.26s	19.06s	2.94s	3.49s	7.42s
Push-Pull	$O(ElogE)$	2.96s	7.03s	3.91s	9.68s	47.71s	0s	10.86s	12.51s

HUBDUP	RADAR	Push-Pull
0.17s	3.93s	9.08