When Is Graph Reordering An Optimization?

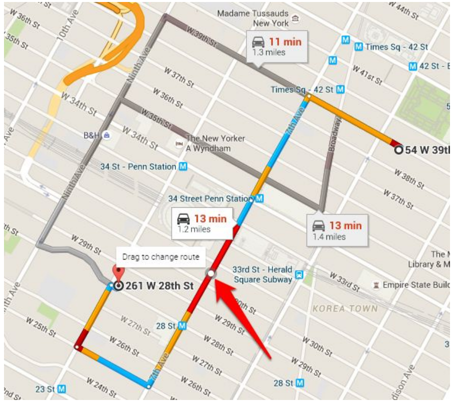
A Cross Application and Input Graph Study on the Effectiveness of Lightweight Graph Reordering

Vignesh Balaji   Brandon Lucia

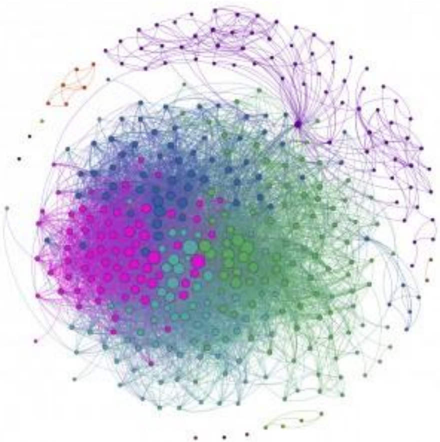
Electrical & Computer ENGINEERING

Carnegie Mellon

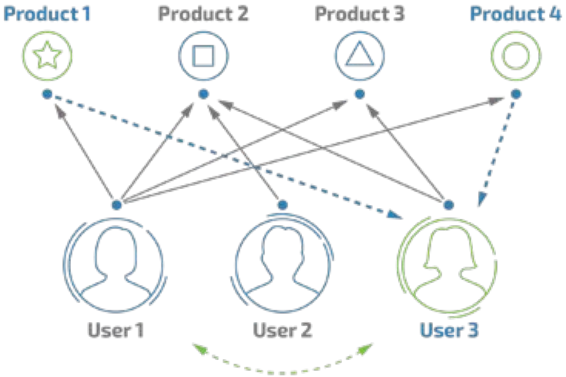
Graph Processing Has Many Applications



Path Planning

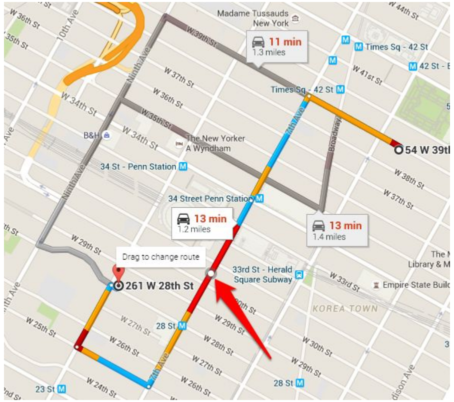


Social network analysis

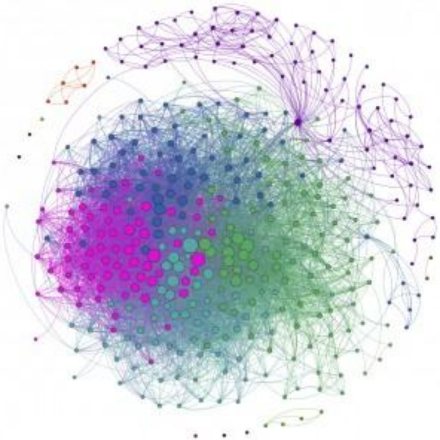


Recommender systems

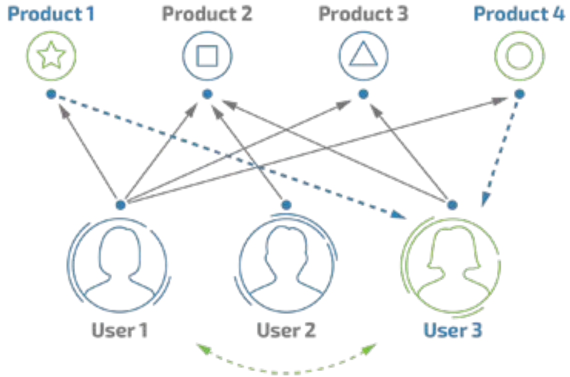
Graph Processing Has Many Applications



Path Planning

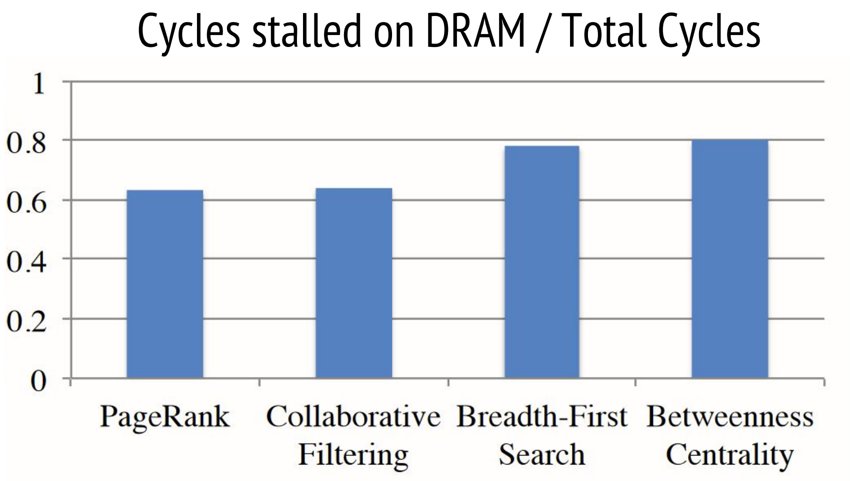


Social network analysis



Recommender systems

Graph Applications Are Memory Bound



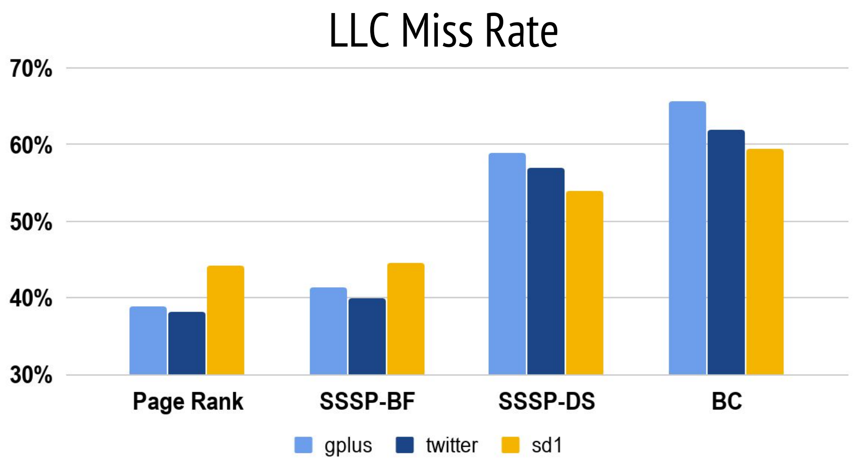
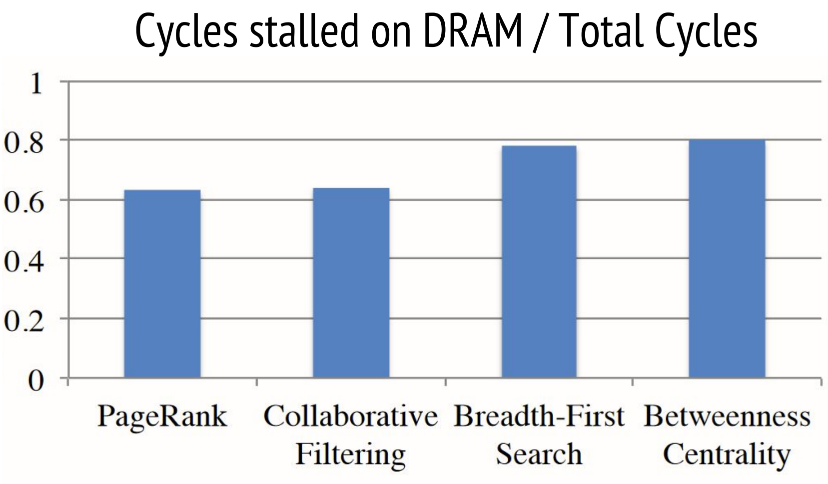
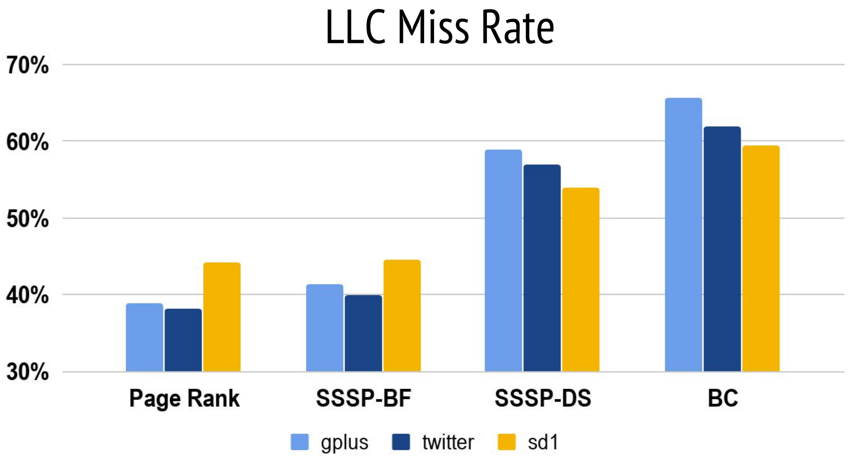


Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

Graph Applications Are Memory Bound





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

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

Reason For Poor Locality - Irregular Memory Accesses

Reason For Poor Locality - Irregular Memory Accesses

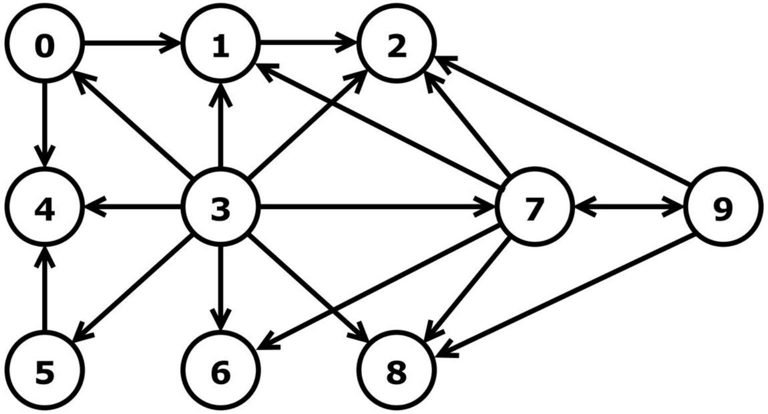
```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u],...)
```

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

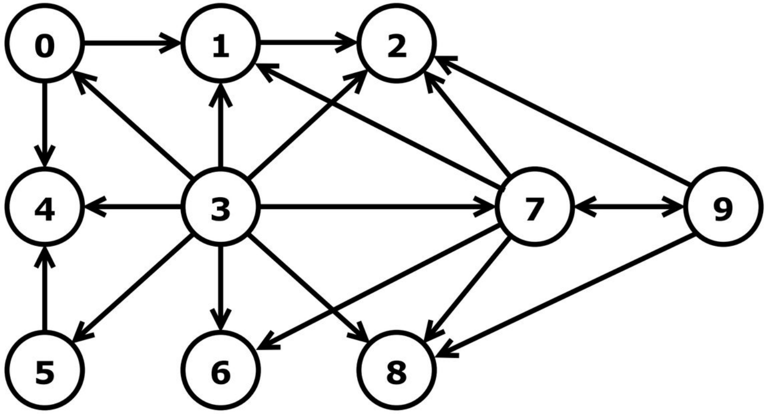


Input Graph

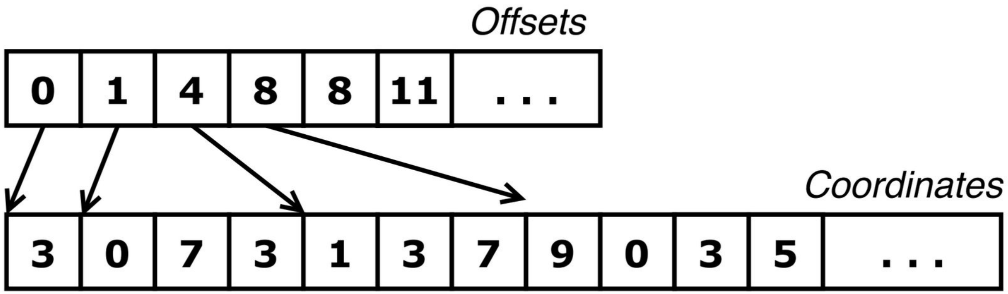
Reason For Poor Locality - Irregular Memory Accesses

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

Typical graph processing kernel



Input Graph



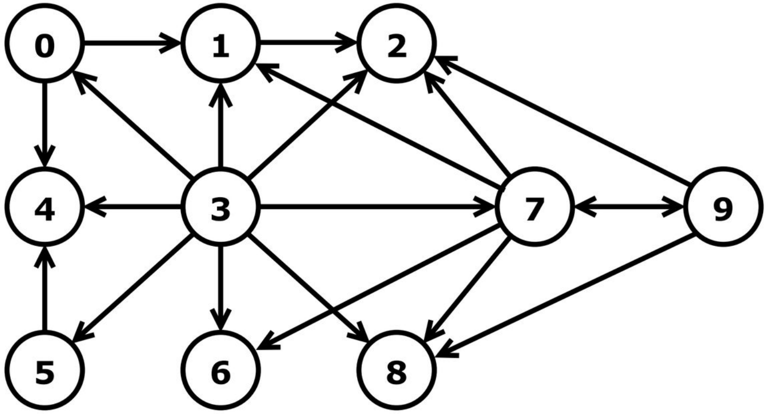
Compressed Sparse Row (CSR) Representation

Reason For Poor Locality - Irregular Memory Accesses

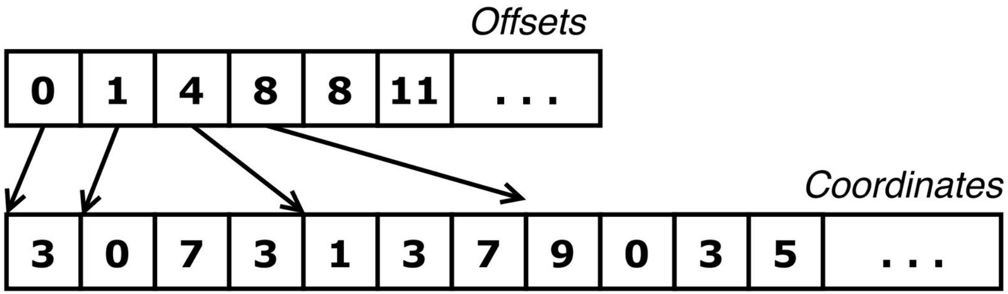
```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u],...)
```

Irregular accesses to vtxData array

Typical graph processing kernel



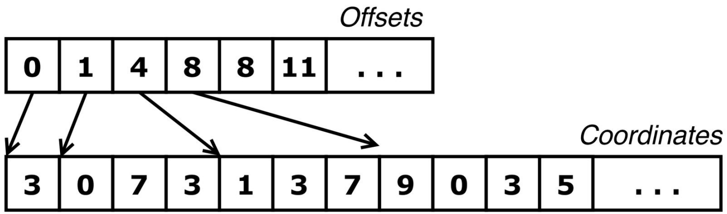
Input Graph

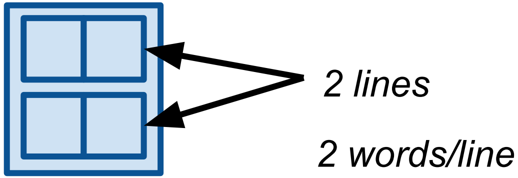


Compressed Sparse Row (CSR) Representation

Irregular Accesses Have Poor Temporal And Spatial Locality

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

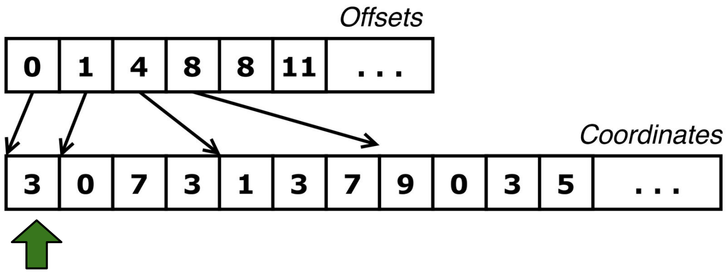




LLC

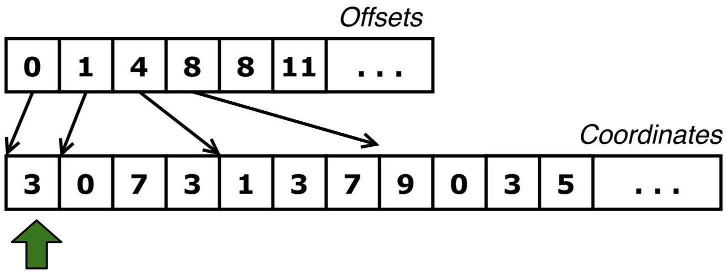
Irregular Accesses Have Poor Temporal And Spatial Locality

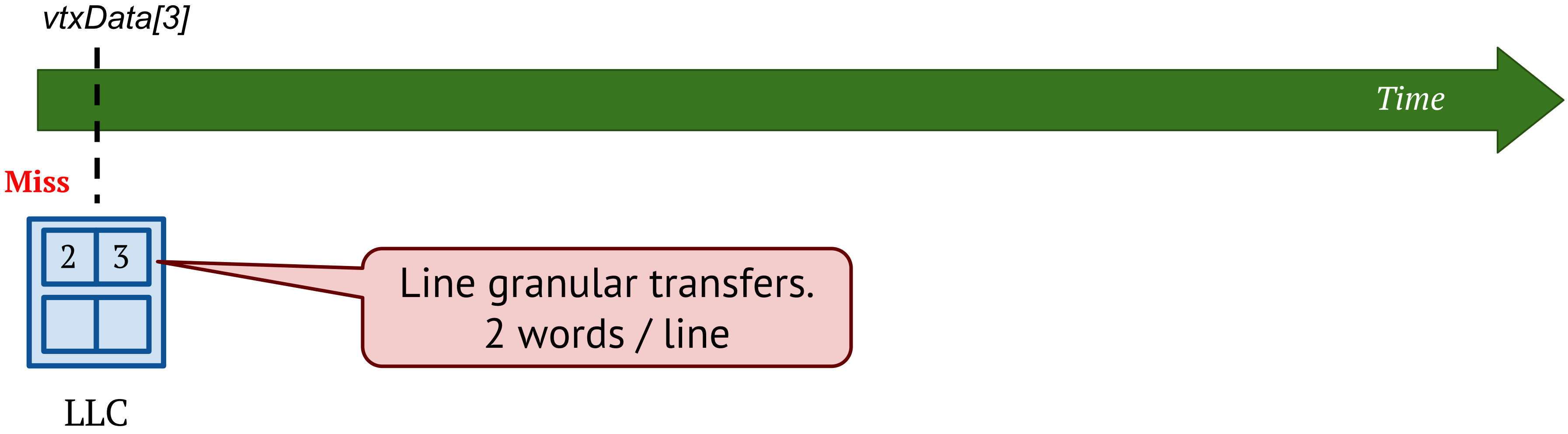
```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u],...)
```



Irregular Accesses Have Poor Temporal And Spatial Locality

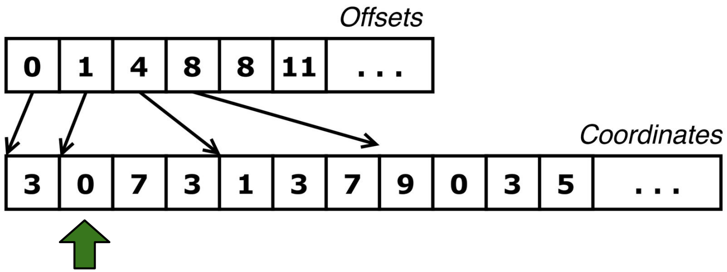
```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u],...)
```





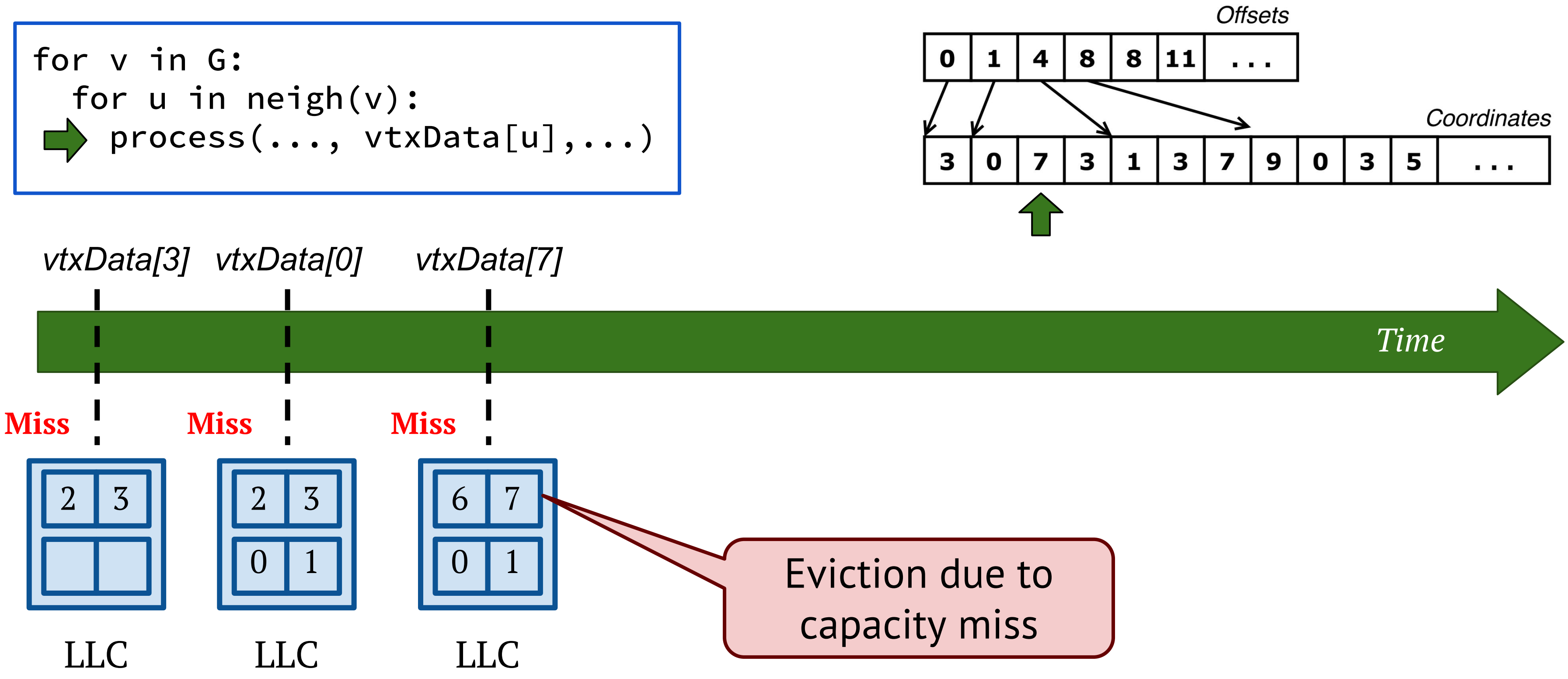
Irregular Accesses Have Poor Temporal And Spatial Locality

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



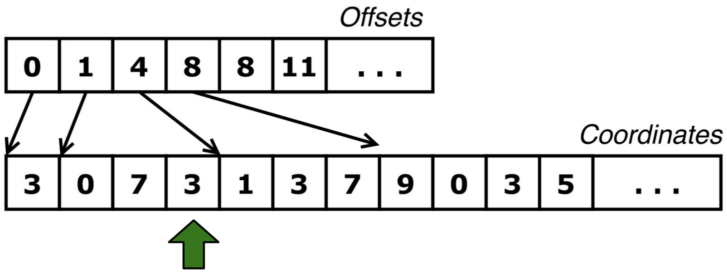
Irregular Accesses Have Poor Temporal And Spatial Locality

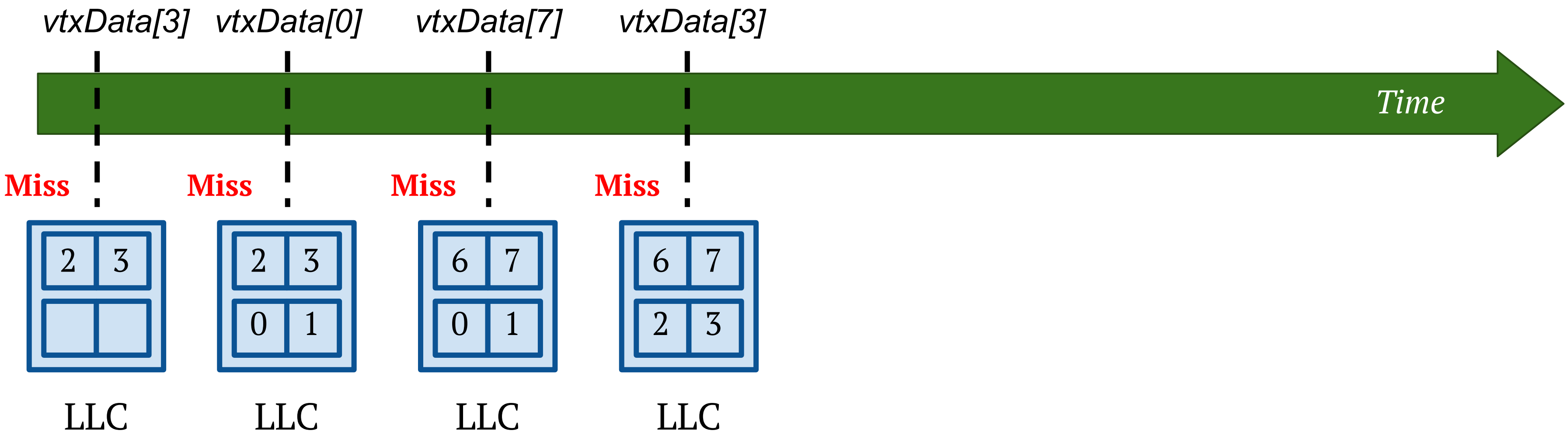
```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u],...)
```



Irregular Accesses Have Poor Temporal And Spatial Locality

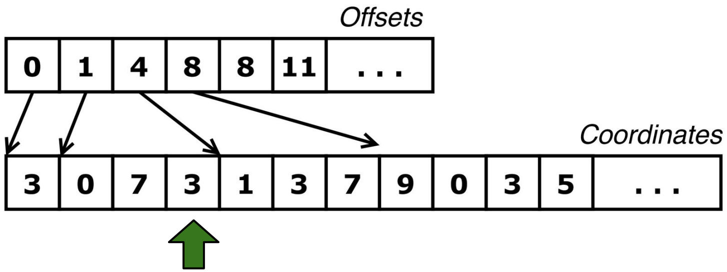
```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u],...)
```

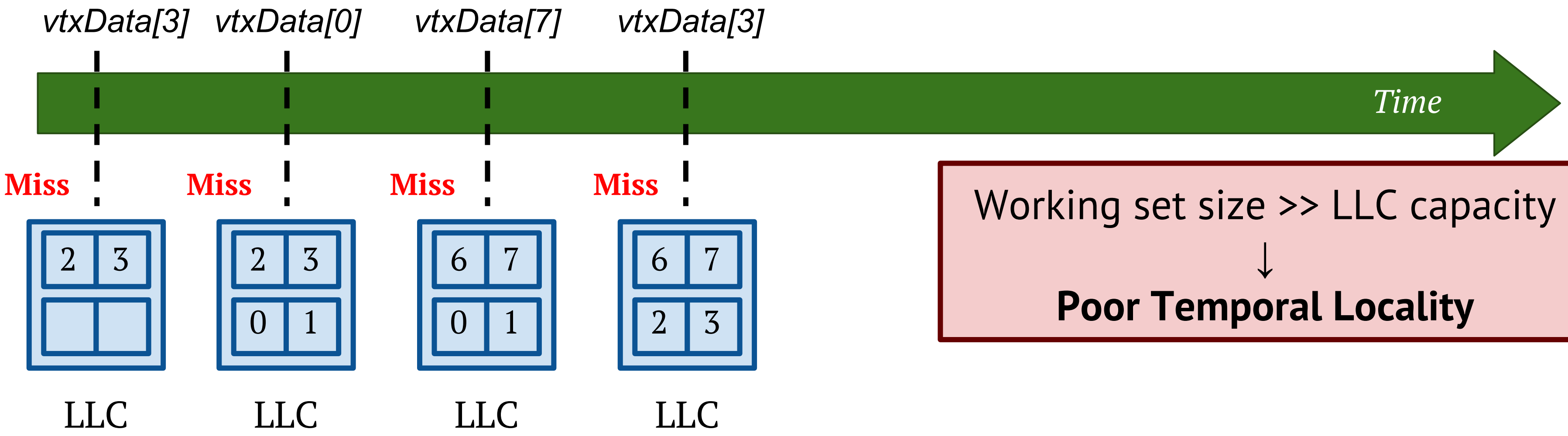




Irregular Accesses Have Poor Temporal And Spatial Locality

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

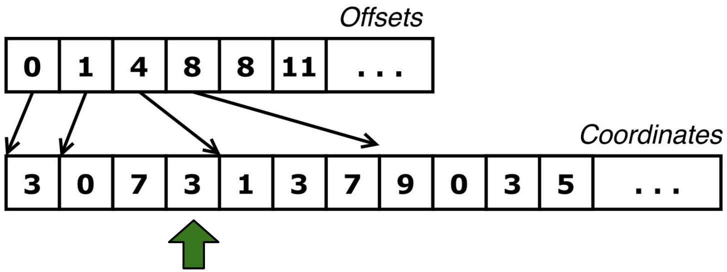


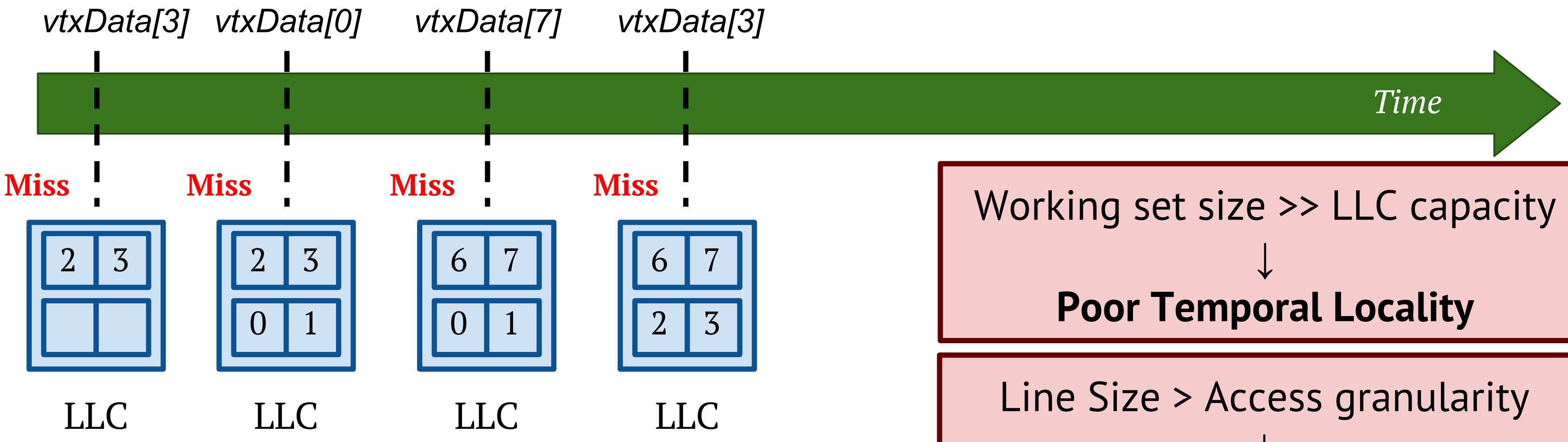


Working set size >> LLC capacity
↓
Poor Temporal Locality

Irregular Accesses Have Poor Temporal And Spatial Locality

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





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

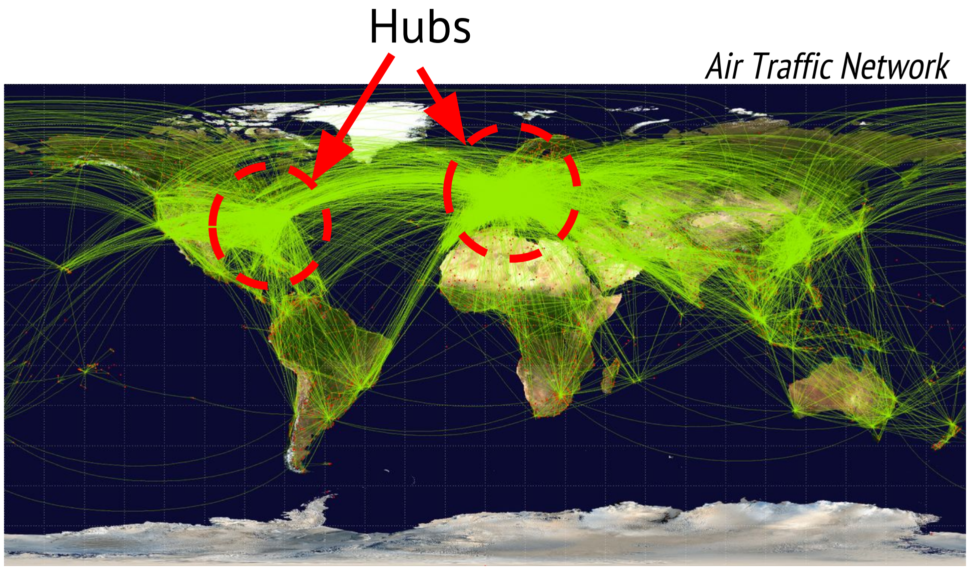
- Poor Locality of Graph Processing Applications ✔
- **Improving locality through Graph Reordering**
- Graph Reordering Challenge - *Application and Input-dependent Speedups*
- When is Graph Reordering an Optimization?
- Selective Graph Reordering

Real-world Graphs Offer Opportunities To Improve Locality

Air Traffic Network

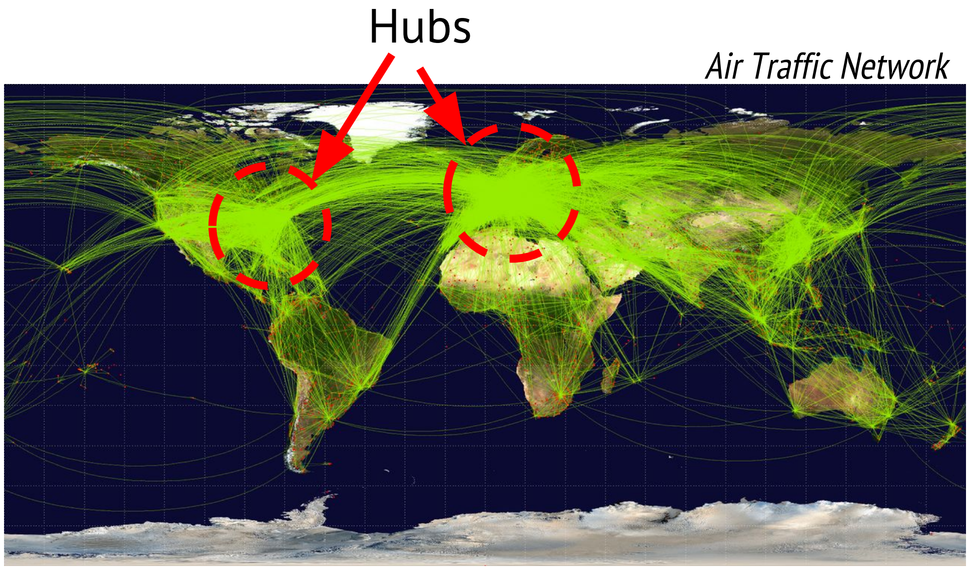
Power-law Degree Distribution

Real-world Graphs Offer Opportunities To Improve Locality

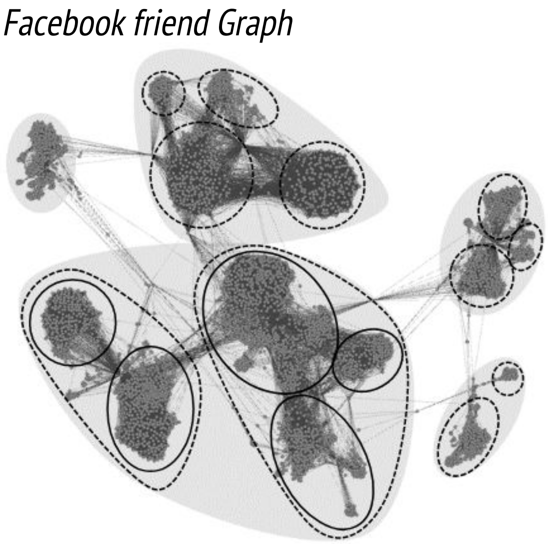


Power-law Degree Distribution

Real-world Graphs Offer Opportunities To Improve Locality



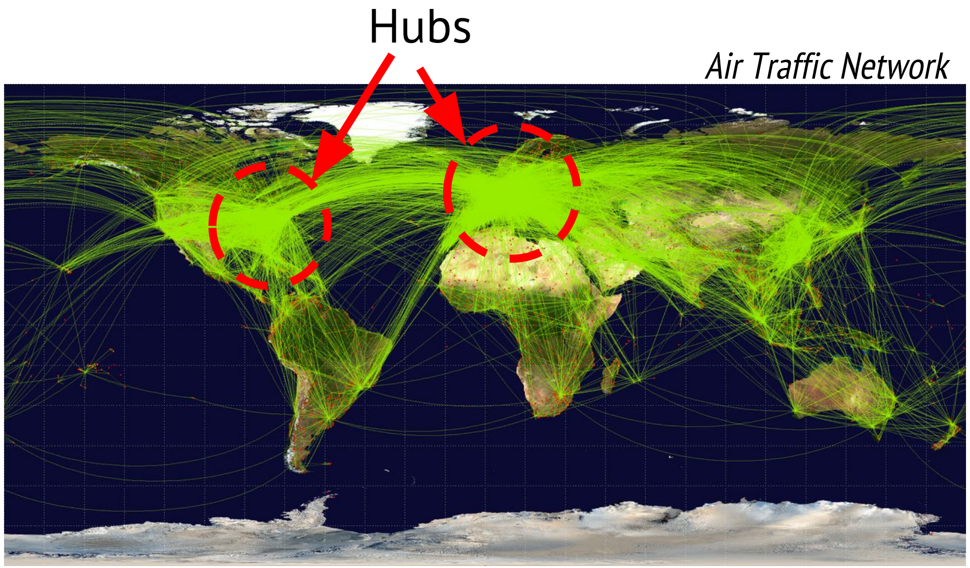
Power-law Degree Distribution



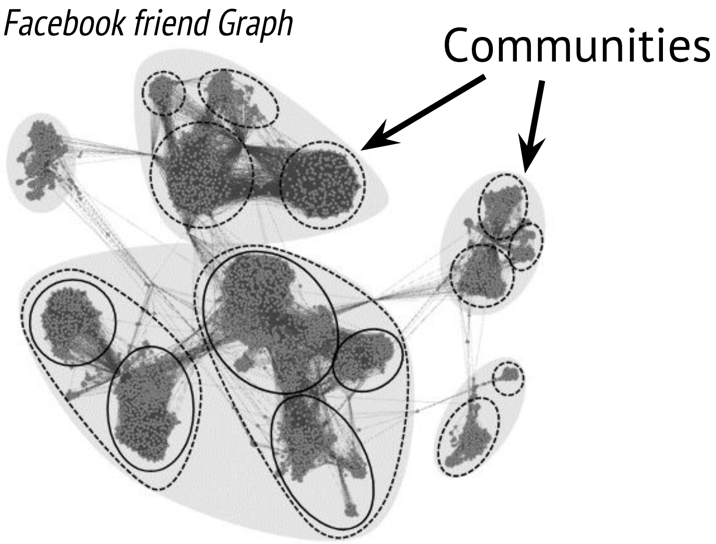
Community Structure

Right figure from "Rabbit Order: Just-in-time Parallel Reordering for Fast Graph Analysis" IPDPS 2016

Real-world Graphs Offer Opportunities To Improve Locality



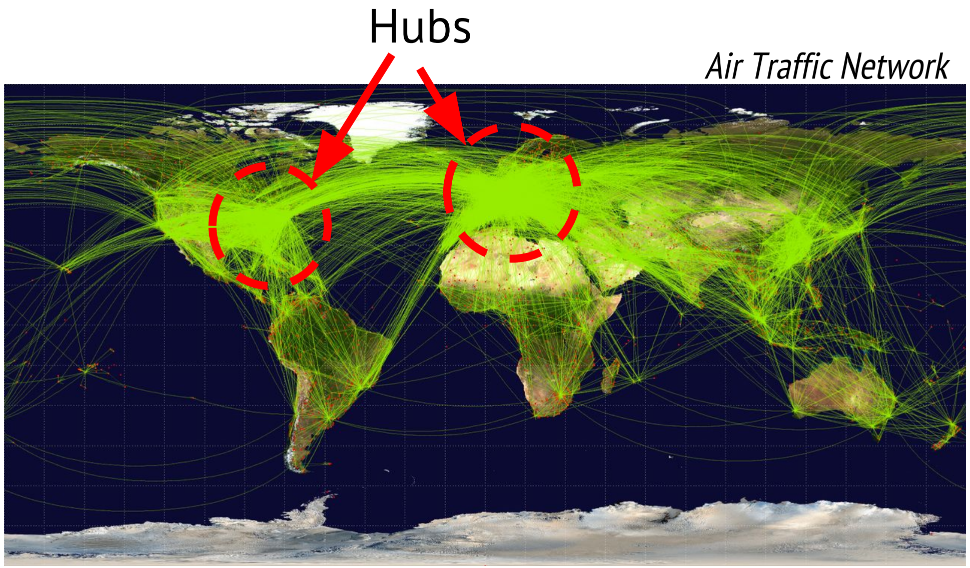
Power-law Degree Distribution



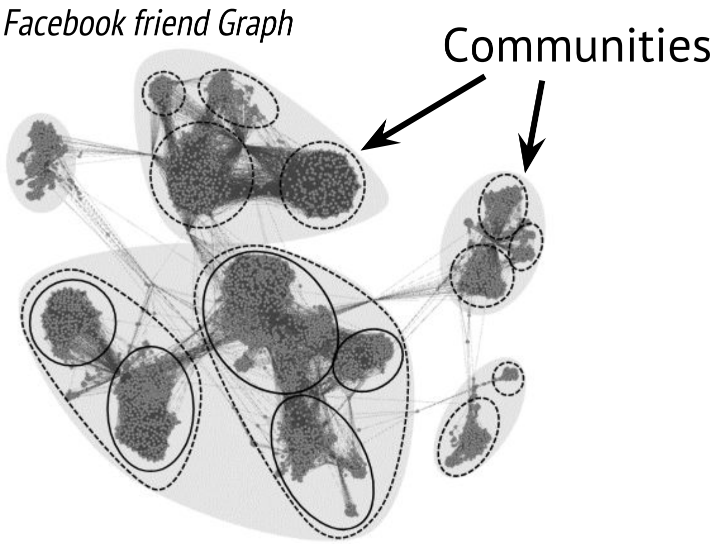
Community Structure

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



Community Structure

Observation: Subset of vertices are accessed together

Right figure from "Rabbit Order: Just-in-time Parallel Reordering for Fast Graph Analysis" IPDPS 2016

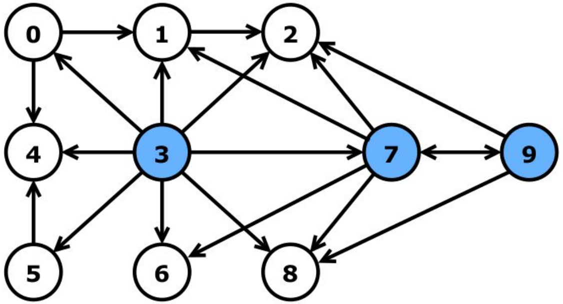
Reordering To Improve Locality of Graph Applications

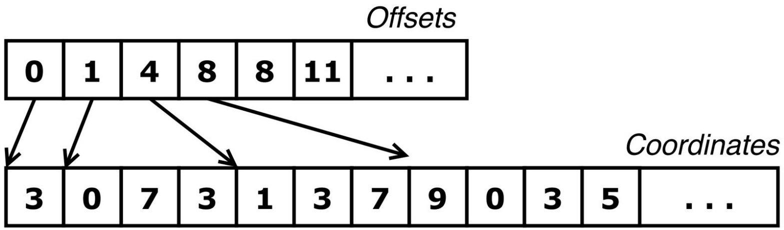
Key Insight: Store commonly accessed vertices contiguously in memory

Reordering To Improve Locality of Graph Applications

Key Insight: Store commonly accessed vertices contiguously in memory

Power-law graph

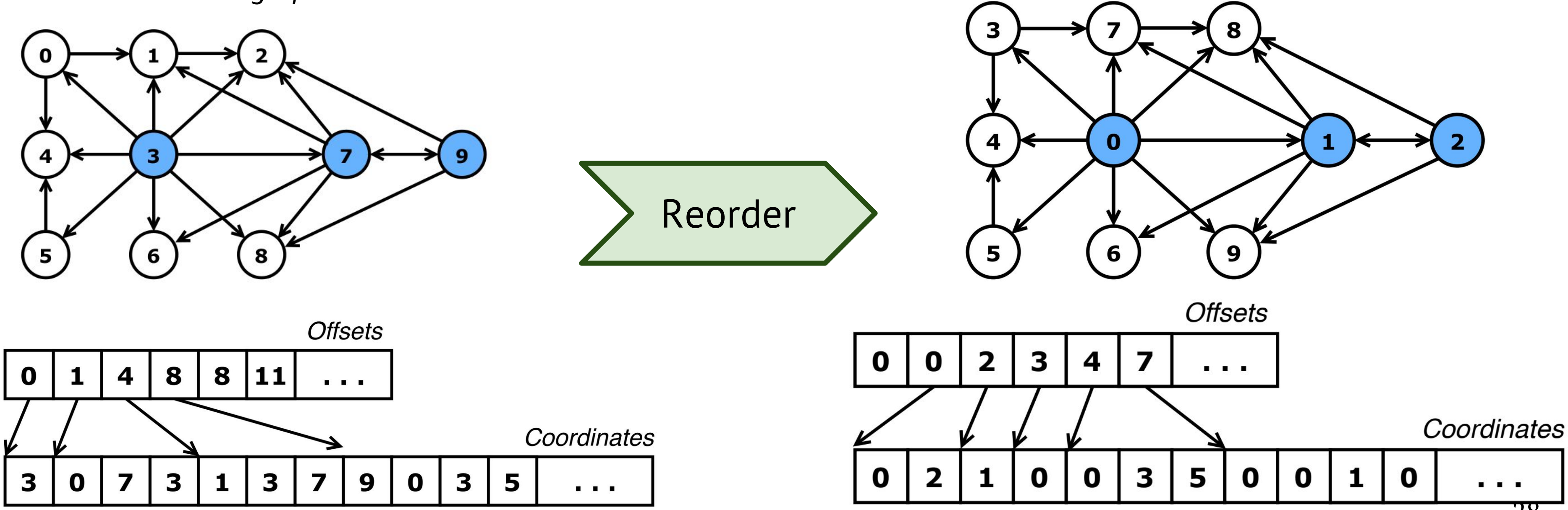




Reordering To Improve Locality of Graph Applications

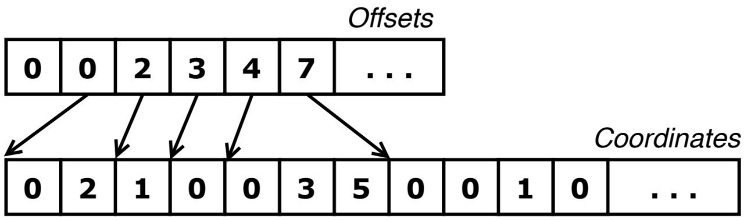
Key Insight: Store commonly accessed vertices contiguously in memory

Power-law graph

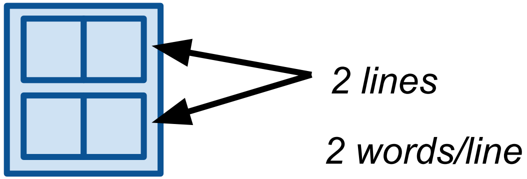


Reordering Improves Spatial & Temporal Locality

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



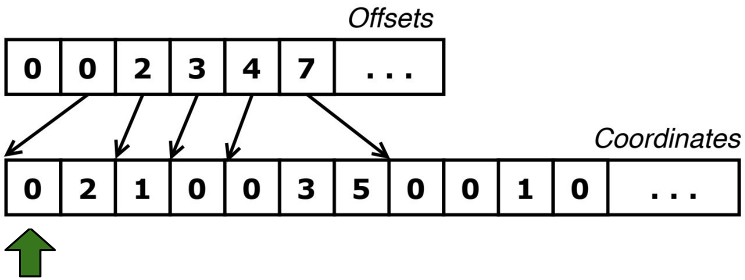
Reordered CSR



LLC

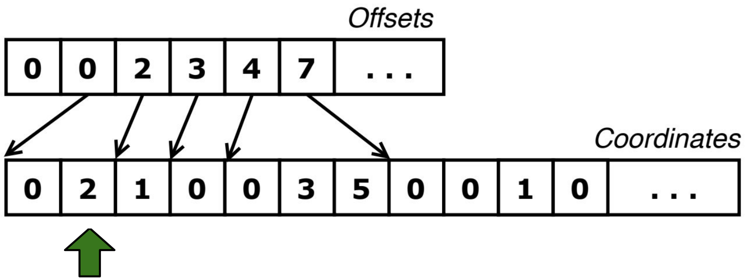
Reordering Improves Spatial & Temporal Locality

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



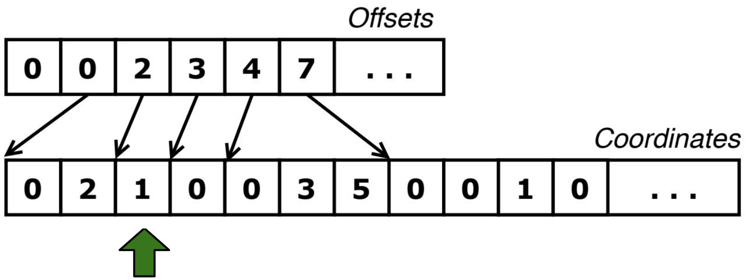
Reordering Improves Spatial & Temporal Locality

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



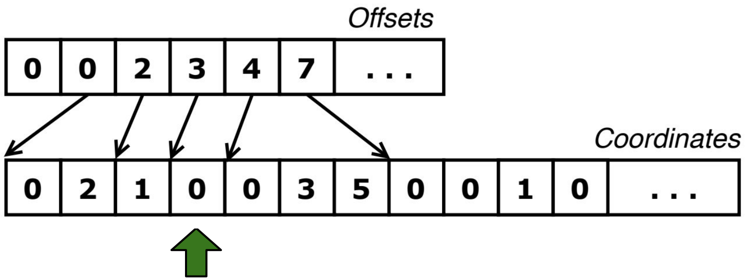
Reordering Improves Spatial & Temporal Locality

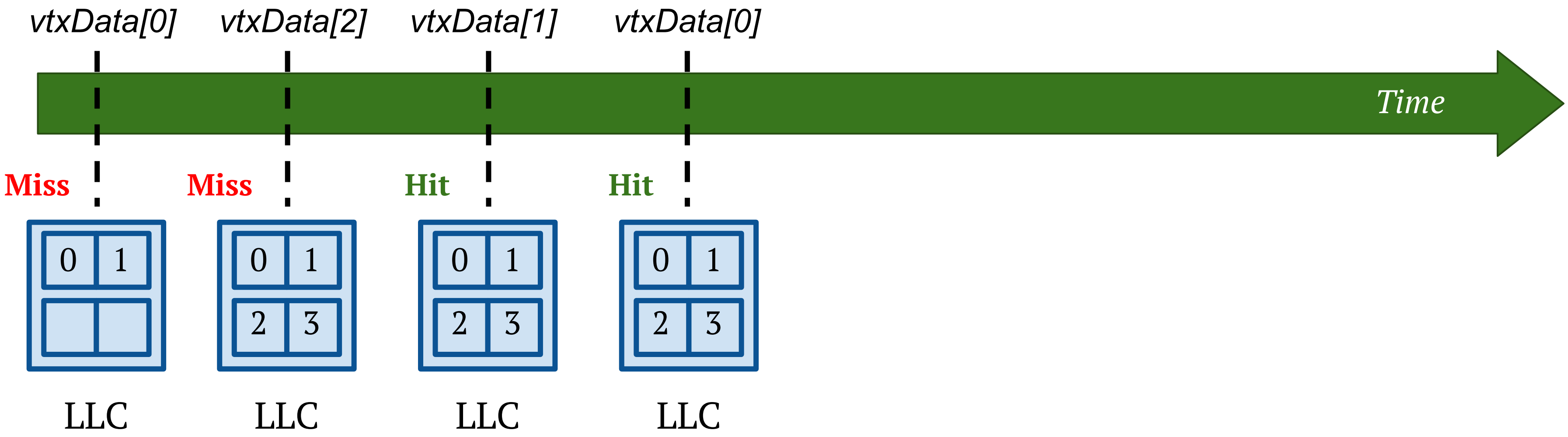
```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u],...)
```



Reordering Improves Spatial & Temporal Locality

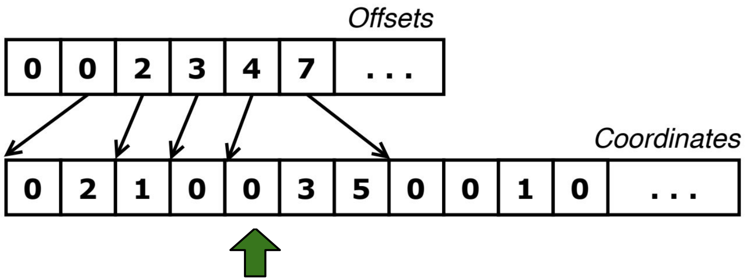
```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u],...)
```

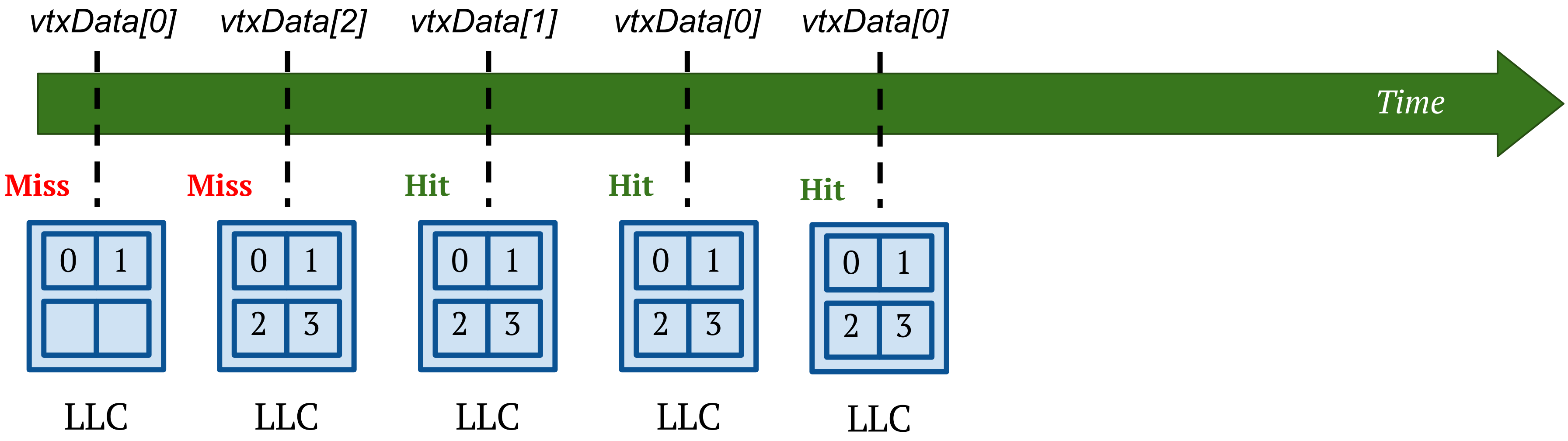




Reordering Improves Spatial & Temporal Locality

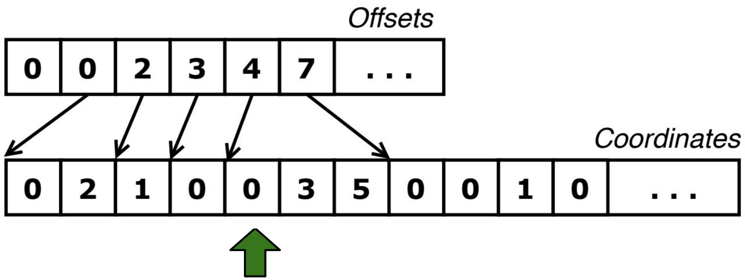
```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u],...)
```

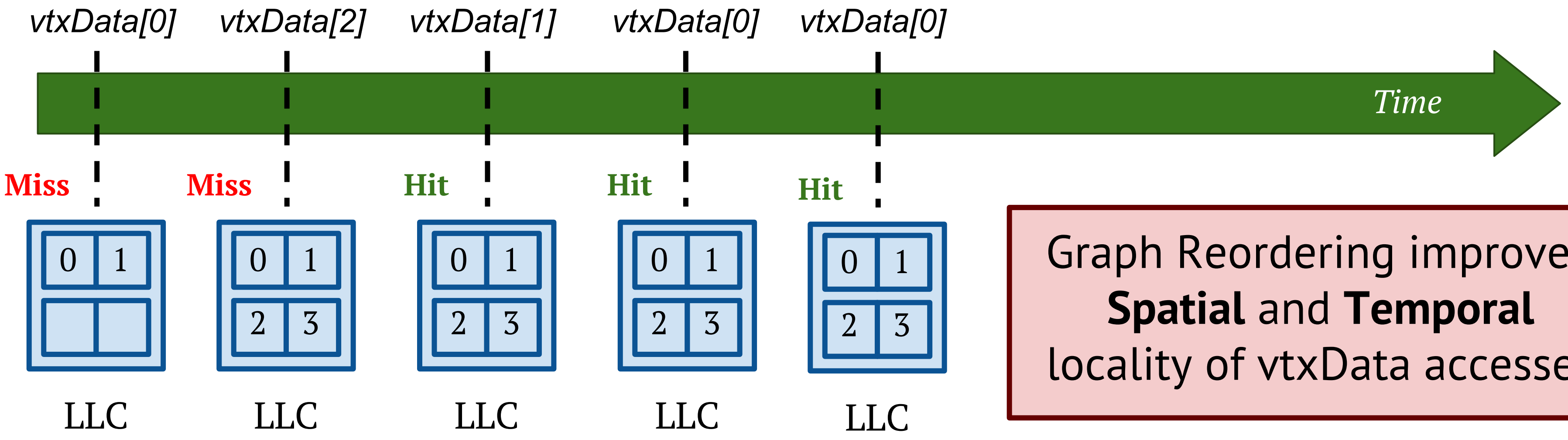




Reordering Improves Spatial & Temporal Locality

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





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

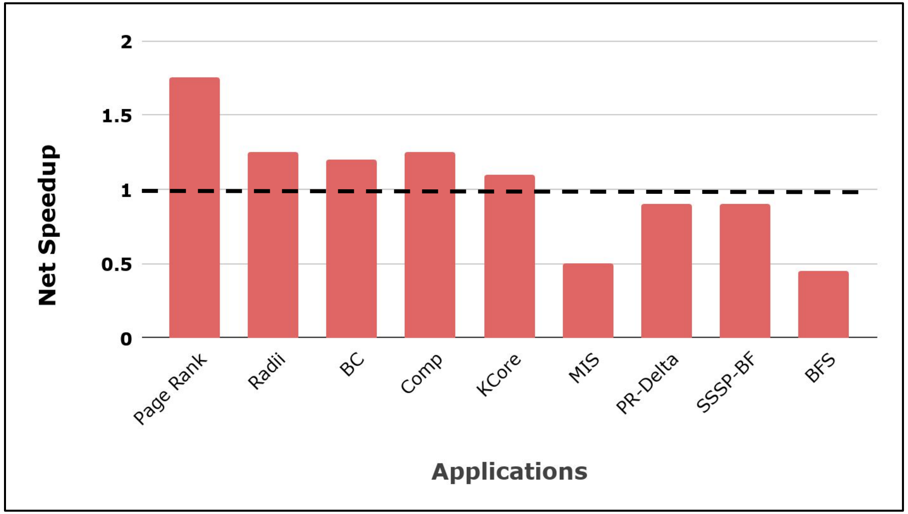
Outline

- 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

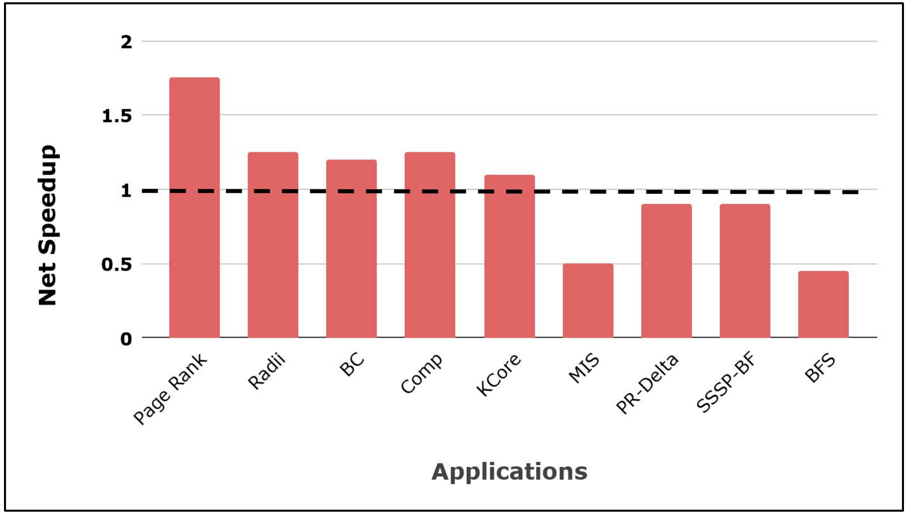
- ❖ Poor Locality of Graph Processing Applications ✔
- ❖ Improving locality through Graph Reordering ✔
- ❖ **Graph Reordering Challenge - *Application and Input-dependent Speedups***
- ❖ When is Graph Reordering an Optimization?
- ❖ Selective Graph Reordering

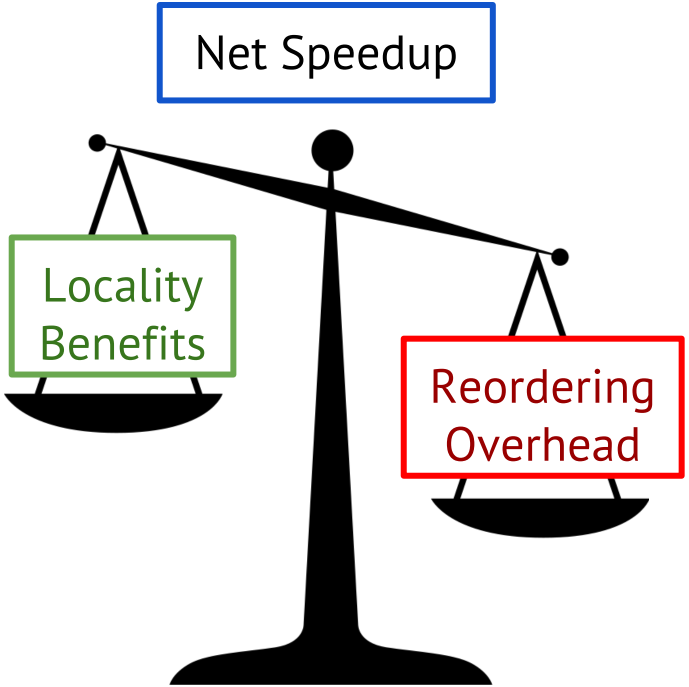
Graph Reordering Is Not A Panacea



$$Speedup = \frac{T_{Original}}{T_{Reordered} + Reordering\,Time}$$

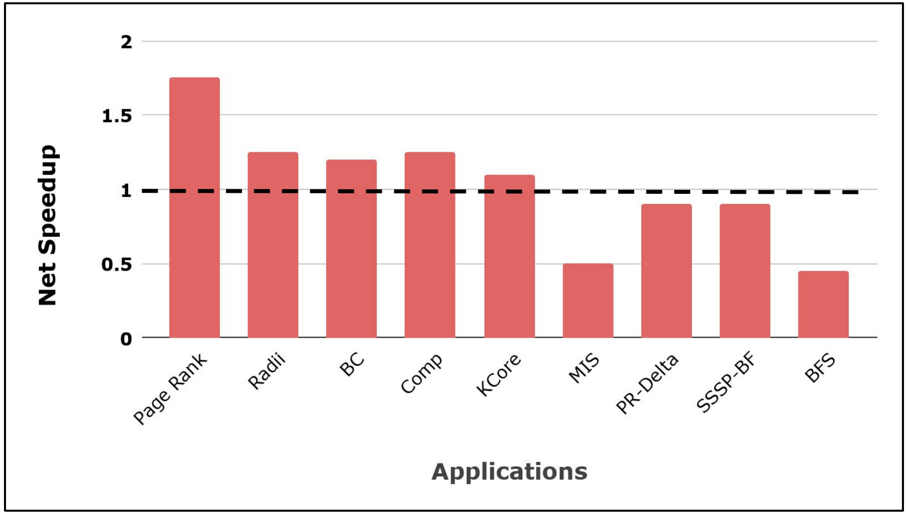
Graph Reordering Is Not A Panacea

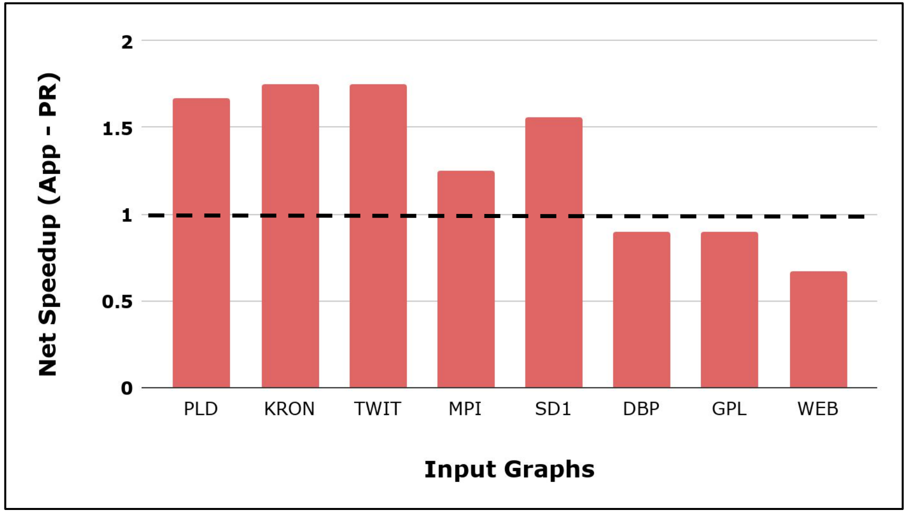




$$Speedup = \frac{T_{Original}}{T_{Reordered} + Reordering\,Time}$$

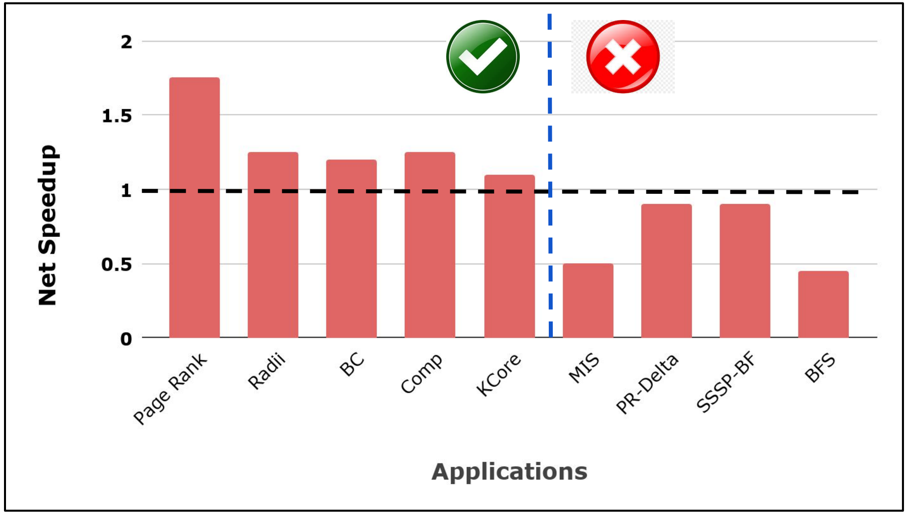
Graph Reordering Is Not A Panacea

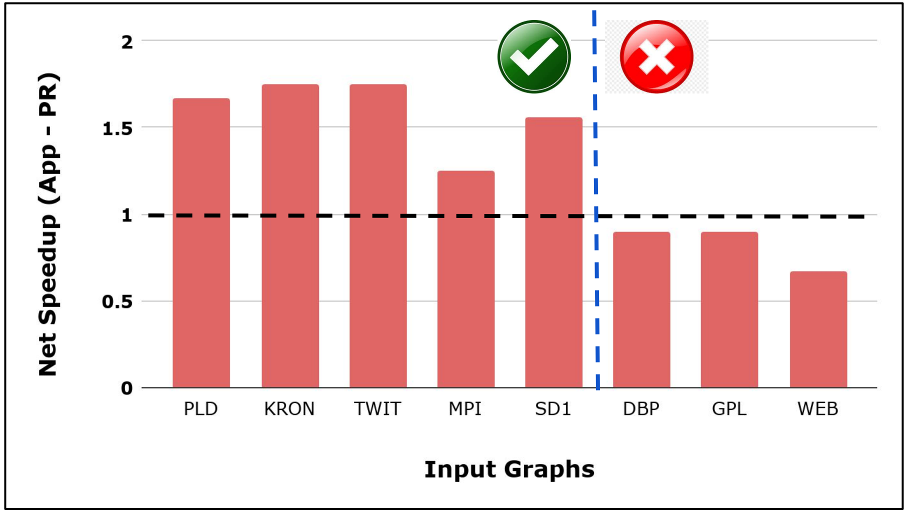




$$Speedup = \frac{T_{Original}}{T_{Reordered} + Reordering\,Time}$$

Graph Reordering Is Not A Panacea





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

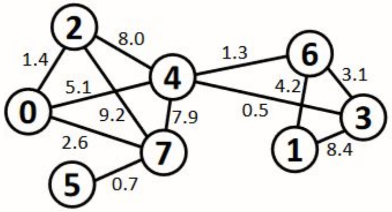
(dual-Socket, 28 cores, 35MB LLC, 64GB DRAM)

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



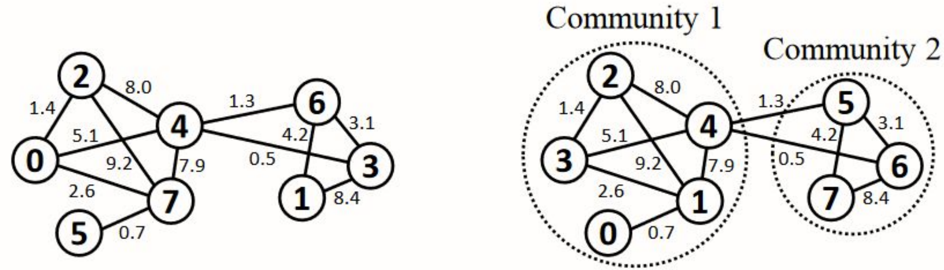
(a) Randomly ordered graph

	0	1	2	3	4	5	6	7
0			1.4		5.1			2.6
1				8.4			4.2	
2	1.4				8.0			9.2
3		8.4			0.5		3.1	
4	5.1		8.0	0.5			1.3	7.9
5								0.7
6		4.2		3.1	1.3			
7	2.6		9.2		7.9	0.7		

(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

	0	1	2	3	4	5	6	7
0			1.4		5.1			2.6
1				8.4			4.2	
2	1.4				8.0			9.2
3		8.4			0.5		3.1	
4	5.1		8.0	0.5			1.3	7.9
5								0.7
6		4.2		3.1	1.3			
7	2.6		9.2		7.9	0.7		

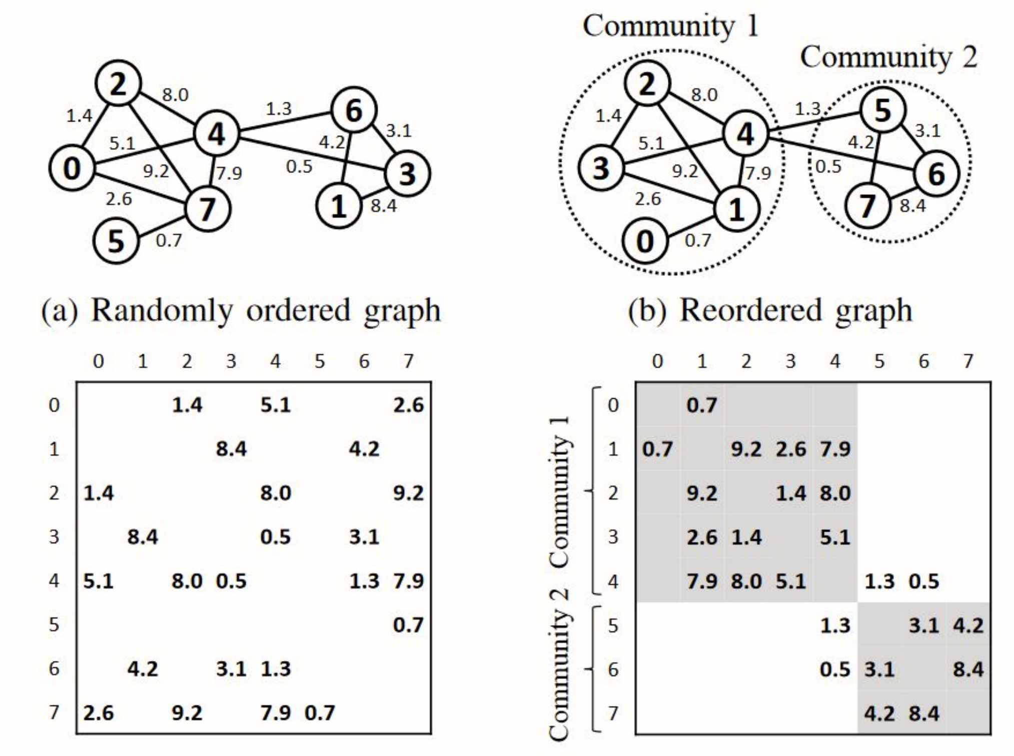
(c) Adjacency matrix of graph (a)

		0	1	2	3	4	5	6	7
Community 1	0		0.7						
	1	0.7		9.2	2.6	7.9			
	2		9.2		1.4	8.0			
	3		2.6	1.4		5.1			
	4		7.9	8.0	5.1		1.3	0.5	
Community 2	5					1.3		3.1	4.2
	6					0.5	3.1		8.4
	7						4.2	8.4	

(d) Adjacency matrix of graph (b)

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

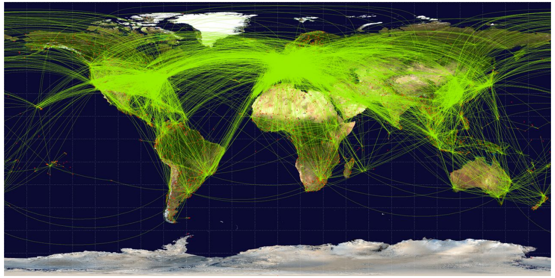
(c) Adjacency matrix of graph (a) (d) Adjacency matrix of graph (b)

- Fast community detection using *incremental aggregation*
- Complexity - **O(|E| - ck|V|)**

  where c = clustering coeff.
  k = avg. degree

Figure from "Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis" IPDPS 2016

LWR 2 & 3 - HubSorting & HubClustering



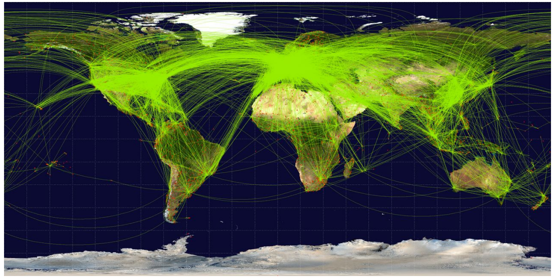
Vertex Degrees (Original)

v0	v1	v2	v3	v4	v5	v6	v7	v8	v9
4	20	4	21	25	99	6	49	64	4

degrees

Vtx IDs

LWR 2 & 3 - HubSorting & HubClustering



Vertex Degrees (Original)

v0	v1	v2	v3	v4	v5	v6	v7	v8	v9
4	20	4	21	25	99	6	49	64	4

degrees

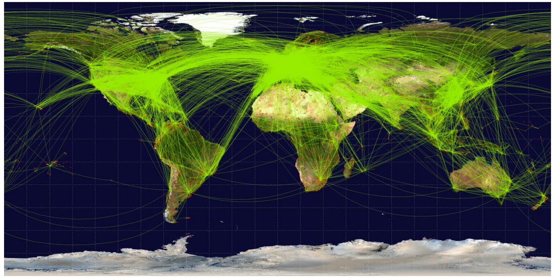
Vtx IDs

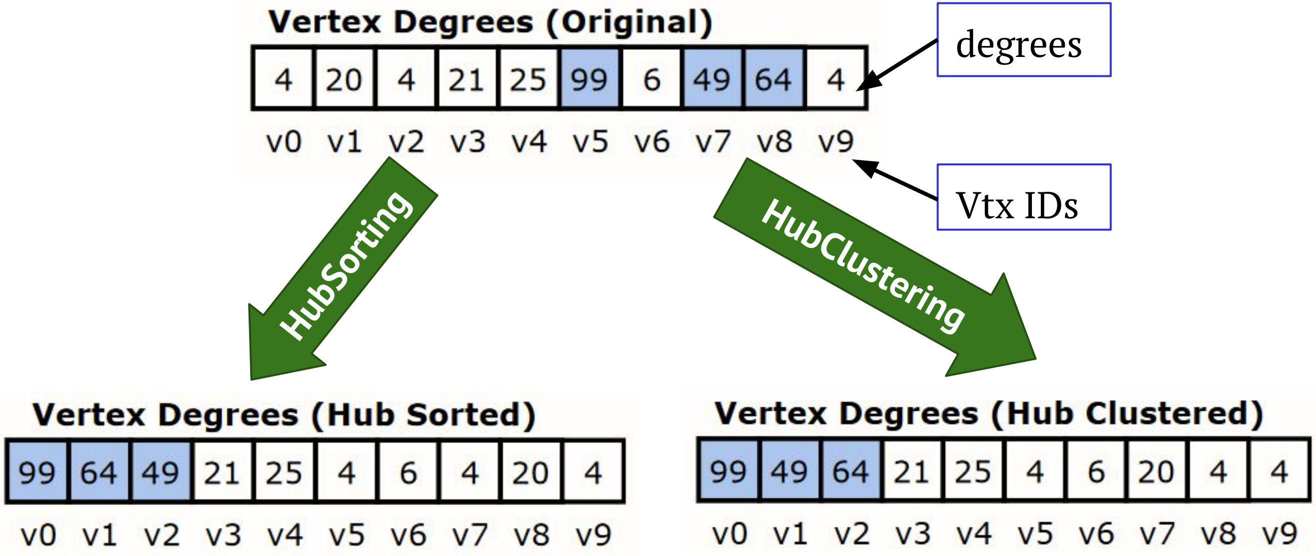
HubSorting

Vertex Degrees (Hub Sorted)

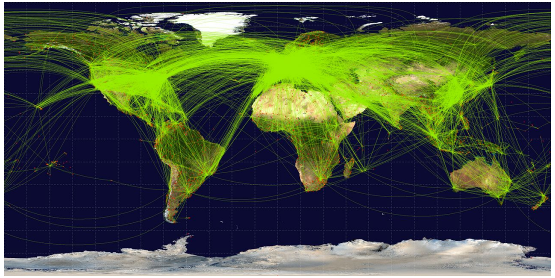
v0	v1	v2	v3	v4	v5	v6	v7	v8	v9
99	64	49	21	25	4	6	4	20	4

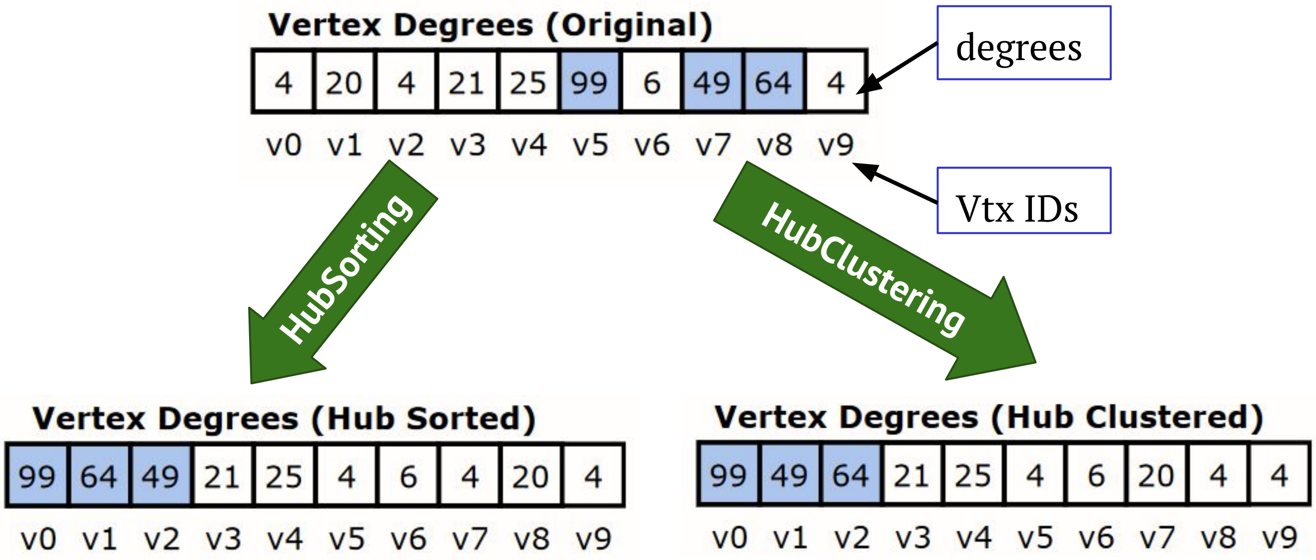
LWR 2 & 3 - HubSorting & HubClustering





LWR 2 & 3 - HubSorting & HubClustering





	HubSorting	HubClustering
Locality Benefits	Temporal *AND* Spatial ↑↑	Temporal ↑
Complexity	O(|V|.logV) ↓↓	O(|V|) ↓

Outline

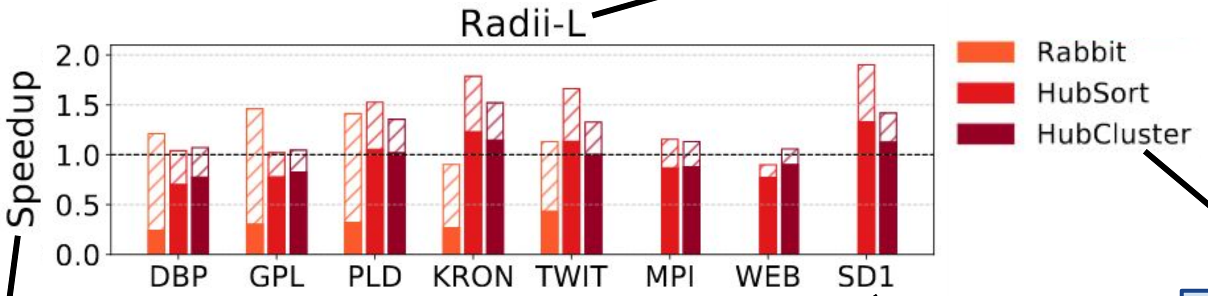
- Poor Locality of Graph Processing Applications ✔
- Improving locality through Graph Reordering ✔
- Graph Reordering Challenge - *Application and Input-dependent Speedups* ✔
- **When is Graph Reordering an Optimization?**
  - Characterization Space ✔
  - Which Applications benefit from Reordering?
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

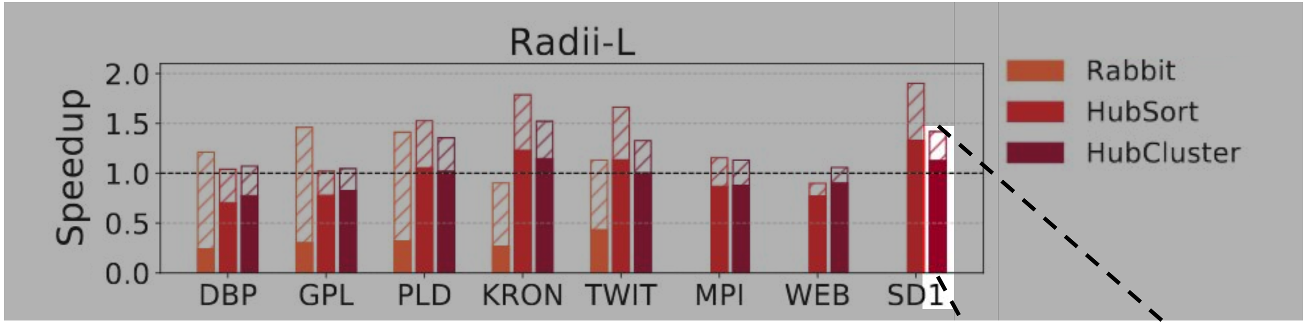


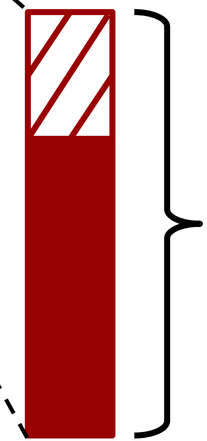
Reordering Technique

Speedup with respect to original ordering of graph

Input Graph

Legend for Results

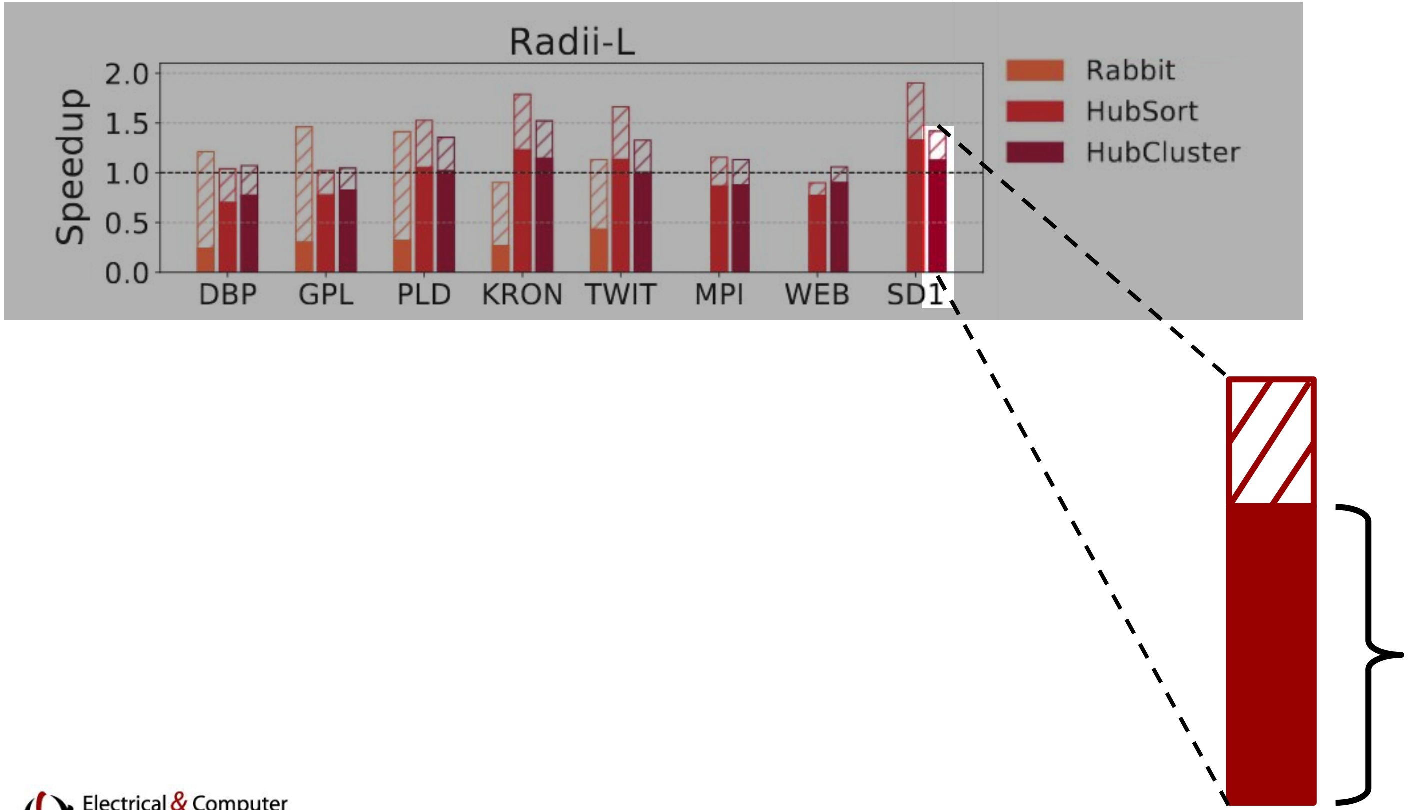




Speedup excluding the overhead of reordering

(Max Speedup)

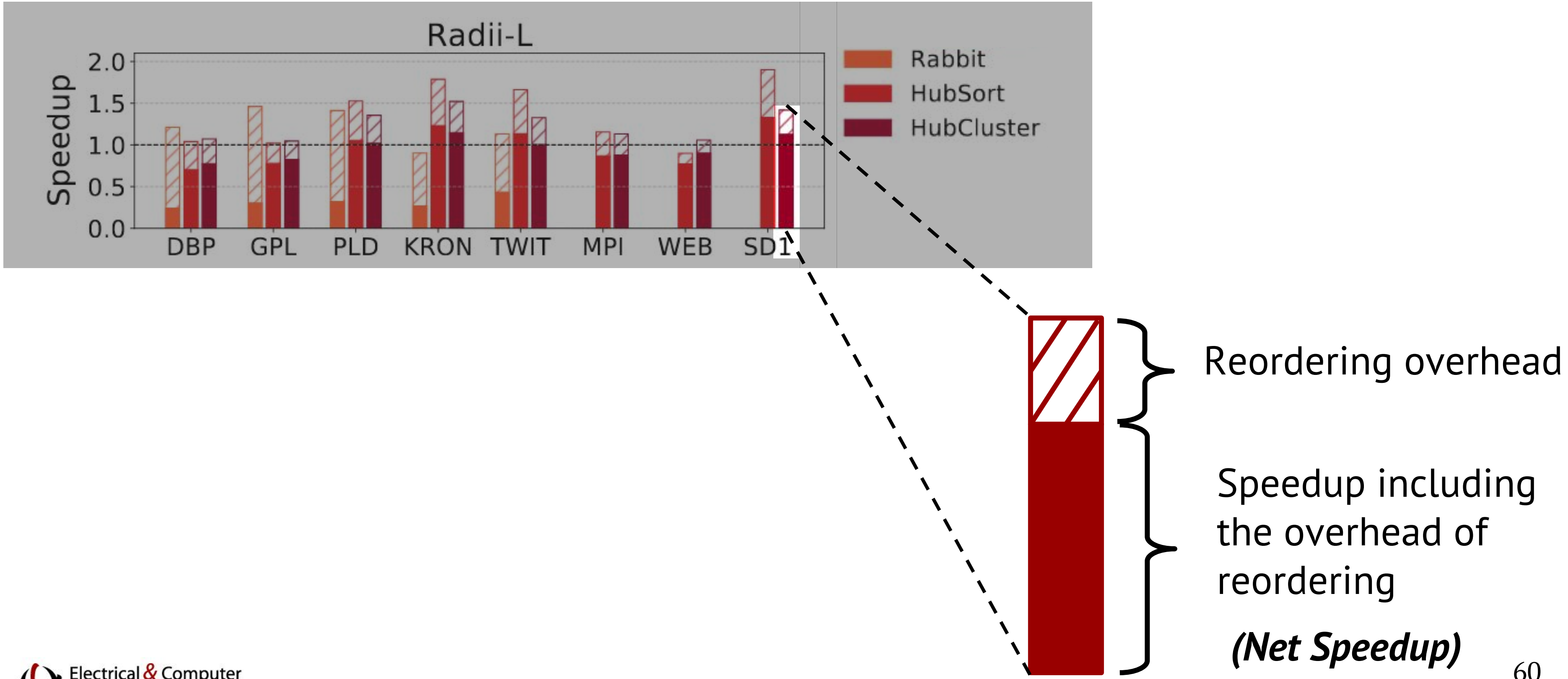
Legend for Results



Speedup including the overhead of reordering

(Net Speedup)

Legend for Results



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

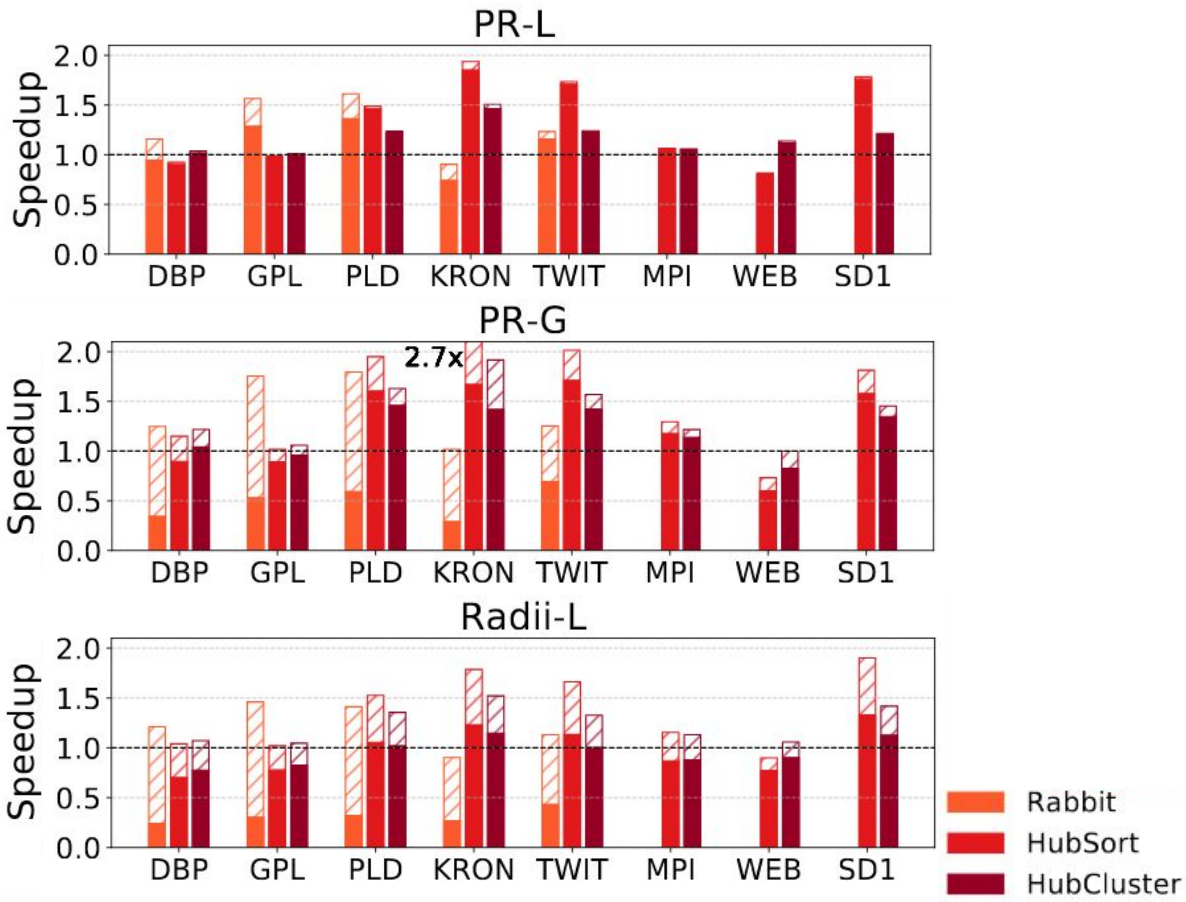
15 Applications →5 Categories

- **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

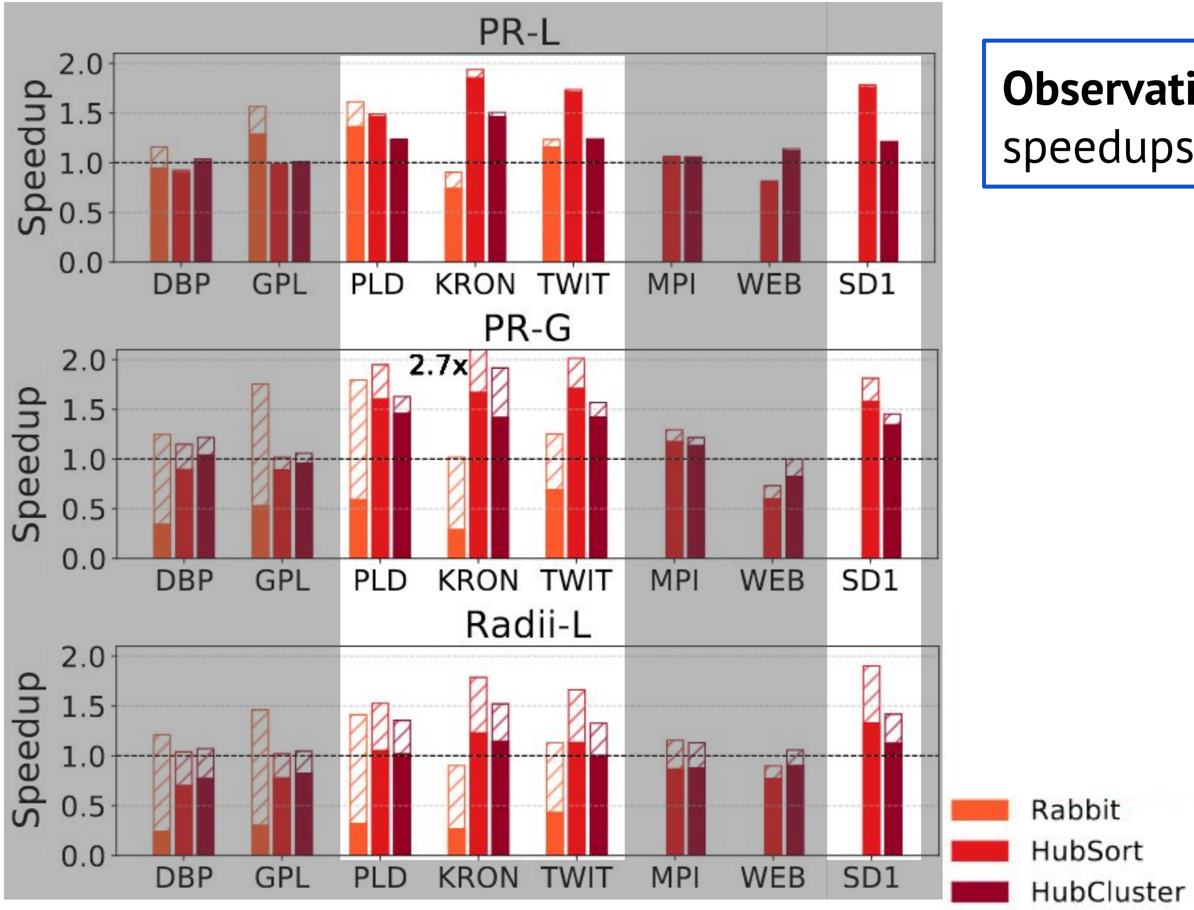
Category I - Applications Processing a Large Fraction Of Edges

- PageRank (Ligra & Gap)
- Graph Radii Estimation (Ligra)

Category I - Applications Processing a Large Fraction Of Edges

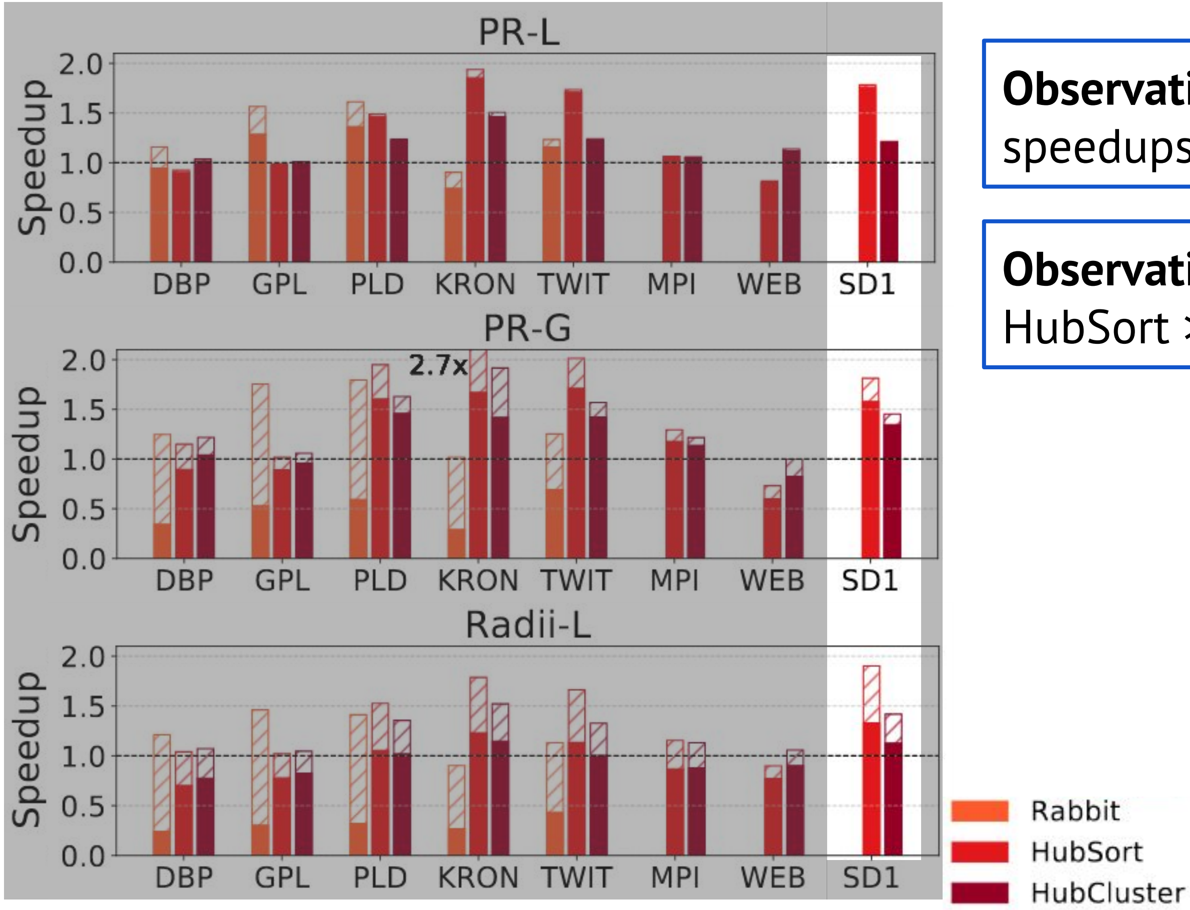


Category I - Applications Processing a Large Fraction Of Edges



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

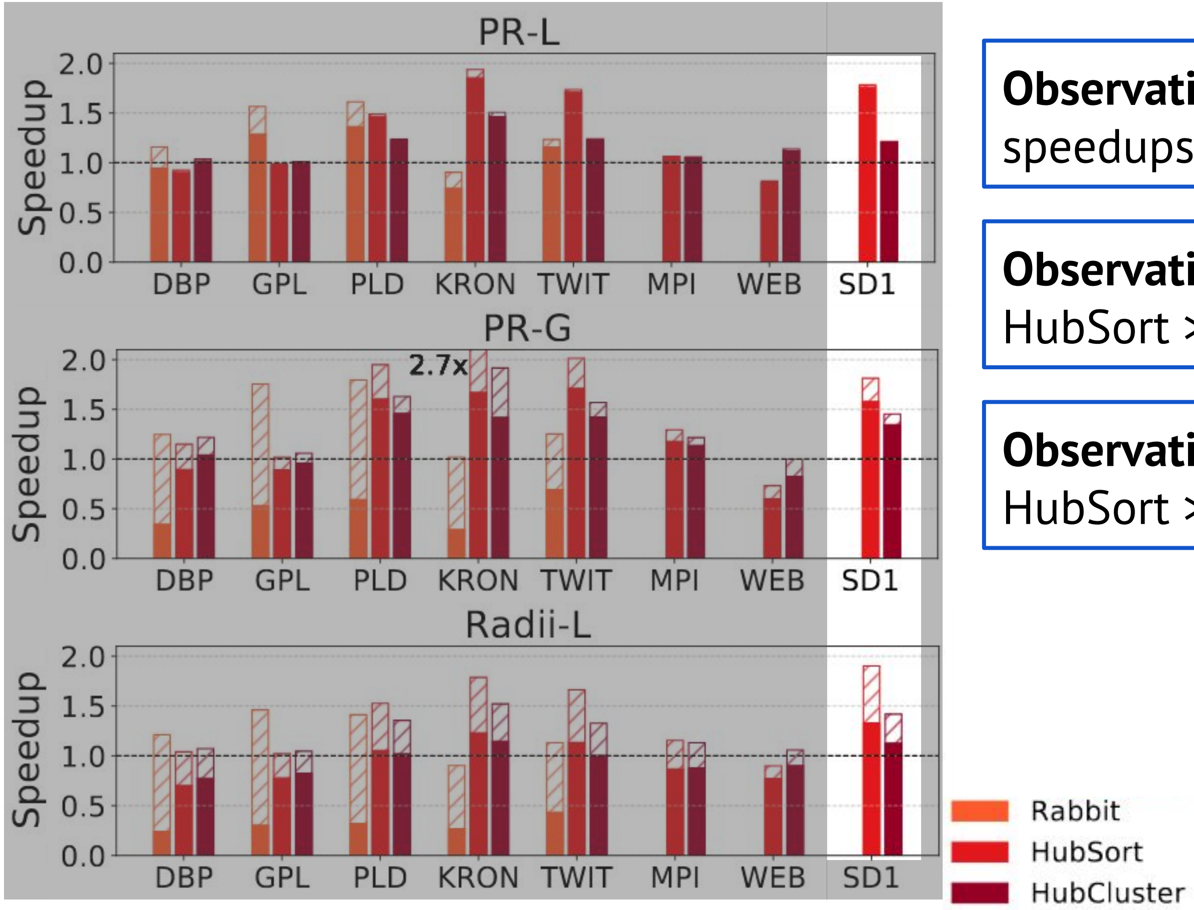
Category I - Applications Processing a Large Fraction Of Edges



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

Observation 2: Maximum speedups from HubSort > HubCluster

Category I - Applications Processing a Large Fraction Of Edges

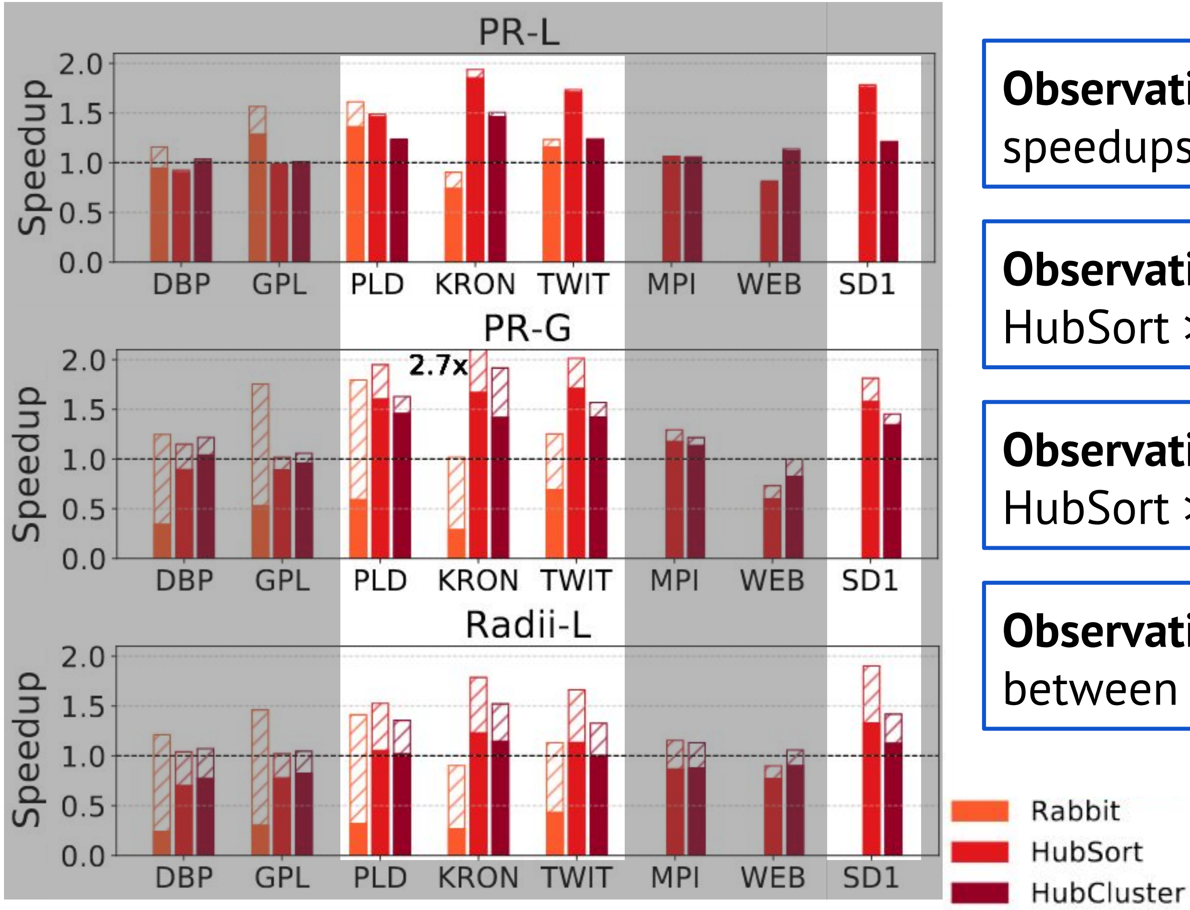


Observation 1: LWR provides ***end-to-end*** speedups in some cases

Observation 2: Maximum speedups from HubSort > HubCluster

Observation 3: Reordering Overhead is HubSort > HubCluster

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

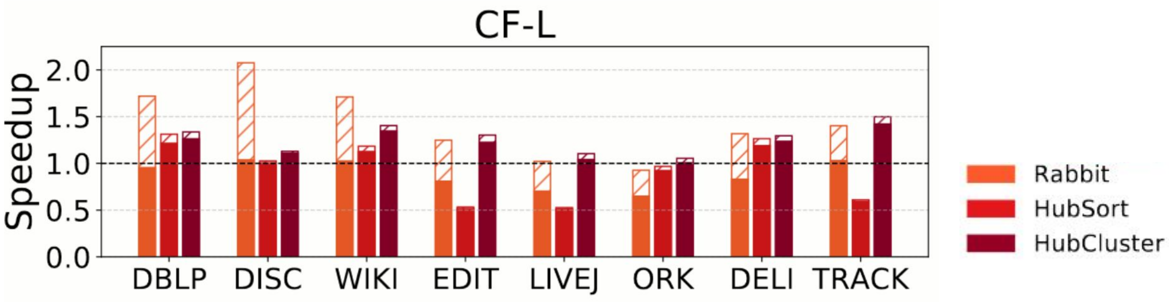
Observation 3: Reordering Overhead is HubSort > HubCluster

Observation 4: HubSort strikes a balance between effectiveness and overhead

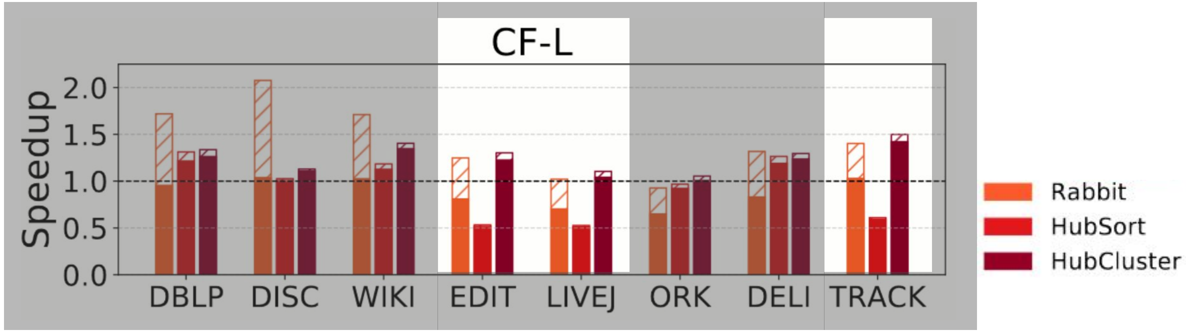
Category II - Executions On Symmetric Bipartite Graphs

- Collaborative Filtering (Ligra)

Category II - Executions On Symmetric Bipartite Graphs

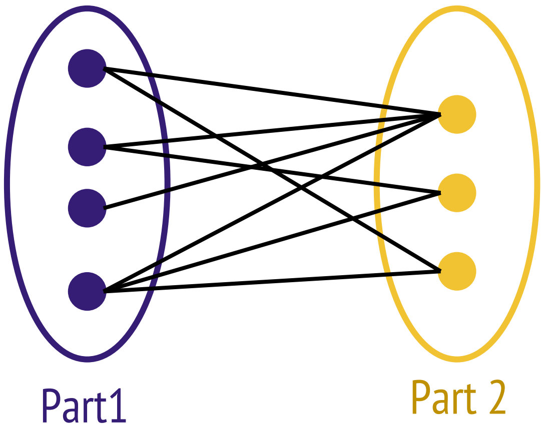


Category II - Executions On Symmetric Bipartite Graphs

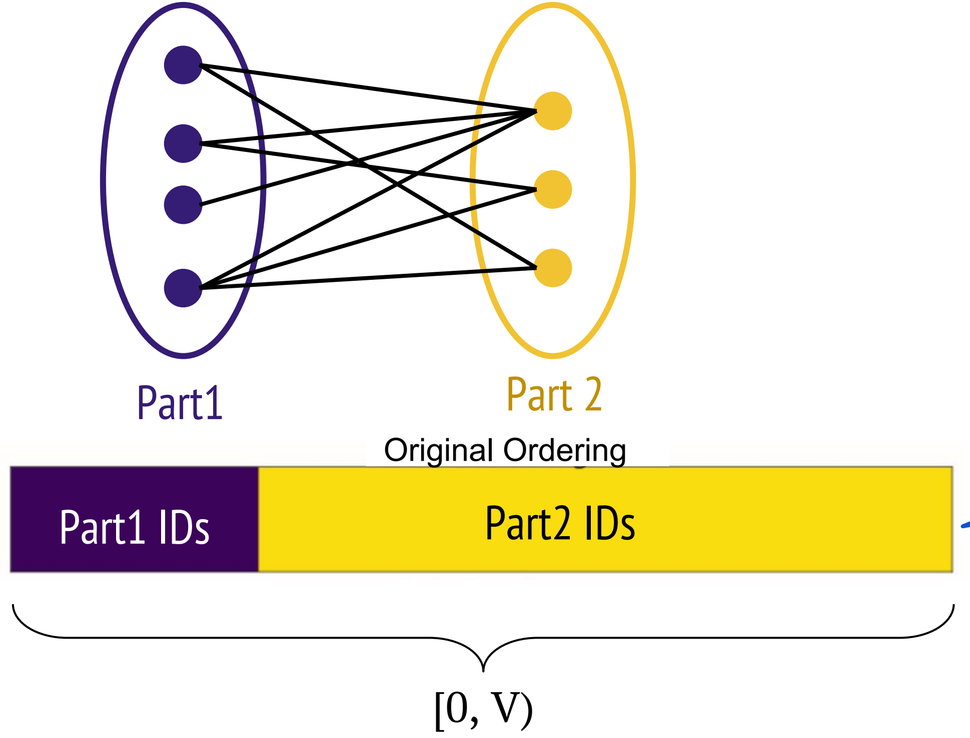


Surprising trend: HubSort causes net slowdowns

Category II - Reason For Slowdown With HubSort

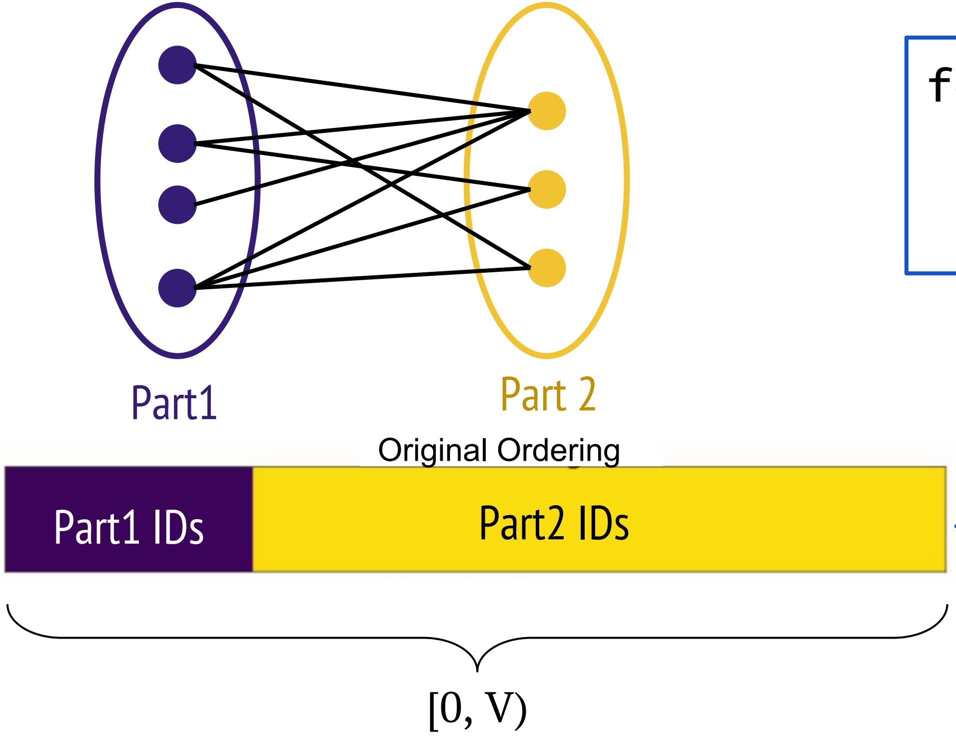


Category II - Reason For Slowdown With HubSort



Heatmap showing the part that each vertex belongs to

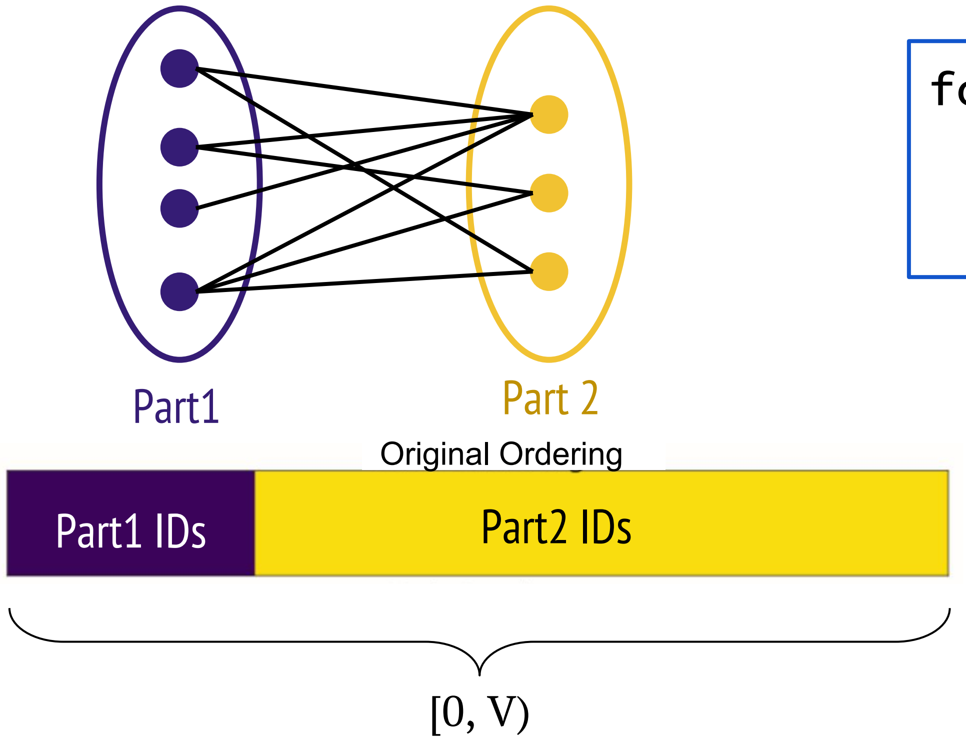
Category II - Reason For Slowdown With HubSort




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

Heatmap showing the part that each vertex belongs to

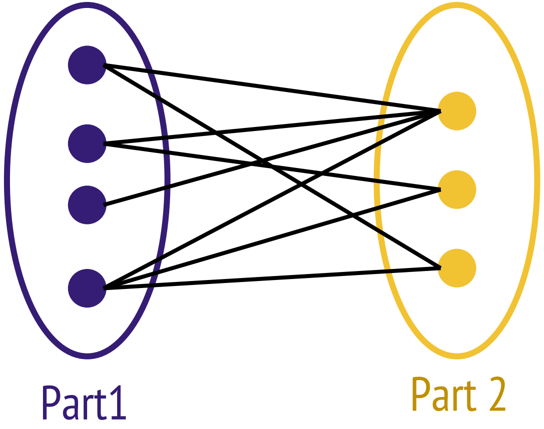
Category II - Reason For Slowdown With HubSort



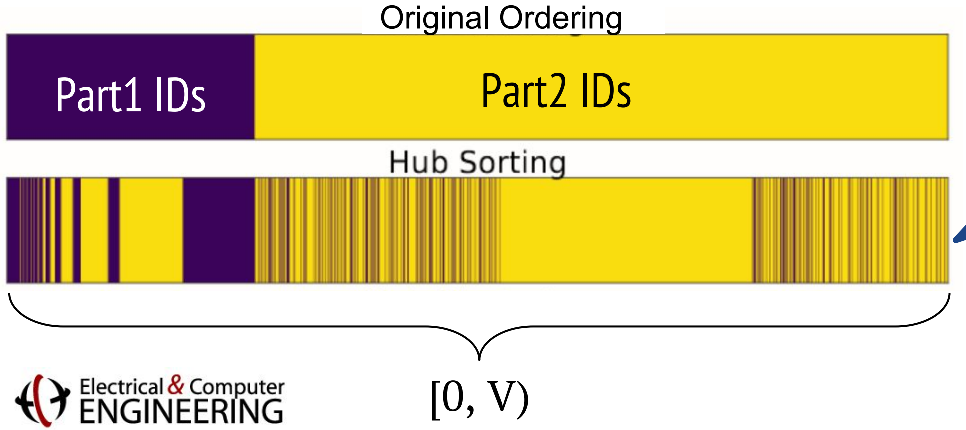

```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u], …)
```

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

Category II - Reason For Slowdown With HubSort

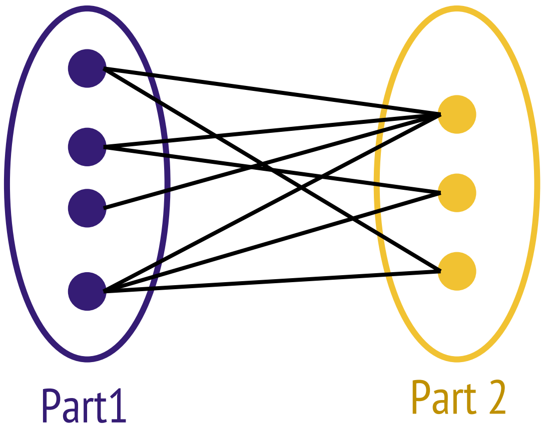



```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u], …)
```

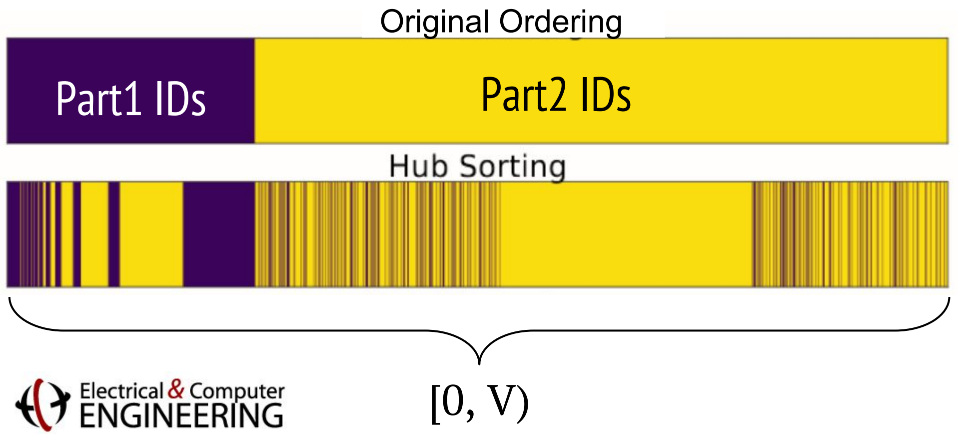


Vertices from different parts assigned consecutive IDs
↓
Increased range of irregular accesses to vtxData

Category II - Reason For Slowdown With HubSort




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

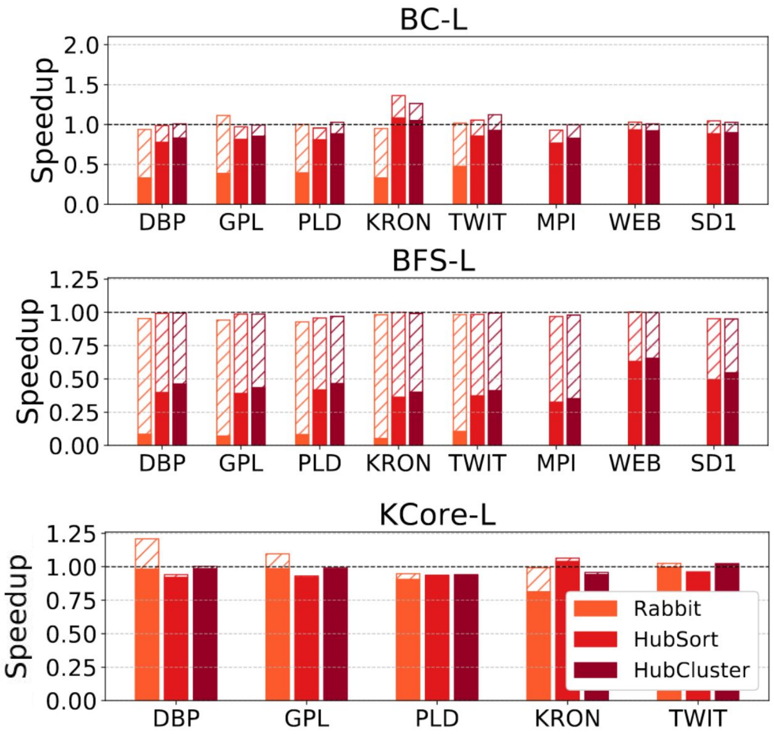


Opportunity: Bipartiteness-aware Graph Reordering Techniques

Category III - Applications Processing a Small Fraction of Edges

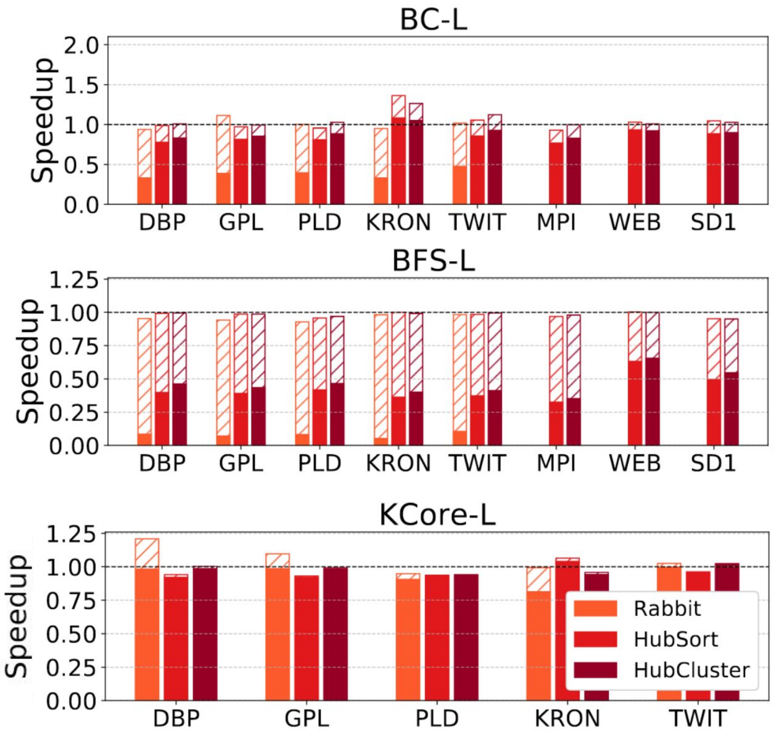
- Betweenness Centrality
- BFS
- K-Core Decomposition

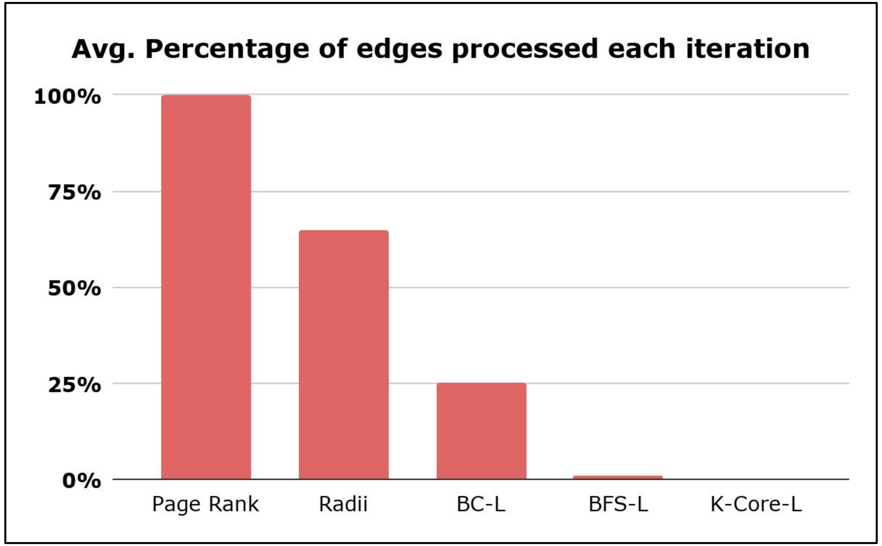
Category III - Applications Processing a Small Fraction of Edges



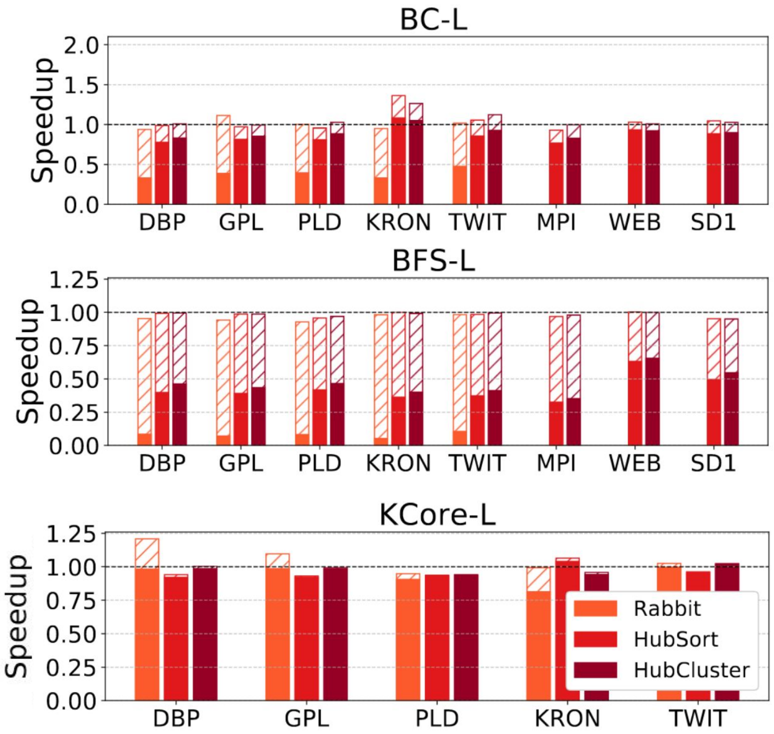
Low speedup even without Reordering overheads

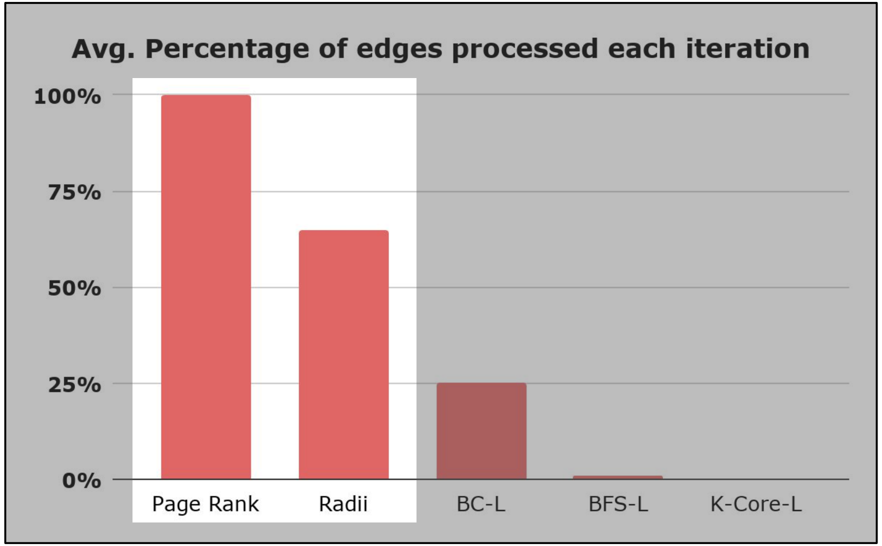
Category III - Applications Processing a Small Fraction of Edges



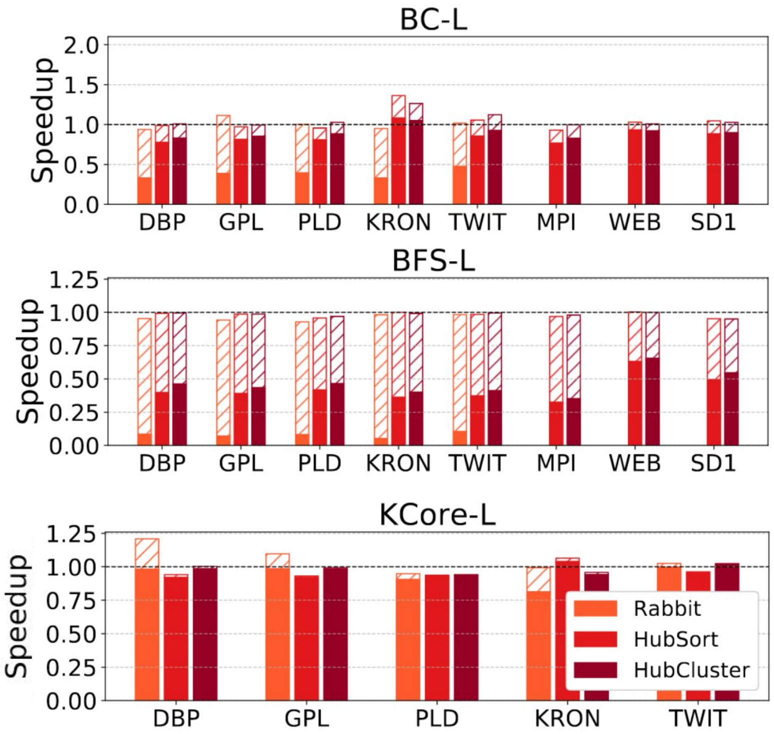


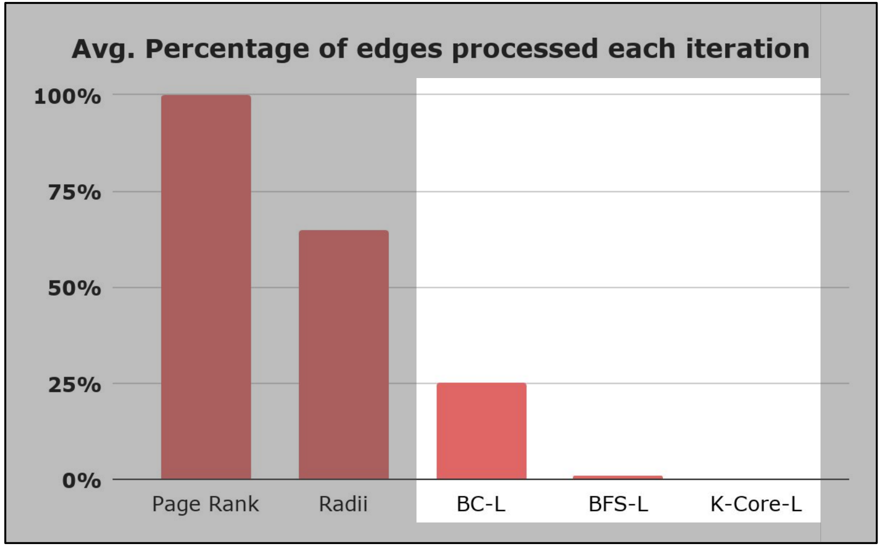
Category III - Applications Processing a Small Fraction of Edges



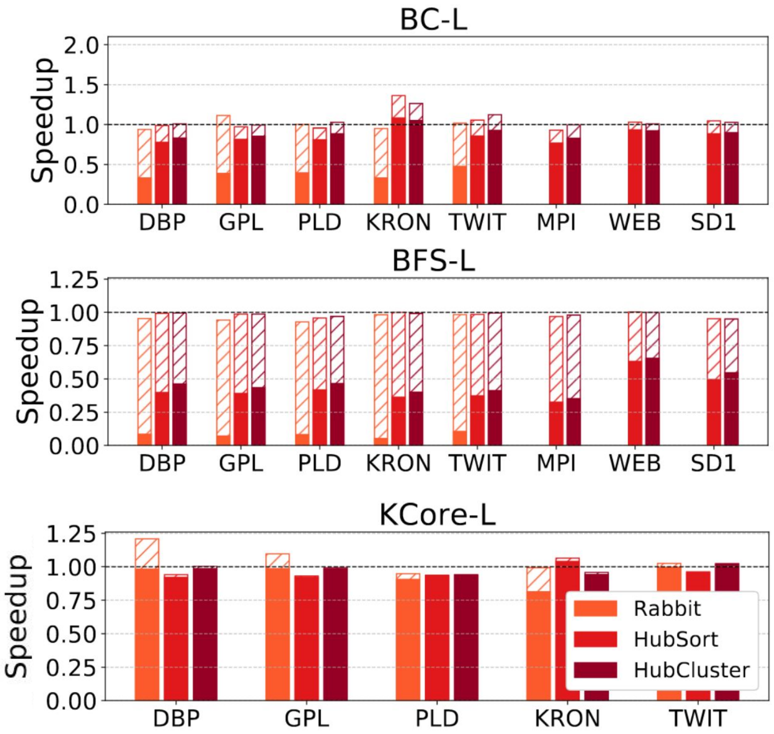


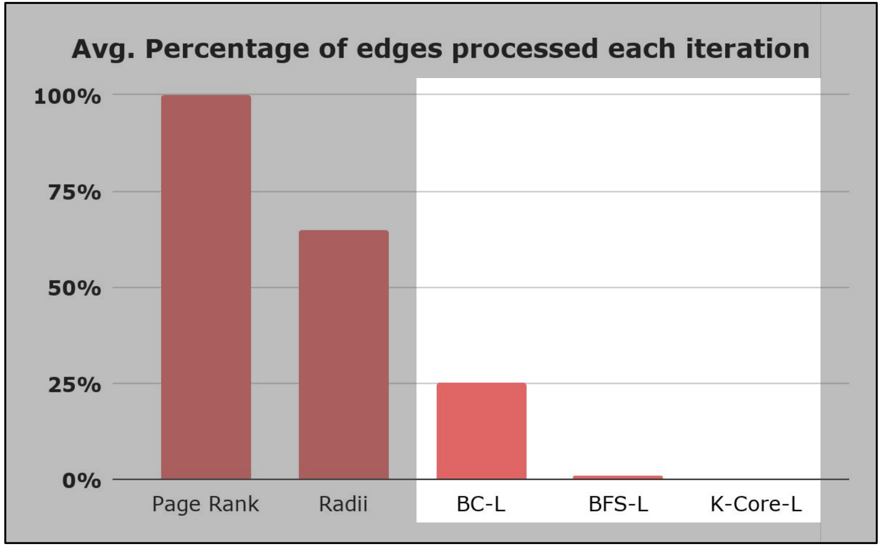
Category III - Applications Processing a Small Fraction of Edges





Category III - Applications Processing a Small Fraction of Edges





Limited reuse in vtxData accesses

↓

Lower headroom for reordering

Outline

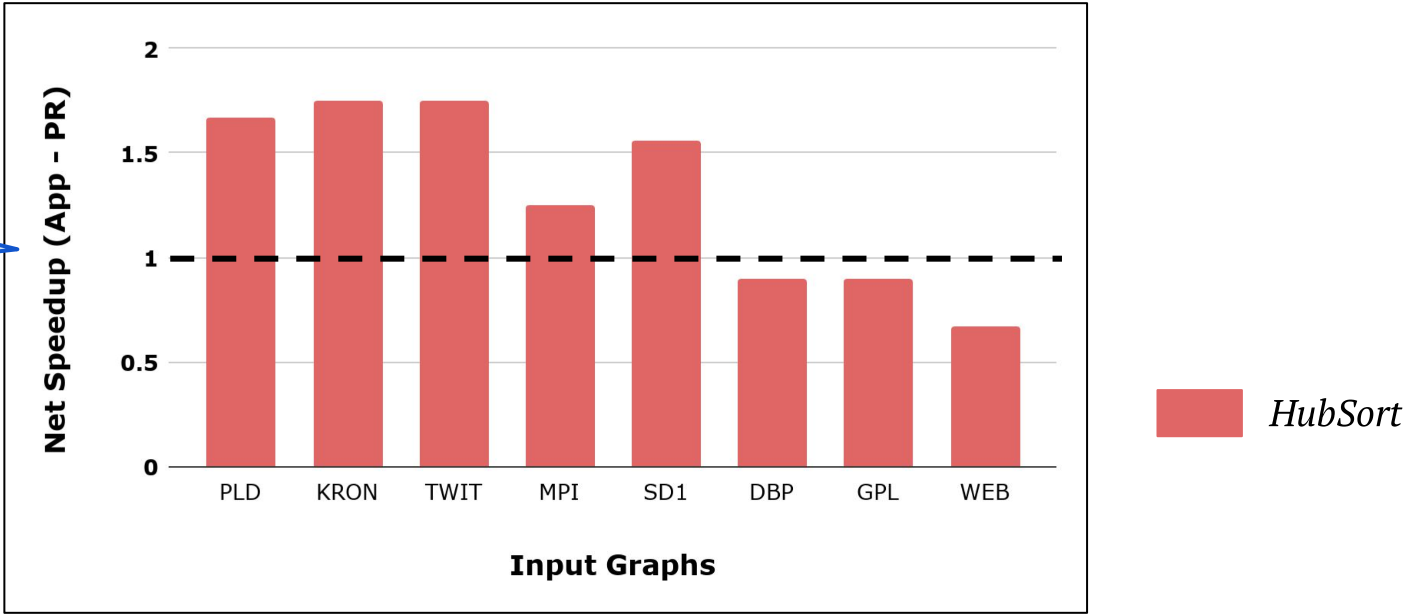
- ❖ Poor Locality of Graph Processing Applications ✔
- ❖ Improving locality through Graph Reordering ✔
- ❖ Graph Reordering Challenge - *Application and Input-dependent Speedups* ✔
- ❖ **When is Graph Reordering an Optimization?**
  - ➢ Characterization Space ✔
  - ➢ Which Applications benefit from Reordering?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

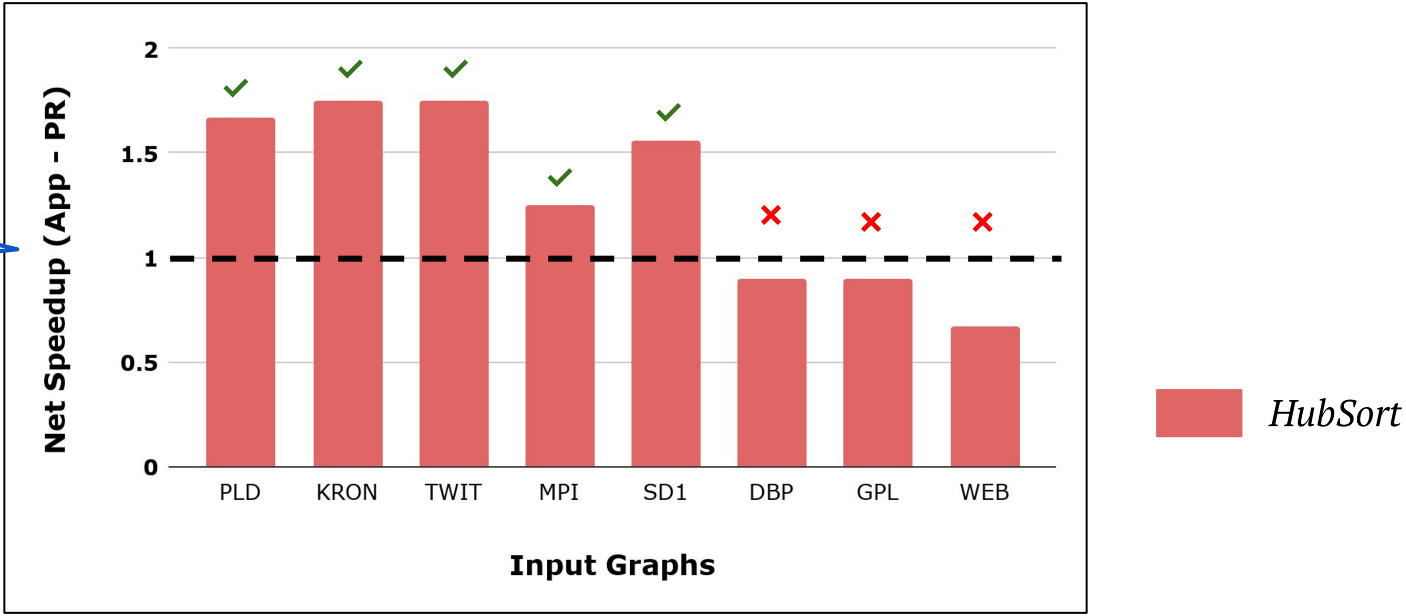
Speedup From HubSorting Varies Across Inputs

Category I Application



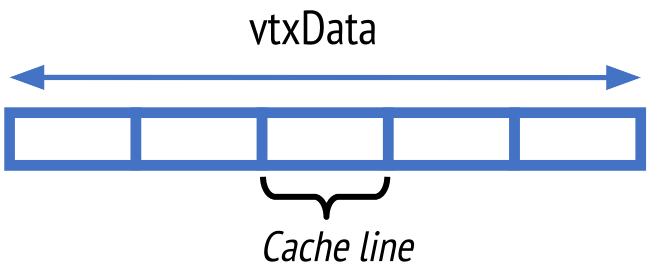
Speedup From HubSorting Varies Across Inputs

Category I Application

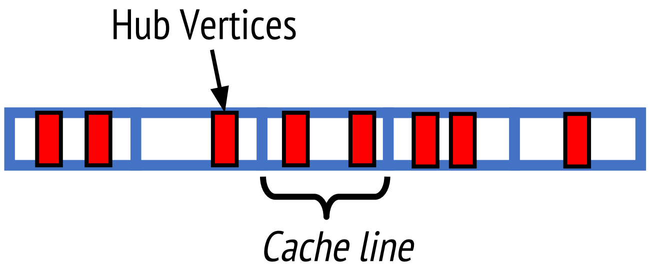


Need to ***predict speedup*** from HubSorting
AND
selectively perform HubSorting

Understanding Performance Improvement From HubSorting

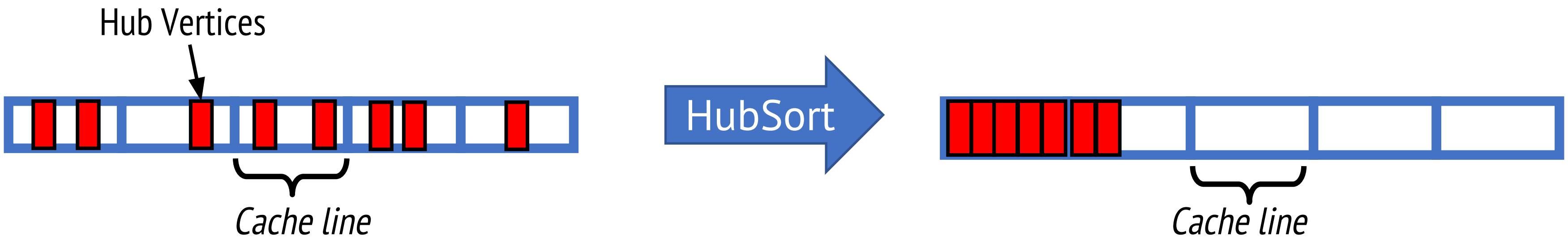


Understanding Performance Improvement From HubSorting



Layout of hubs in original ordering

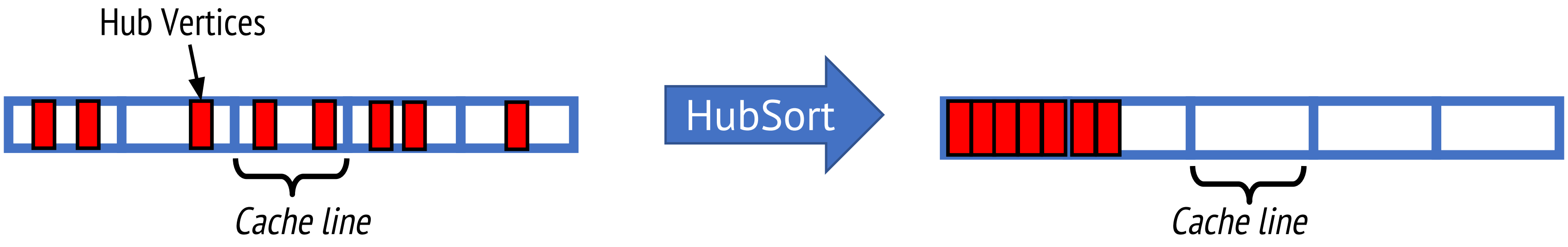
Understanding Performance Improvement From HubSorting



Layout of hubs in original ordering

Layout of hubs after reordering vertices

Understanding Performance Improvement From HubSorting



Layout of hubs in original ordering

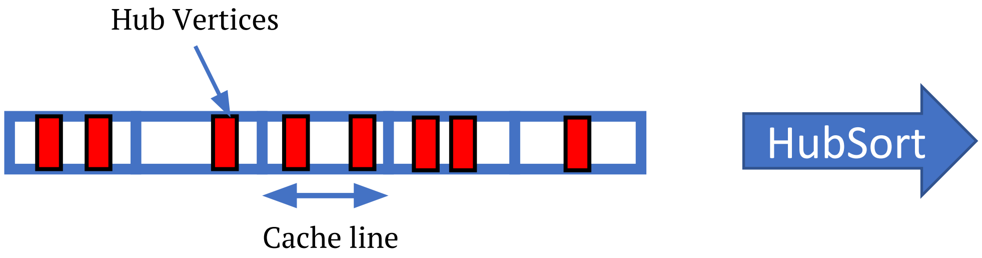
Layout of hubs after reordering vertices

HubSorting will be most effective for Graphs with:

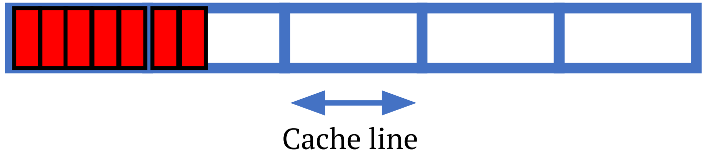
- ❖ **Property #1:** Skew in the degree-distribution (*Presence of Hubs*)
- ❖ **Property #2:** Sparsely distributed hub vertices (*Quality of original ordering*)

Packing Factor - A Measure of Hub Density

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



Layout of the original ordering of vertices

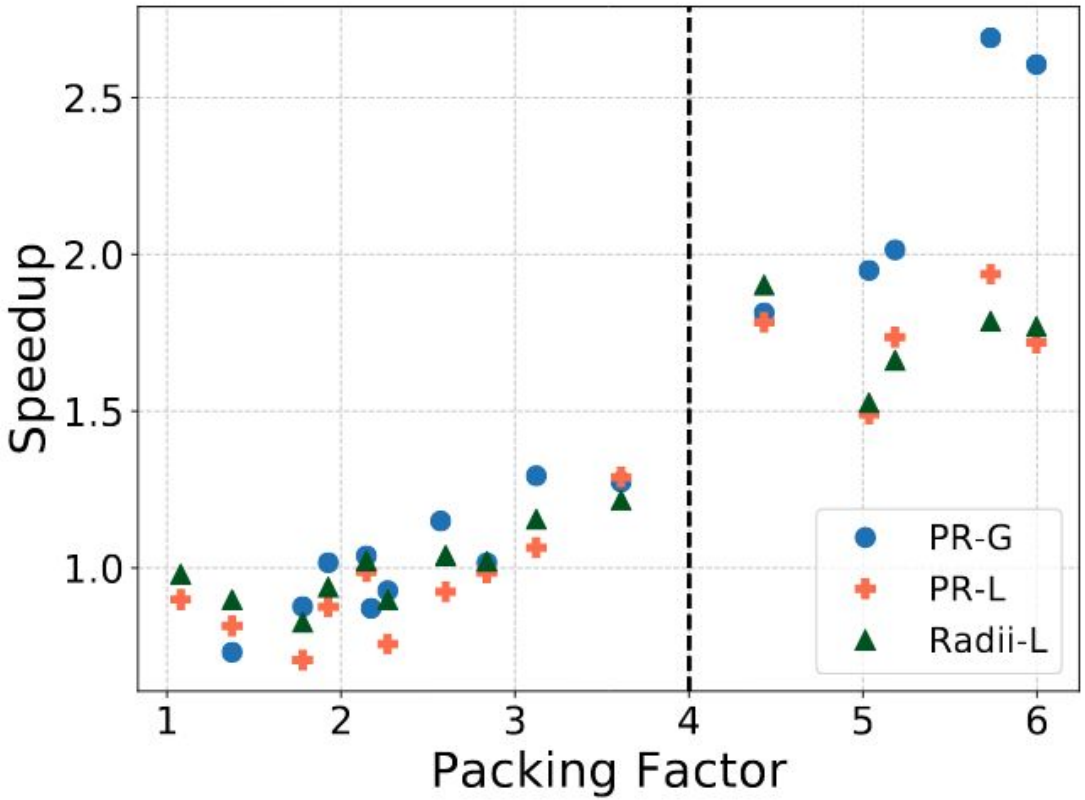


Layout after reordering vertices by HubSorting

$$Packing\ Factor = \frac{5}{2} = 2.5$$

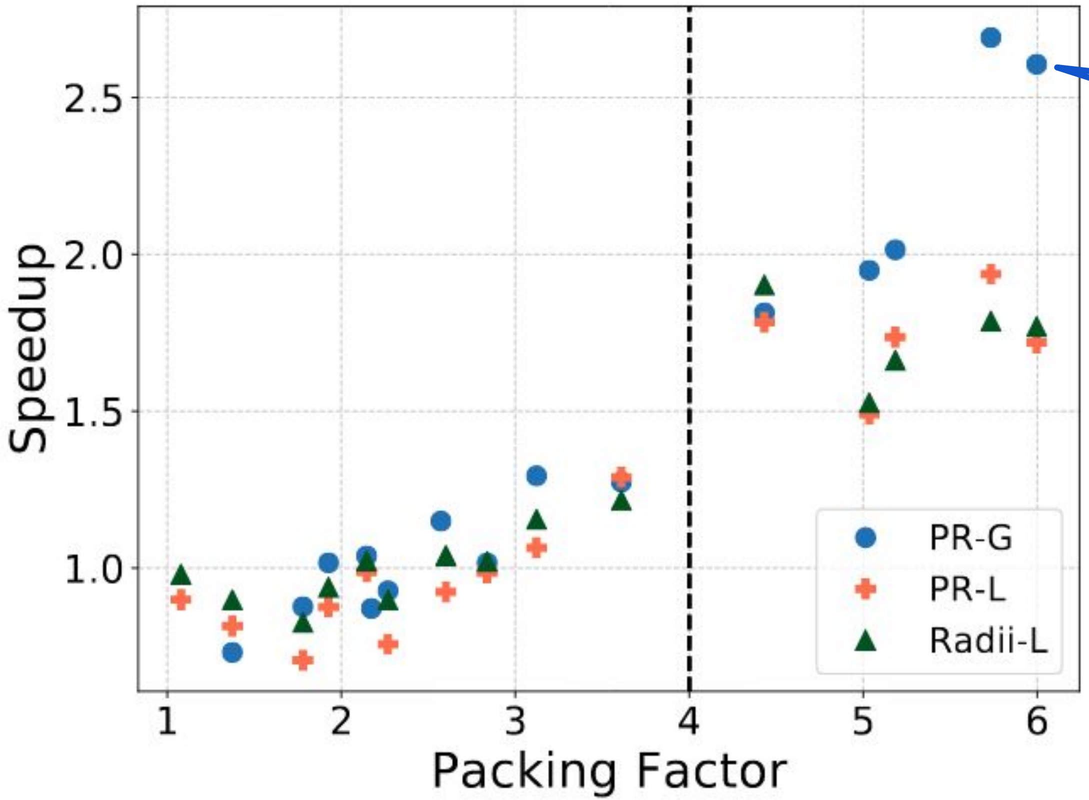
Packing Factor Can Predict Speedup From HubSorting

Pearson Correlation = 0.92



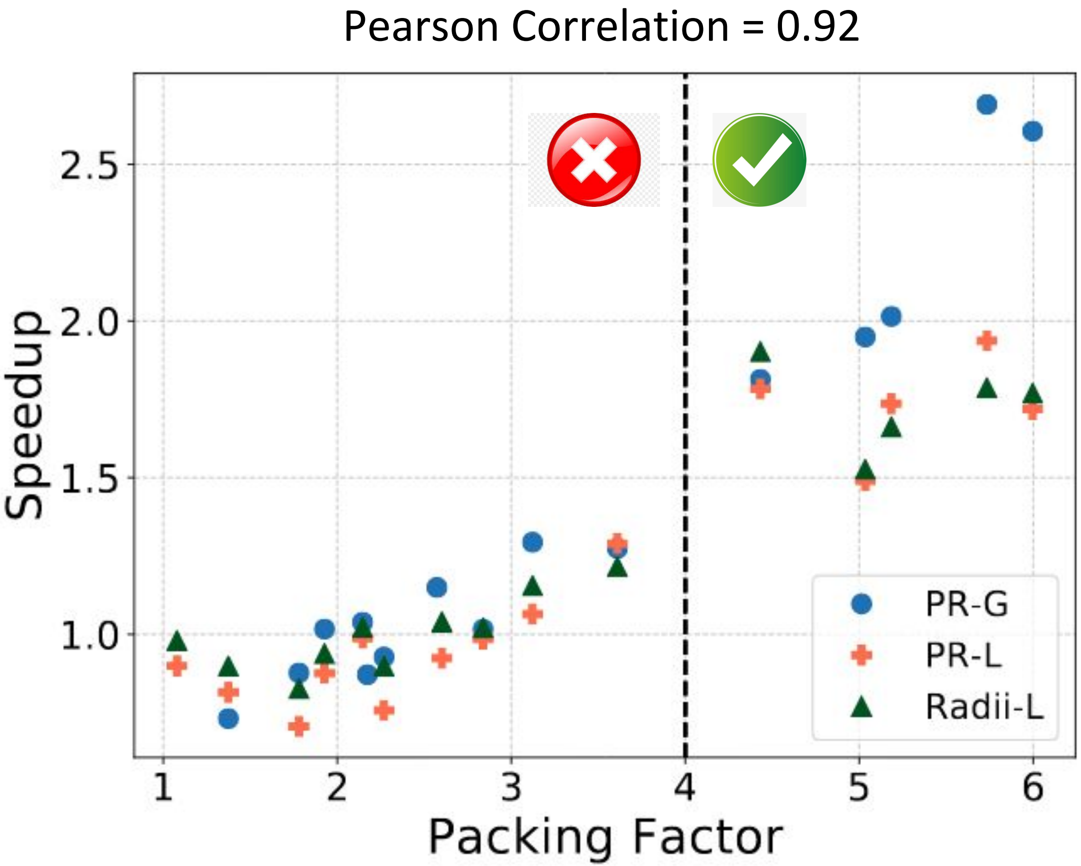
Packing Factor Can Predict Speedup From HubSorting

Pearson Correlation = 0.92



Speedup from HubSorting for a given pair of (Application, Input Graph)

Packing Factor Can Predict Speedup From HubSorting



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

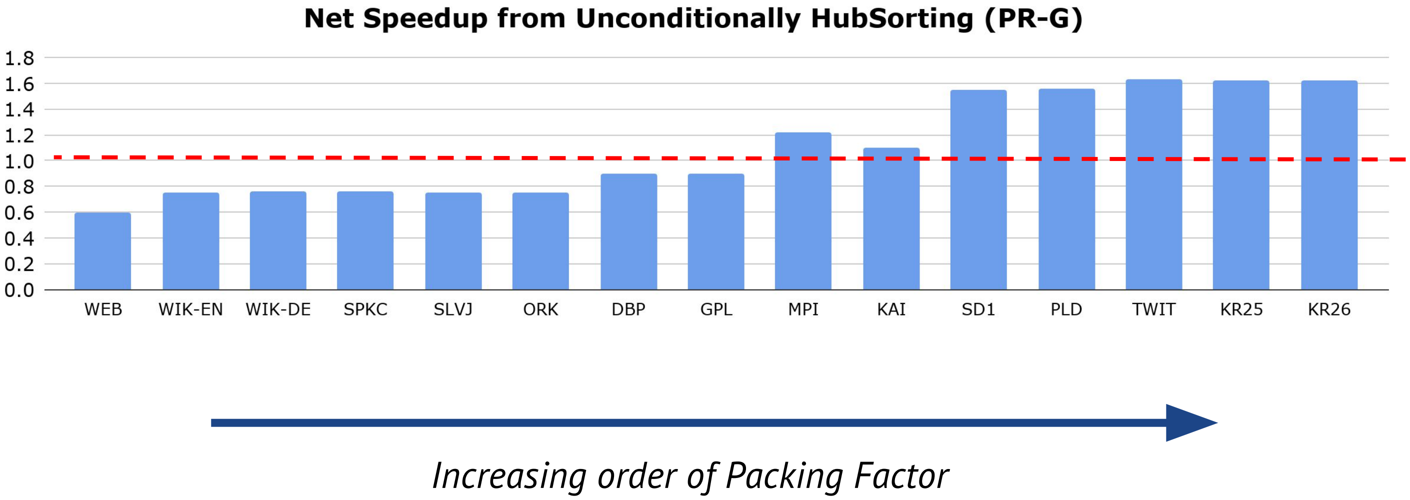
- Poor Locality of Graph Processing Applications ✔
- Improving locality through Graph Reordering ✔
- Graph Reordering Challenge - *Application and Input-dependent Speedups* ✔
- When is Graph Reordering an Optimization?
  - Characterization Space ✔
  - Which Applications benefit from Reordering? ✔
  -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

Selective Graph Reordering

```
G`= HubSort(G)

Process(G`)
```

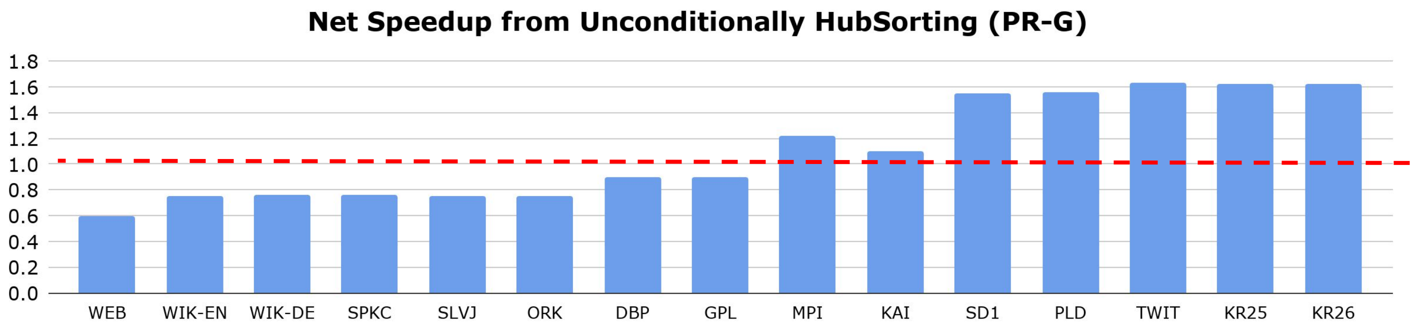
Selective Graph Reordering




```
G`= HubSort(G)

Process(G`)
```

Selective Graph Reordering

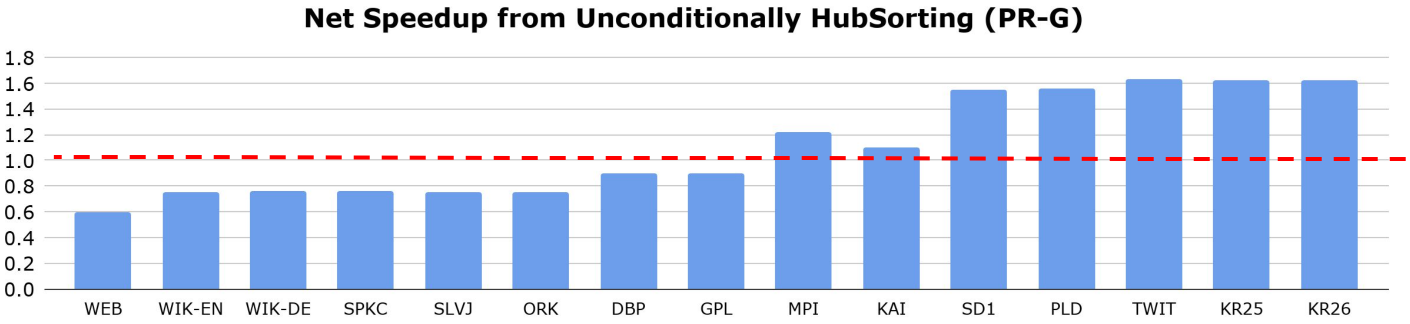



```
G`= HubSort(G)

Process(G`)
```

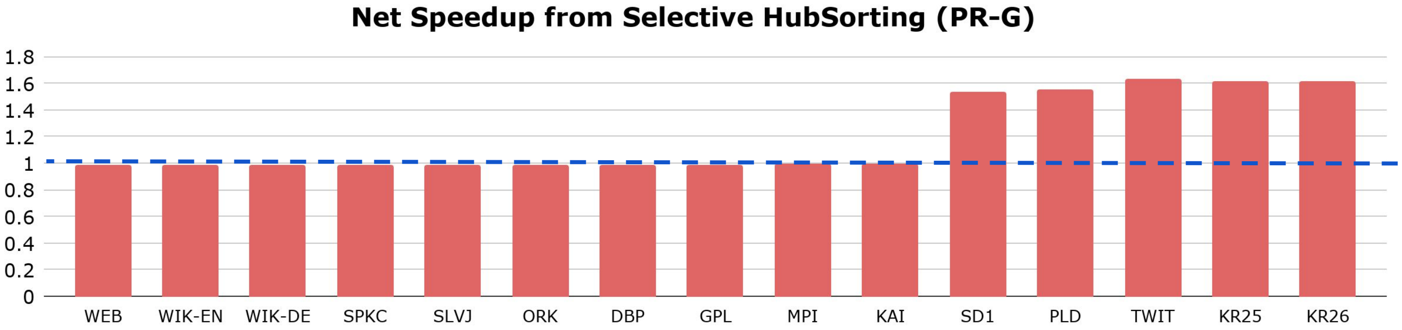
```
PF = ComputePF(G)
if (PF > 4):
  G` = HubSort(G)
  Process(G`)
else:
  Process(G)
```

Selective Graph Reordering



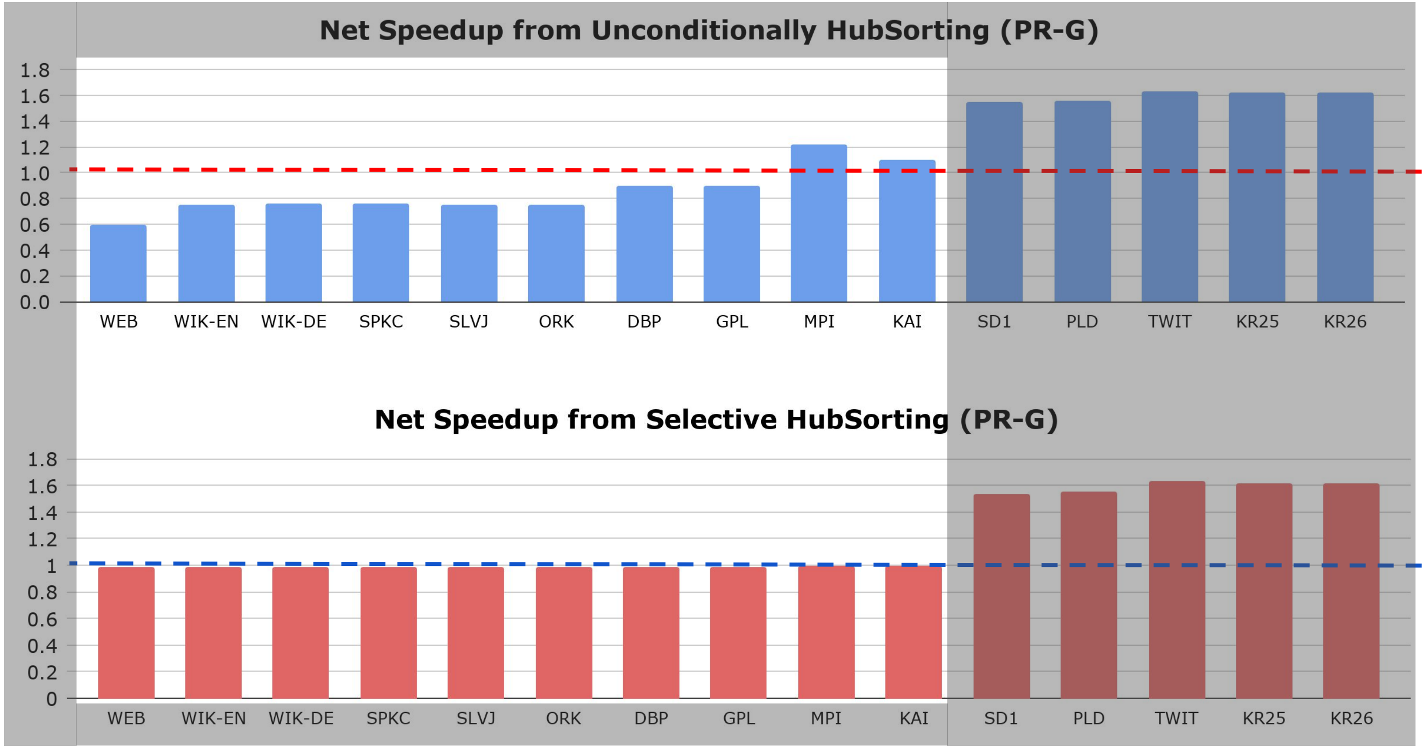

```
G`= HubSort(G)

Process(G`)
```



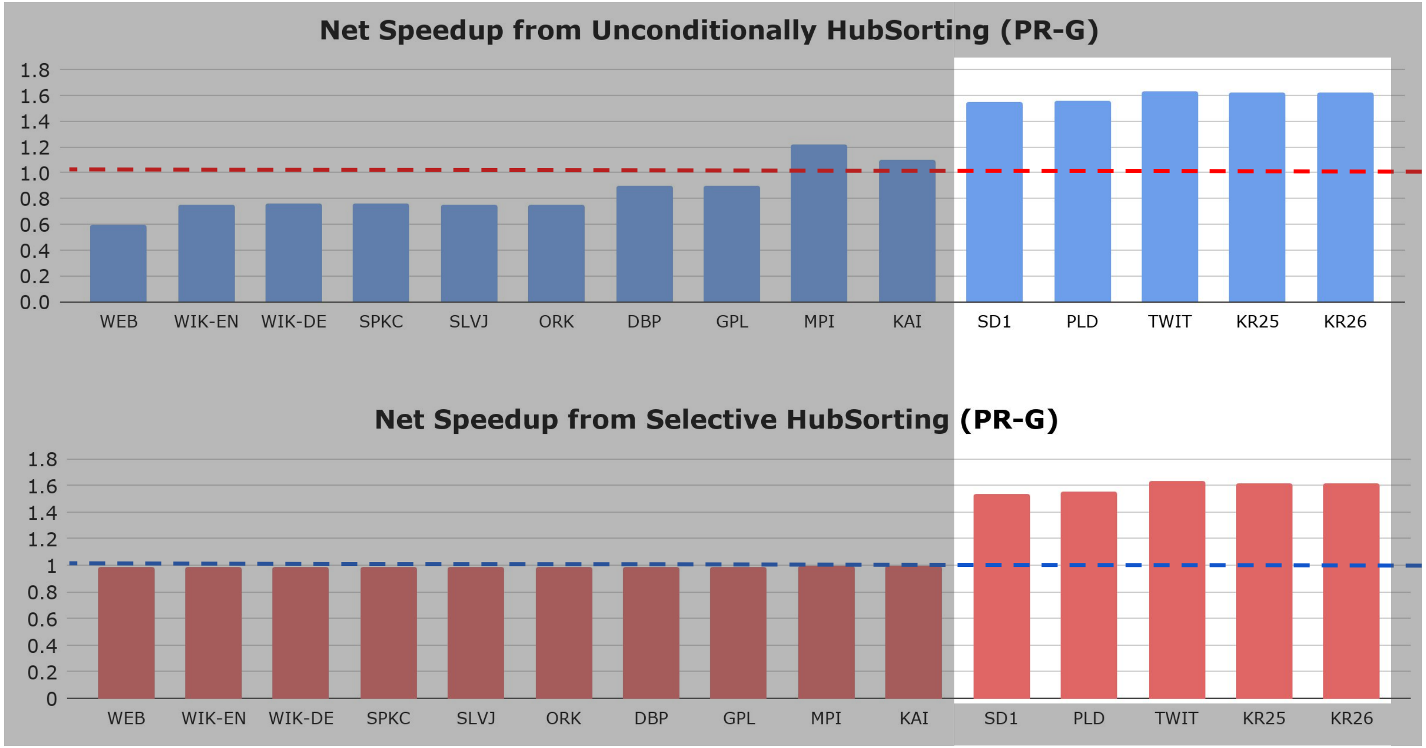

```
PF = ComputePF(G)
if (PF > 4):
  G` = HubSort(G)
  Process(G`)
else:
  Process(G)
```

Selective Graph Reordering



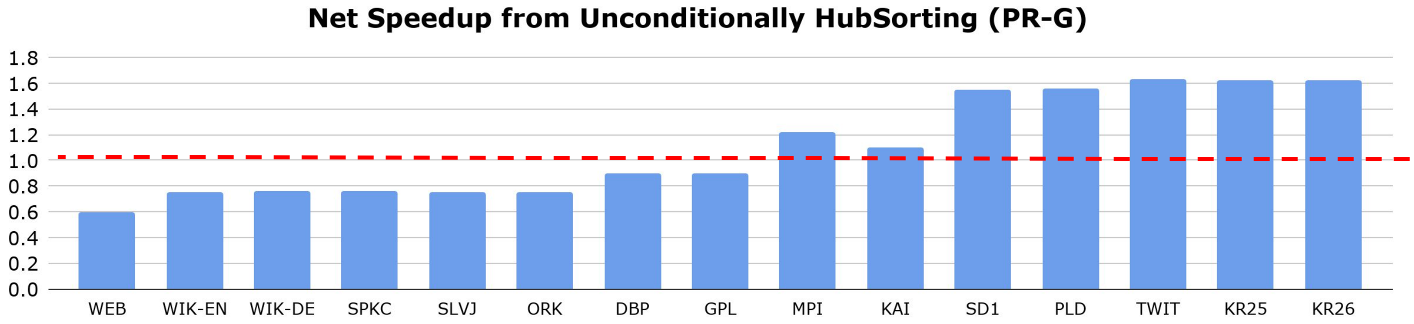
Selective Reordering avoids slowdowns

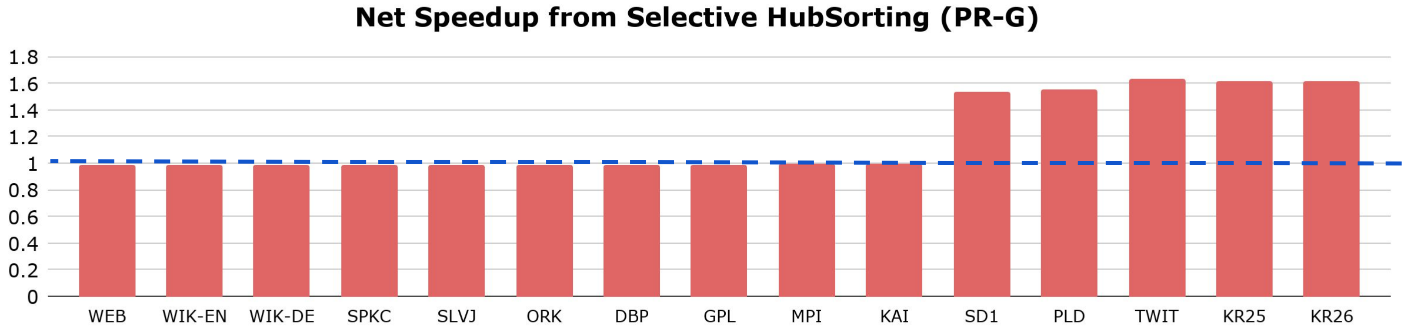
Selective Graph Reordering



Computing Packing Factor does not degrade performance

Selective Graph Reordering





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

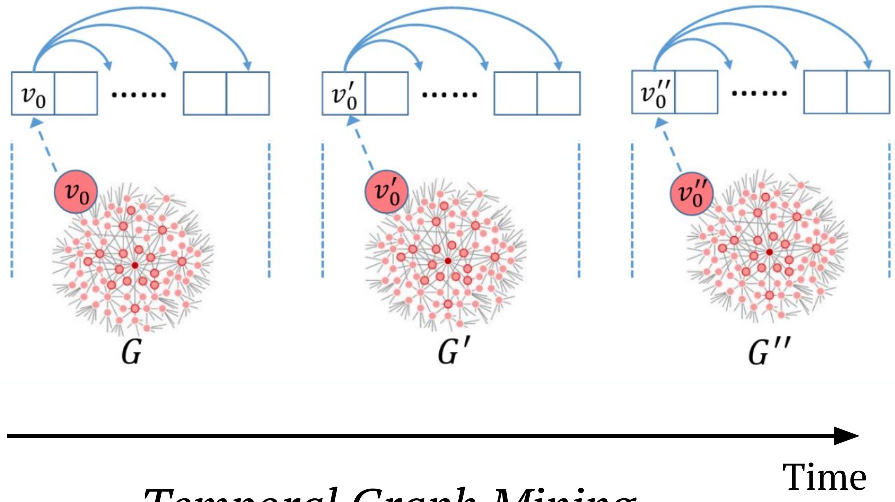
Vignesh Balaji Brandon Lucia

Electrical & Computer ENGINEERING

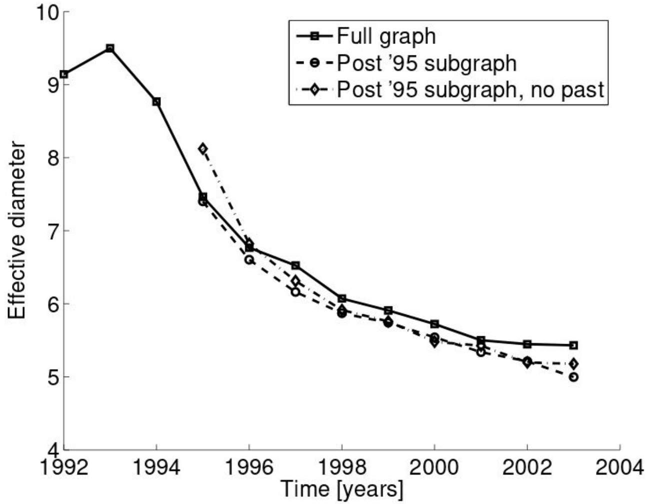
Carnegie Mellon

Backup Slides

Use-cases Where Reordering Overhead Cannot be Amortized



Temporal Graph Mining



arXiv citation graph

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

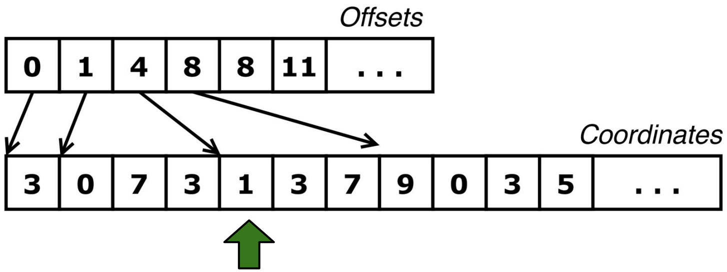
Sophisticated Reordering Techniques Impose *High* Overhead

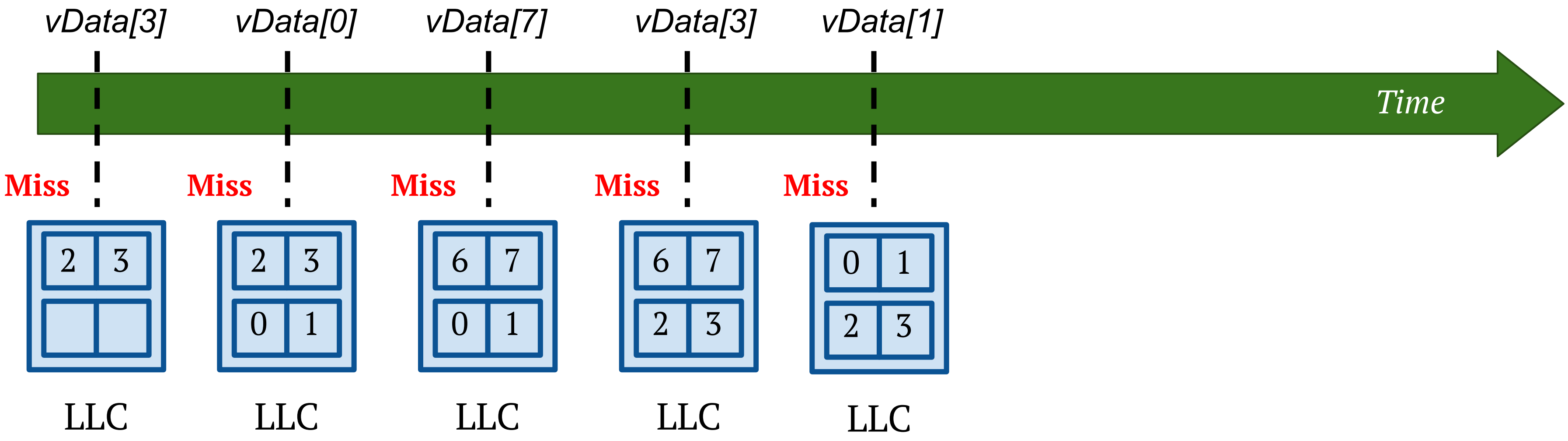
	gplus	web	pld-arc	twitter	kron26
Run Time (baseline)	6.40s	7.84s	12.40s	21.3s	12.88s
Run Time (Gorder)	4.48s	7.77s	6.54s	13.09s	5.01s
Overhead (Gorder)	1685.9s	459.8s	7255s	25200s	53234s
#Runs to amortize ovhd	873	6477	1237	3072	6771

Assumption: Reordered graph will be processed multiple times

Irregular Accesses Have Poor Temporal And Spatial Locality

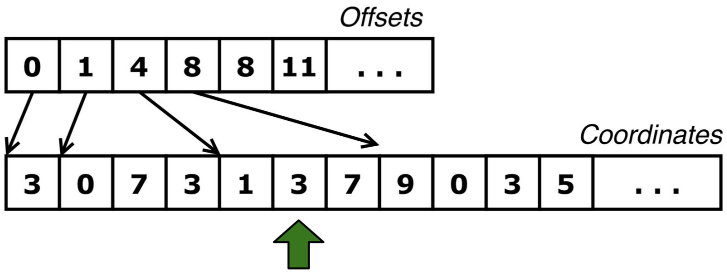
```
for v in G:
  for u in neigh(v):
    process(..., vData[u],...)
```

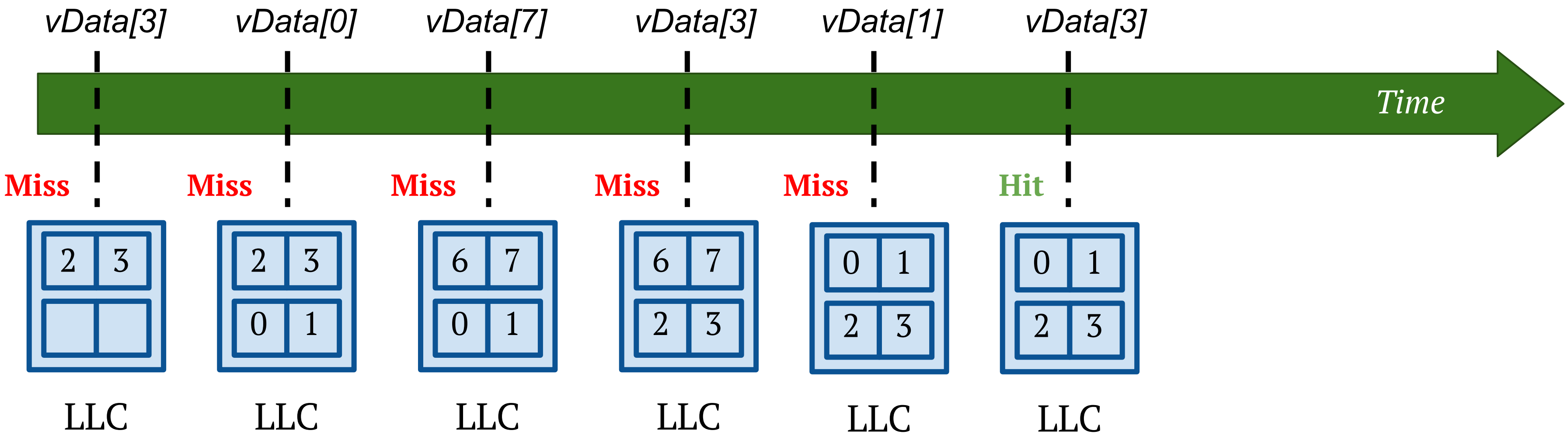




Irregular Accesses Have Poor Temporal And Spatial Locality

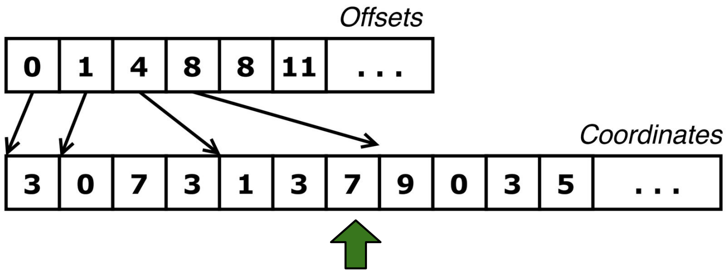
```
for v in G:
  for u in neigh(v):
    process(..., vData[u],...)
```

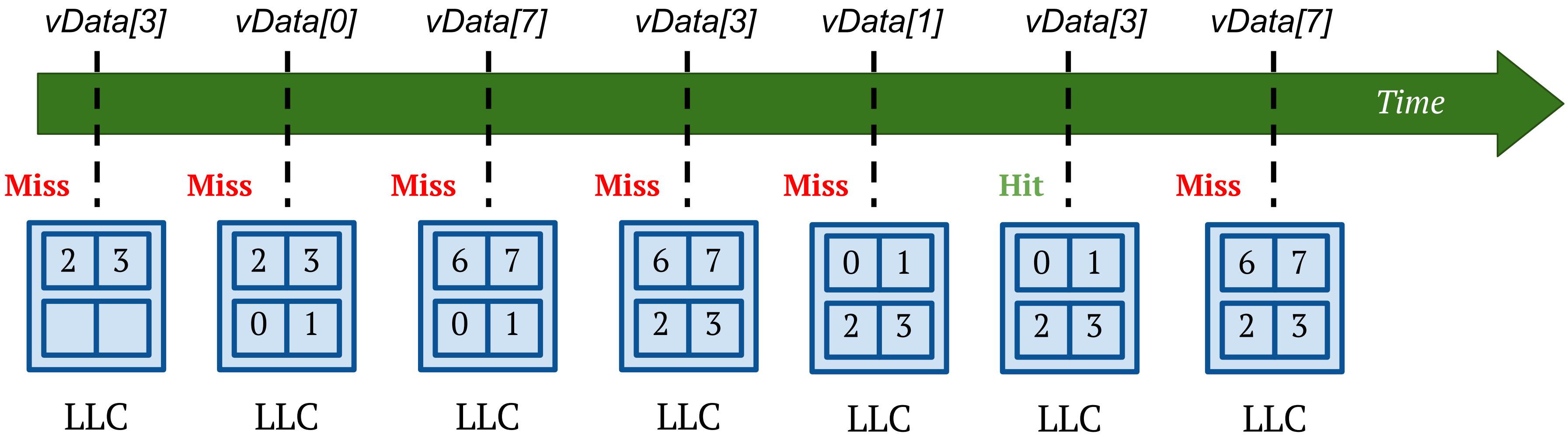




Irregular Accesses Have Poor Temporal And Spatial Locality

```
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

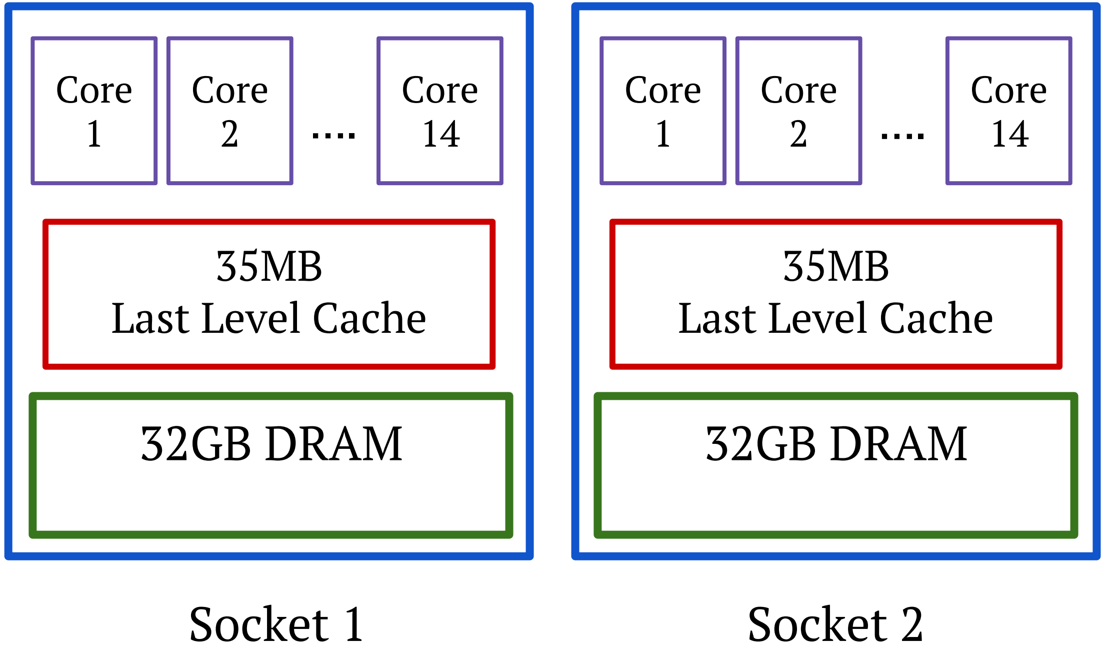
GAP

- 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



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
`vData` **Sz (MB)**	146.16	231.52	343.12	268.4	498.64	420.64	405.12	759.6
CSR Sz (GB)	1.41	3.66	4.96	8.05	11.34	15.02	14.75	15.13

Input Graphs

	DBP	GPL	PLD	KRON	TWIT	MPI	WEB	SD1
$\lvert V \rvert$ (in M)	18.27	28.94	42.89	33.55	61.58	52.58	50.64	94.95
$\lvert E \rvert$ (in B)	0.172	0.462	0.623	1.047	1.468	1.963	1.93	1.937
vData **Sz (MB)**	146.16	231.52	343.12	268.4	498.64	420.64	405.12	759.6
CSR Sz (GB)	1.41	3.66	4.96	8.05	11.34	15.02	14.75	15.13

Irregular working set size >> Aggregate LLC Capacity

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
vData Sz (MB)	146.16	231.52	343.12	268.4	498.64	420.64	405.12	759.6
CSR Sz (GB)	1.41	3.66	4.96	8.05	11.34	15.02	14.75	15.13

Irregular working set size >> Aggregate LLC Capacity

We use the original ordering of Input Graphs

Lightweight Reordering Can Provide End-to-end Speedups

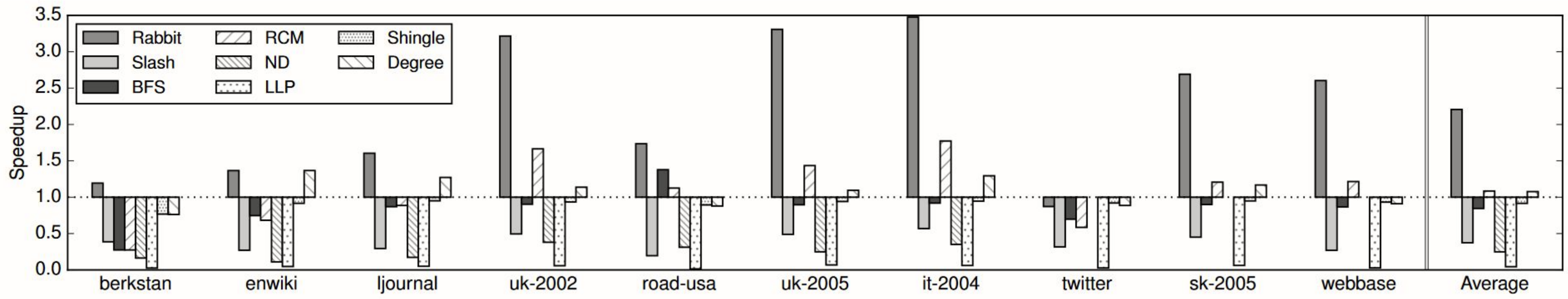
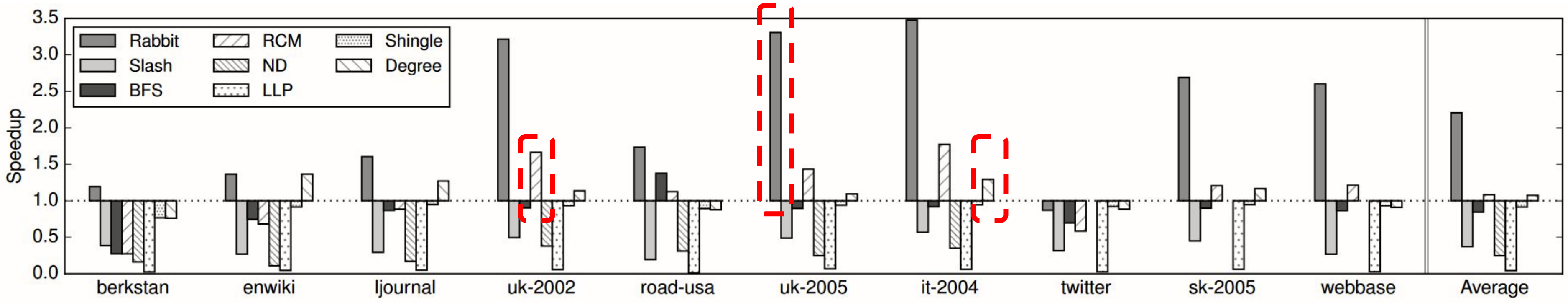


Figure from "Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis" IPDPS 2016

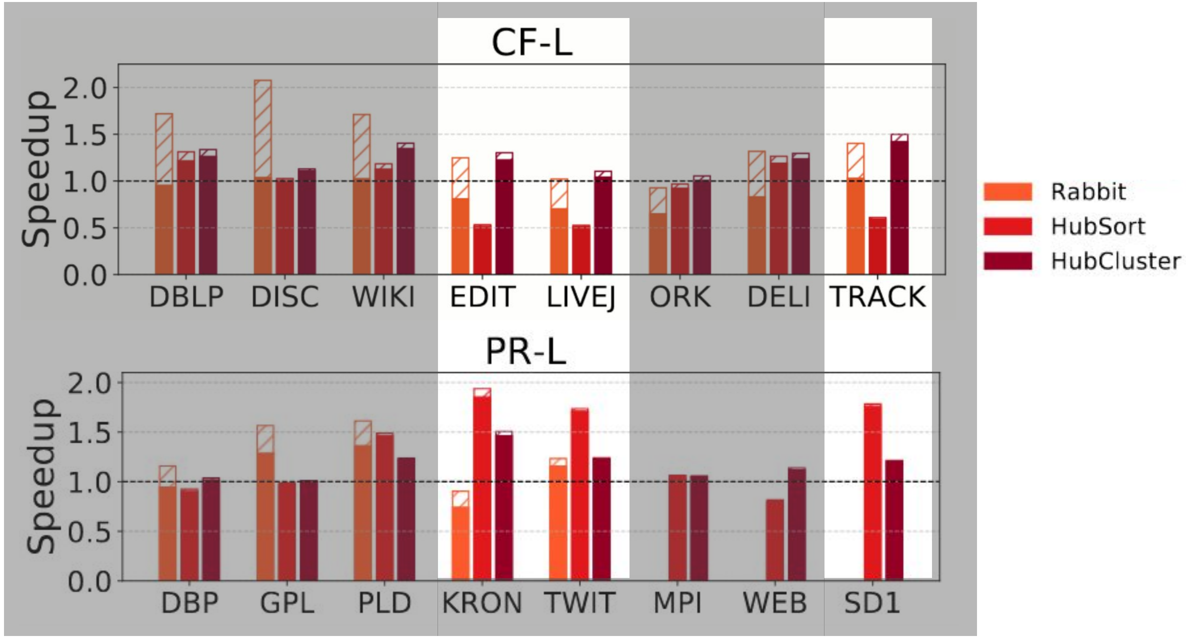
Lightweight Reordering Can Provide End-to-end Speedups



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

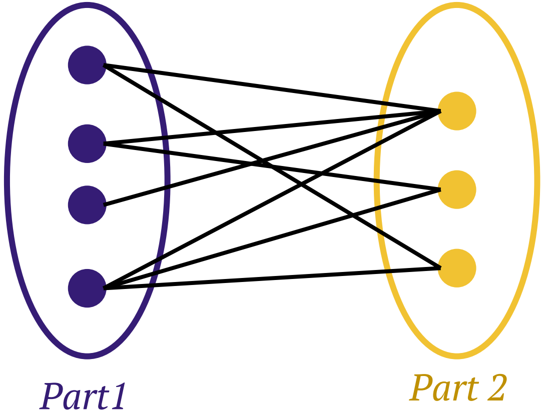
Category II - Executions On Symmetric Bipartite Graphs



Surprising trends:

- HubSort offers the least performance benefits
- HubSort causes slowdowns

Category II - Reason For Slowdown With HubSort




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

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

Need a simple mechanism to assign hub vertices from the same part a contiguous range of IDs

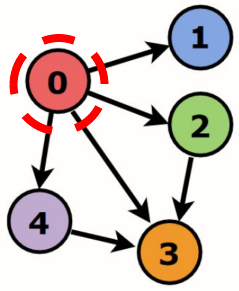
Category IV - Push-based Graph Applications

Push-phase

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

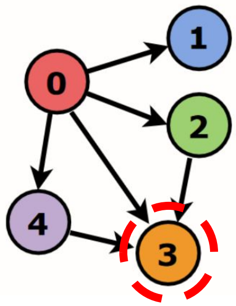
Pull-phase

```
parallel for dst in G:
    for src in inNeigh(v):
        if src in Frontier:
            parent[dst] = src
```



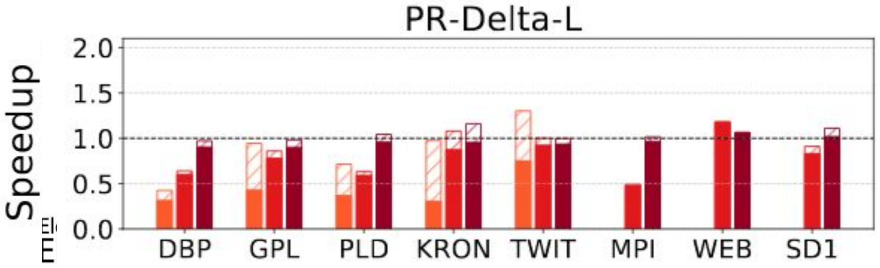
\+ Work Efficient execuztion

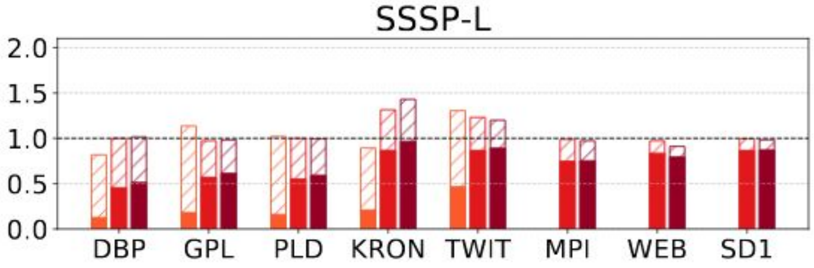
\- Overhead of synchronization



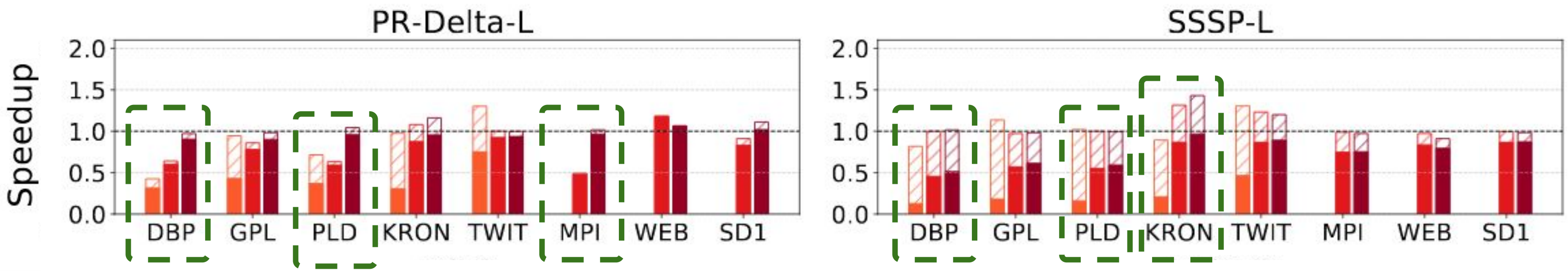
\+ No synchronization required

\- Work-inefficient (iterate over all Vertices)





Category IV - Push-based Graph Applications

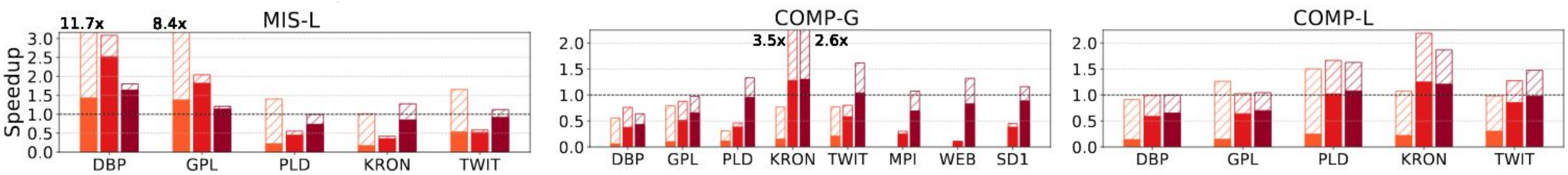


Reordering causes an increase in false-sharing

		DBP	GPL	PLD	KRON	TWIT
PR-δ-L	Rabbit	1.11x	1.53x	1.53x	0.92x	1.26x
	HubSort	0.94x	0.99x	1.43x	1.77x	1.77x
	HubCluster	1.06x	1.01x	1.24x	1.46x	1.27x
SSSP-L	Rabbit	0.87x	1.36x	1.2x	0.95x	0.97x
	HubSort	1.02x	1.14x	1.58x	2.0x	1.4x
	HubCluster	1.14x	1.07x	1.47x	1.58x	1.4x

LWR favors pull-style graph applications

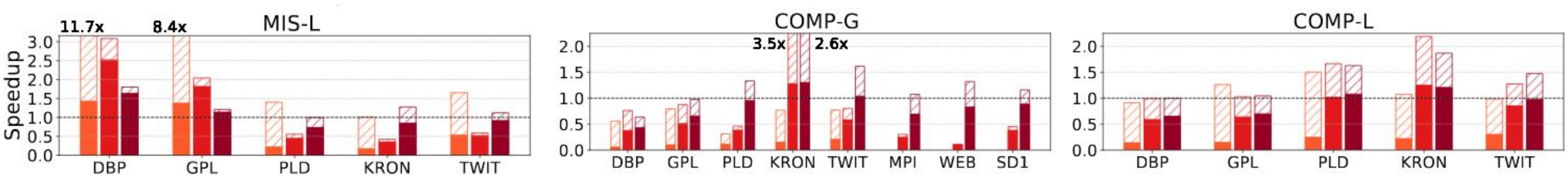
Category V - LWR can affect convergence




```
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

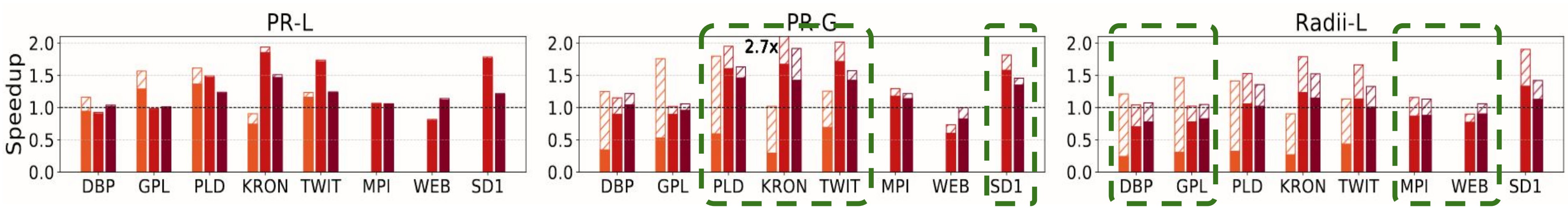


		DBP	GPL	PLD	KRON	TWIT
Comp-G	Rabbit	2.39x	2.0x	5.23x	1.33x	1.47x
	HubSort	1.36x	1.0x	2.09x	0.67x	0.9x
	HubCluster	1.81x	1.0x	1.05x	0.67x	0.88x
Comp-L	Rabbit	1.5x	1.25x	1.27x	1.0x	0.99x
	HubSort	1.25x	1.0x	1.0x	0.67x	0.93x
	HubCluster	1.25x	1.0x	1.0x	0.83x	0.94x
MIS-L	Rabbit	0.3x	0.56x	0.56x	0.96x	0.52x
	HubSort	0.69x	0.56x	0.79x	2.27x	1.01x
	HubCluster	0.85x	0.85x	0.98x	1.19x	1.02x

Increase in Iterations until convergence due to LWR

Opportunity to accelerate convergence by reordering vertices

The Need For Selective Lightweight Reordering

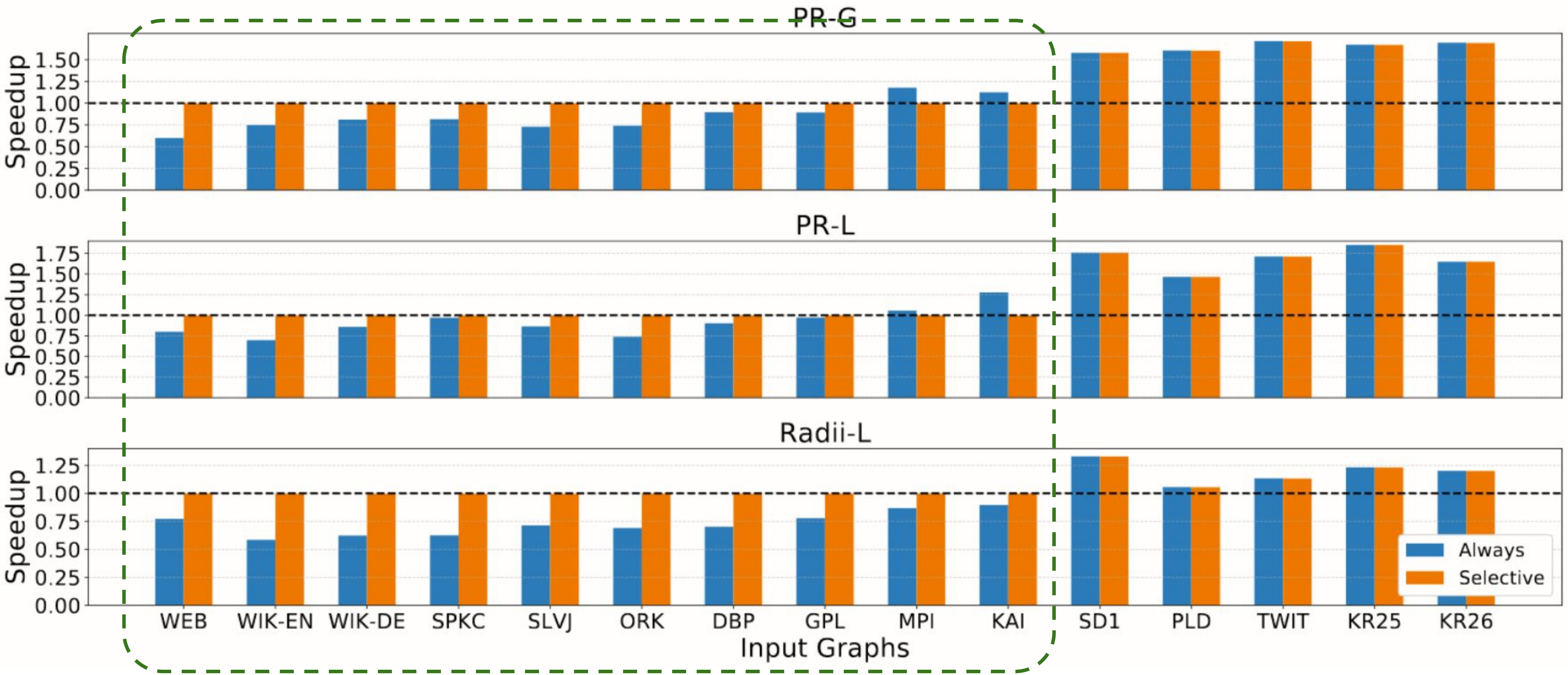


Unconditionally performing LWR causes net slowdowns on some input graphs

Completely avoid LWR misses speedups up to 1.8x

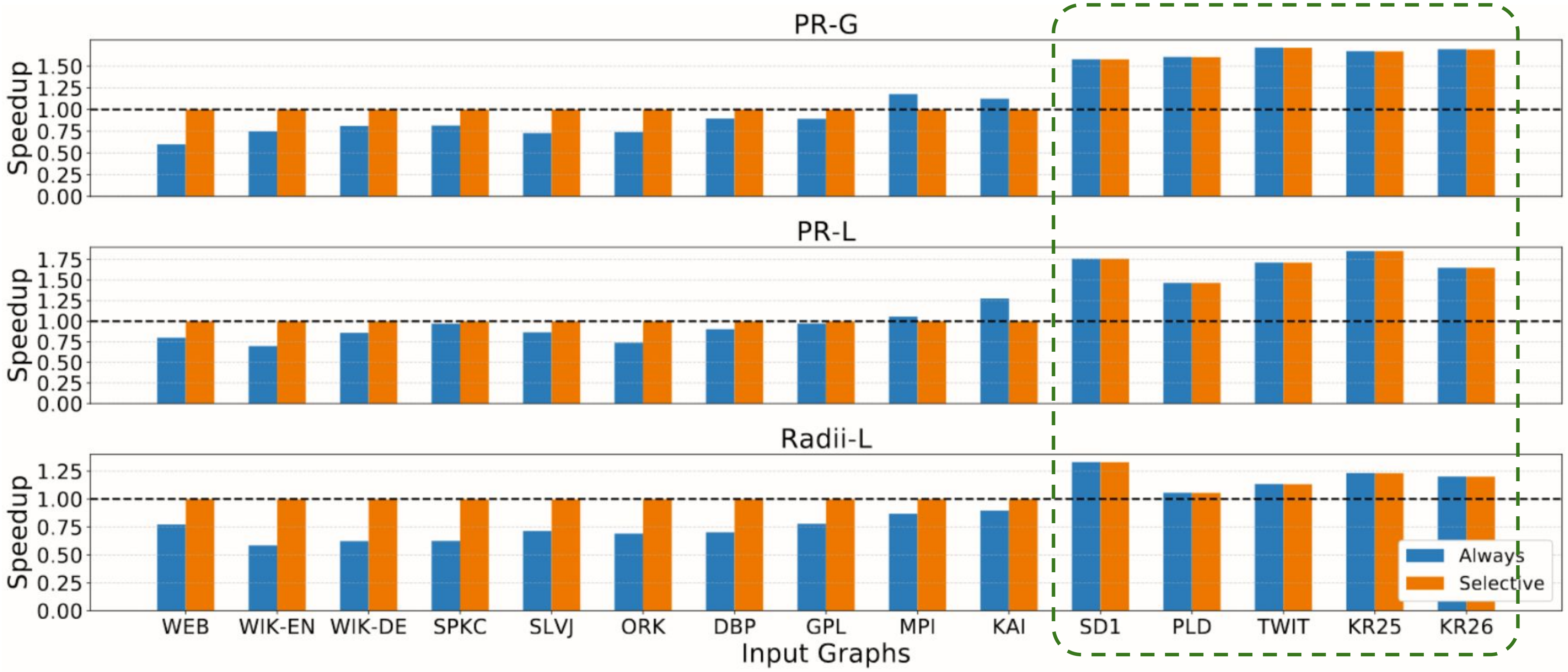
Need to predict speedup from LWR for an input graph and only selectively perform LWR

Using Packing Factor for Selective Reordering



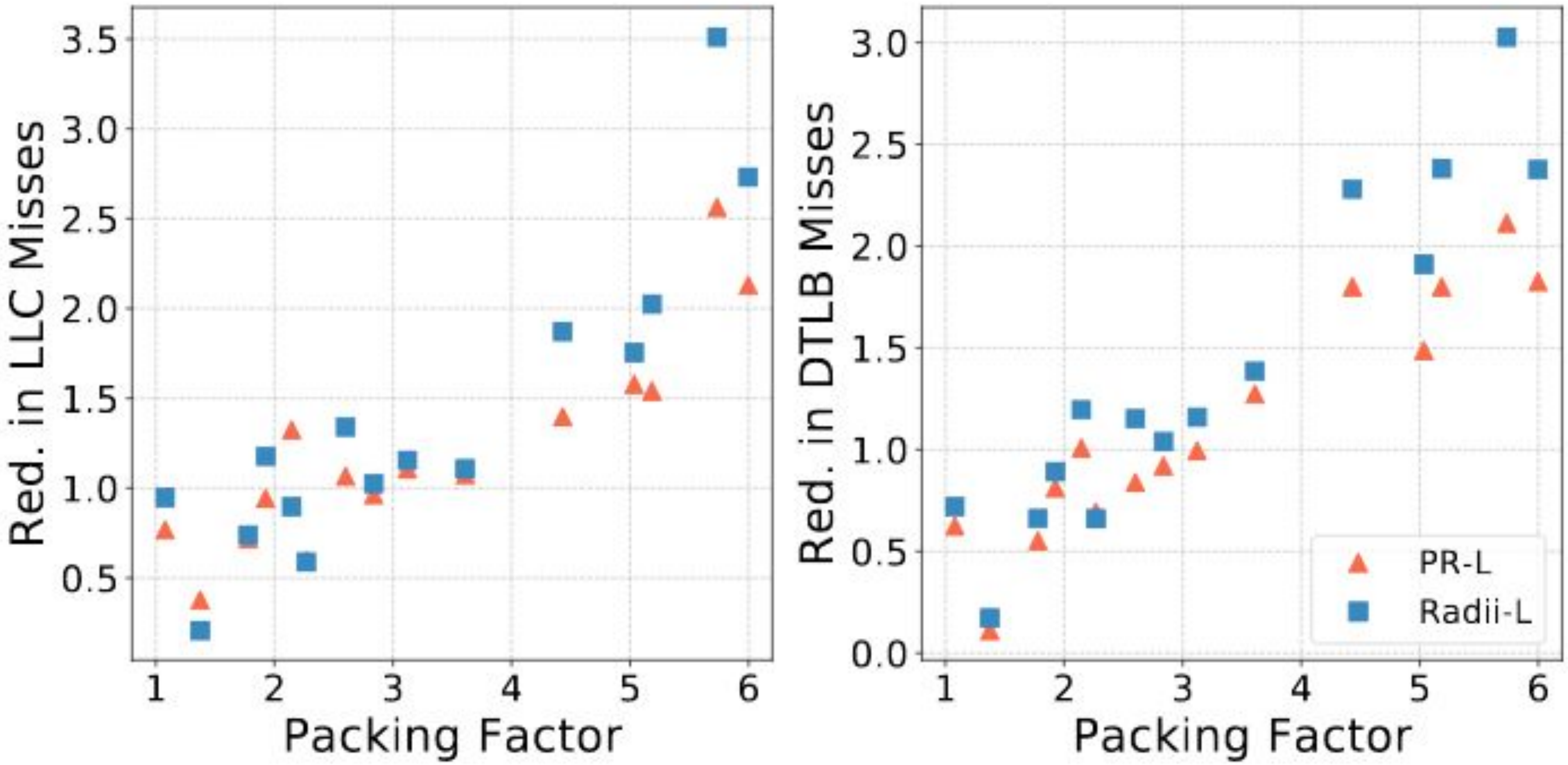
Selective Reordering avoids slowdowns for graphs with low Packing Factor

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

```
procedure COMPUTEPACKINGFACTOR(G)
    numHubs ← 0
    hubWSet_Original ← 0
    for CacheLine in vDataLines do
        containsHub ← False
        for vtx in CacheLine do
            if ISHUB(vtx) then
                numHubs += 1
                containsHub ← True
        if containsHub = True then
            hubWSet_Original += 1
    hubWSet_Sorted ← CEIL(numHubs/VtxPerLine)
    PackingFactor ← hubWSet_Original/hubWSet_Sorted
return PackingFactor
```