Vertex Ordering in graphs

And their evaluation metrics
(Partly based on slides by: Vignesh Balaji and Brandon Lucia)

Reet Barik

School of Electrical Engineering and Computer Science
Washington State University

January 21, 2020

Summary

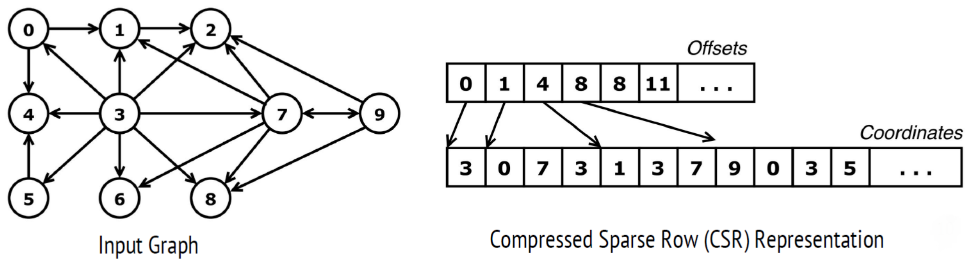
Motivation

Reordering Improves Spatial and Temporal Locality

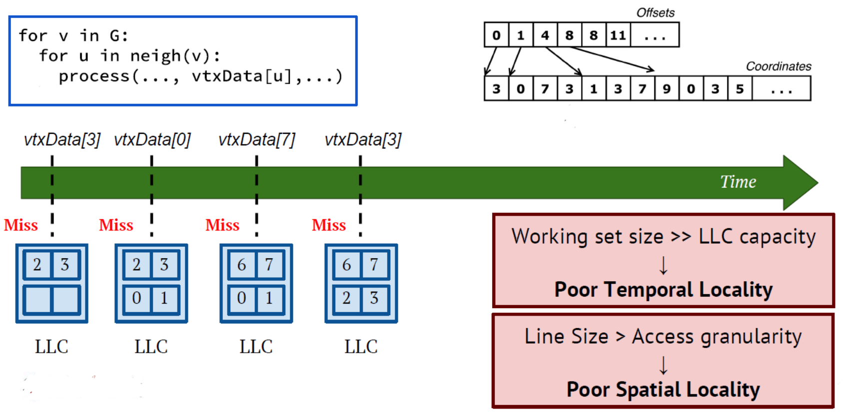
```
for v in G:
  for u in neigh(v):
    process(..., vtxData[u],...)
```

Typical graph processing kernel

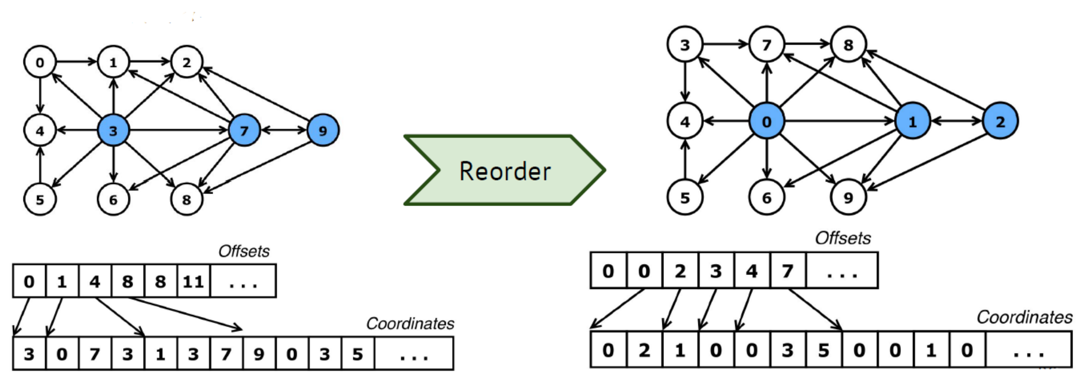
Reordering Improves Spatial and Temporal Locality



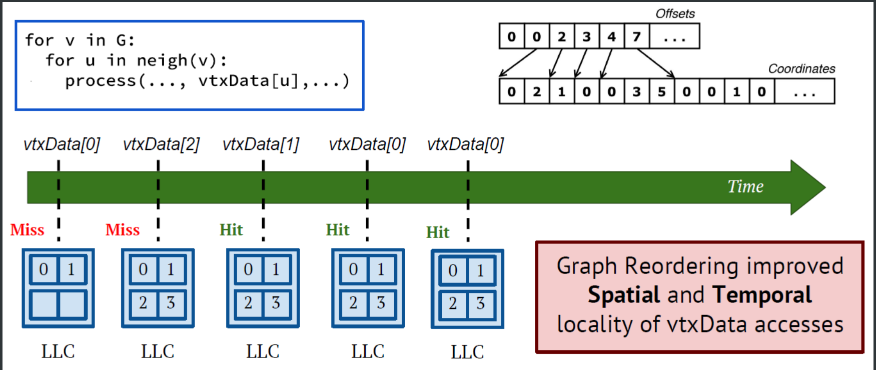
Reordering Improves Spatial and Temporal Locality



Reordering Improves Spatial and Temporal Locality



Reordering Improves Spatial and Temporal Locality



Gorder

Terminology

The following is the terminology used in the paper:

1. It takes a directed graph G = (V,E) as the input where V(G) represents the set of nodes and E(G) represents the set of edges.
2. The number of nodes and edges are denoted as $n = |V(G)|$ and $m = |E(G)|$, respectively.
3. The out-neighbor set and in-neighbor set of a node u is denoted by $N_O(u)$ and $N_I(u)$ such that $N_O(u) = \{v \mid (u,v) \in E(G)\}$ and $N_I(u) = \{v \mid (v,u) \in E(G)\}$.
4. The in-degree, out-degree, and the degree of a node u is denoted as, $d_I(u) = |N_I(u)|$, $d_O(u) = |N_O(u)|$, and $d(u) = d_I(u) + d_O(u)$.
5. Neighbors: two nodes are neighbors if there exists an edge between them.
6. Siblings: two nodes are sibling nodes if they share a common in-neighbor.

Terminology

The following is the terminology used in the paper:

1. It takes a directed graph G = (V,E) as the input where V(G) represents the set of nodes and E(G) represents the set of edges.
2. The number of nodes and edges are denoted as $n = |V(G)|$ and $m = |E(G)|$, respectively.
3. The out-neighbor set and in-neighbor set of a node u is denoted by $N_O(u)$ and $N_I(u)$ such that $N_O(u) = \{v \mid (u, v) \in E(G)\}$ and $N_I(u) = \{v \mid (v, u) \in E(G)\}$.
4. The in-degree, out-degree, and the degree of a node u is denoted as, $d_I(u) = |N_I(u)|$, $d_O(u) = |N_O(u)|$, and $d(u) = d_I(u) + d_O(u)$.
5. Neighbors: two nodes are neighbors if there exists an edge between them.
6. Siblings: two nodes are sibling nodes if they share a common in-neighbor.

Terminology

The following is the terminology used in the paper:

1. It takes a directed graph G = (V,E) as the input where V(G) represents the set of nodes and E(G) represents the set of edges.
2. The number of nodes and edges are denoted as $n = |V(G)|$ and $m = |E(G)|$, respectively.
3. The out-neighbor set and in-neighbor set of a node u is denoted by $N_O(u)$ and $N_I(u)$ such that $N_O(u) = \{v \mid (u, v) \in E(G)\}$ and $N_I(u) = \{v \mid (v, u) \in E(G)\}$.
4. The in-degree, out-degree, and the degree of a node u is denoted as, $d_I(u) = |N_I(u)|$, $d_O(u) = |N_O(u)|$, and $d(u) = d_I(u) + d_O(u)$.
5. Neighbors: two nodes are neighbors if there exists an edge between them.
6. Siblings: two nodes are sibling nodes if they share a common in-neighbor.

Terminology

The following is the terminology used in the paper:

1. It takes a directed graph G = (V,E) as the input where V(G) represents the set of nodes and E(G) represents the set of edges.
2. The number of nodes and edges are denoted as $n = |V(G)|$ and $m = |E(G)|$, respectively.
3. The out-neighbor set and in-neighbor set of a node u is denoted by $N_O(u)$ and $N_I(u)$ such that $N_O(u) = \{v \mid (u, v) \in E(G)\}$ and $N_I(u) = \{v \mid (v, u) \in E(G)\}$.
4. The in-degree, out-degree, and the degree of a node u is denoted as, $d_I(u) = |N_I(u)|$, $d_O(u) = |N_O(u)|$, and $d(u) = d_I(u) + d_O(u)$.
5. Neighbors: two nodes are neighbors if there exists an edge between them.
6. Siblings: two nodes are sibling nodes if they share a common in-neighbor.

Terminology

The following is the terminology used in the paper:

1. It takes a directed graph G = (V,E) as the input where V(G) represents the set of nodes and E(G) represents the set of edges.
2. The number of nodes and edges are denoted as $n = |V(G)|$ and $m = |E(G)|$, respectively.
3. The out-neighbor set and in-neighbor set of a node u is denoted by $N_O(u)$ and $N_I(u)$ such that $N_O(u) = \{v \mid (u,v) \in E(G)\}$ and $N_I(u) = \{v \mid (v,u) \in E(G)\}$.
4. The in-degree, out-degree, and the degree of a node u is denoted as, $d_I(u) = |N_I(u)|$, $d_O(u) = |N_O(u)|$, and $d(u) = d_I(u) + d_O(u)$.
5. Neighbors: two nodes are neighbors if there exists an edge between them.
6. Siblings: two nodes are sibling nodes if they share a common in-neighbor.

Terminology

The following is the terminology used in the paper:

1. It takes a directed graph G = (V,E) as the input where V(G) represents the set of nodes and E(G) represents the set of edges.
2. The number of nodes and edges are denoted as $n = |V(G)|$ and $m = |E(G)|$, respectively.
3. The out-neighbor set and in-neighbor set of a node u is denoted by $N_O(u)$ and $N_I(u)$ such that $N_O(u) = \{v \mid (u, v) \in E(G)\}$ and $N_I(u) = \{v \mid (v, u) \in E(G)\}$.
4. The in-degree, out-degree, and the degree of a node u is denoted as, $d_I(u) = |N_I(u)|$, $d_O(u) = |N_O(u)|$, and $d(u) = d_I(u) + d_O(u)$.
5. Neighbors: two nodes are neighbors if there exists an edge between them.
6. Siblings: two nodes are sibling nodes if they share a common in-neighbor.

Algorithm

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

Typical graph processing kernel

It can be observed that both the neighbor and sibling type of relationships need to be taken into account.

Algorithm

The metric defined is aimed to capture the locality between two vertices. For two nodes u and v, the scoring function is given by:

$$S(u, v) = S_s(u, v) + S_n(u, v)$$

where,

- $S_s(u, v)$ is the number of the times that u and v co-exist in sibling relationships, which is the number of their common in-neighbors.
- $S_n(u, v)$ is the number of times that u and v are neighbors, which is either 0, 1, or 2.

Algorithm

The metric defined is aimed to capture the locality between two vertices. For two nodes u and v, the scoring function is given by:

$$S(u, v) = S_s(u, v) + S_n(u, v)$$

where,

- $S_s(u, v)$ is the number of the times that u and v co-exist in sibling relationships, which is the number of their common in-neighbors.
- $S_n(u, v)$ is the number of times that u and v are neighbors, which is either 0, 1, or 2.

Algorithm

The metric defined is aimed to capture the locality between two vertices. For two nodes u and v, the scoring function is given by:

$$S(u,v) = S_s(u,v) + S_n(u,v)$$

where,

- $S_s(u,v)$ is the number of the times that u and v co-exist in sibling relationships, which is the number of their common in-neighbors.
- $S_n(u,v)$ is the number of times that u and v are neighbors, which is either 0, 1, or 2.

Algorithm

The metric defined is aimed to capture the locality between two vertices. For two nodes u and v, the scoring function is given by:

$$S(u,v) = S_s(u,v) + S_n(u,v)$$

where,

- $S_s(u,v)$ is the number of the times that u and v co-exist in sibling relationships, which is the number of their common in-neighbors.
- $S_n(u,v)$ is the number of times that u and v are neighbors, which is either 0, 1, or 2.

Algorithm

The metric defined is aimed to capture the locality between two vertices. For two nodes u and v, the scoring function is given by:

$$S(u, v) = S_s(u, v) + S_n(u, v)$$

where,

- $S_s(u, v)$ is the number of the times that u and v co-exist in sibling relationships, which is the number of their common in-neighbors.
- $S_n(u, v)$ is the number of times that u and v are neighbors, which is either 0, 1, or 2.

Algorithm

- The solution offered takes the 'sliding window' approach.
- If there are two nodes u and v with ordering $\phi(u)$ and $\phi(v)$ respectively such that u comes before v in the ordering. For a fixed v and window size w, the algorithm takes a look at all the combination of u and v, for all nodes u that come before v in the sliding window of size w.
- The problem statement is as follows:
  Find the optimal graph ordering $\phi(\cdot)$, that maximizes $Gscore$ (the sum of locality score), F(·), based on a sliding window model with a window size w, where,

$$F(\phi) = \sum_{0<\phi(v)-\phi(u)\leq w} S(u,v)$$

Algorithm

- The solution offered takes the 'sliding window' approach.
- If there are two nodes u and v with ordering $\phi(u)$ and $\phi(v)$ respectively such that u comes before v in the ordering. For a fixed v and window size w, the algorithm takes a look at all the combination of u and v, for all nodes u that come before v in the sliding window of size w.
- The problem statement is as follows:
  Find the optimal graph ordering $\phi(\cdot)$, that maximizes $Gscore$ (the sum of locality score), F(·), based on a sliding window model with a window size w, where,

$$F(\phi) = \sum_{0<\phi(v)-\phi(u)\leq w} S(u,v)$$

Algorithm

- The solution offered takes the ‘sliding window’ approach.
- If there are two nodes u and v with ordering $\phi(u)$ and $\phi(v)$ respectively such that u comes before v in the ordering. For a fixed v and window size w, the algorithm takes a look at all the combination of u and v, for all nodes u that come before v in the sliding window of size w.
- The problem statement is as follows:
  Find the optimal graph ordering $\phi(\cdot)$, that maximizes $Gscore$ (the sum of locality score), F(·), based on a sliding window model with a window size w, where,

$$F(\phi) = \sum_{0<\phi(v)-\phi(u)\leq w} S(u,v)$$

Algorithm

- If window size is 1, the problem reduces to the maximum traveling salesman problem.
- This problem can be thought of as a variant of maxTSP.
- solved by constructing an edge-weighted complete undirected graph G_w from the original graph G where the vertex set of G_w is the same as G and since it is a complete graph, there is an edge between every pair of nodes in G_w.
- The weight of an edge in G_w is the score of the two end vertices of that edge computed over the original graph G.
- Under this setting, the optimal maxTSP-w over G is the solution of maxTSP over G_w.

Algorithm

- If window size is 1, the problem reduces to the maximum traveling salesman problem.
- This problem can be thought of as a variant of maxTSP.
- solved by constructing an edge-weighted complete undirected graph G_w from the original graph G where the vertex set of G_w is the same as G and since it is a complete graph, there is an edge between every pair of nodes in G_w.
- The weight of an edge in G_w is the score of the two end vertices of that edge computed over the original graph G.
- Under this setting, the optimal maxTSP-w over G is the solution of maxTSP over G_w.

Algorithm

- If window size is 1, the problem reduces to the maximum traveling salesman problem.
- This problem can be thought of as a variant of maxTSP.
- solved by constructing an edge-weighted complete undirected graph G_w from the original graph G where the vertex set of G_w is the same as G and since it is a complete graph, there is an edge between every pair of nodes in G_w.
- The weight of an edge in G_w is the score of the two end vertices of that edge computed over the original graph G.
- Under this setting, the optimal maxTSP-w over G is the solution of maxTSP over G_w.

Algorithm

- If window size is 1, the problem reduces to the maximum traveling salesman problem.
- This problem can be thought of as a variant of maxTSP.
- solved by constructing an edge-weighted complete undirected graph G_w from the original graph G where the vertex set of G_w is the same as G and since it is a complete graph, there is an edge between every pair of nodes in G_w.
- The weight of an edge in G_w is the score of the two end vertices of that edge computed over the original graph G.
- Under this setting, the optimal maxTSP-w over G is the solution of maxTSP over G_w.

Algorithm

- If window size is 1, the problem reduces to the maximum traveling salesman problem.
- This problem can be thought of as a variant of maxTSP.
- solved by constructing an edge-weighted complete undirected graph G_w from the original graph G where the vertex set of G_w is the same as G and since it is a complete graph, there is an edge between every pair of nodes in G_w.
- The weight of an edge in G_w is the score of the two end vertices of that edge computed over the original graph G.
- Under this setting, the optimal maxTSP-w over G is the solution of maxTSP over G_w.

Experimental Space

9
Graph
Reordering
Techniques

9
Algorithms

8
Input Graphs

Server Configuration
Intel Core i7-4770@3.40GHz
CPU and 32 GB memory

Experimental Space

Nine graph ordering techniques:

- Original
- MINLA
- MLOGA
- RCM
- DegSort
- CHDFS
- SlashBurn
- LDG
- METIS

Experimental Space

Nine graph applications:

- Neighbors Query (NQ)
- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Strongly Connected Component (SCC) detection
- Shortest Paths (SP) by the Bellman-Ford algorithm
- PageRank (PR)
- Dominating Set (DS)
- graph decomposition (Kcore)
- graph diameter (Diam)

Experimental Space

Eight real world graphs:

- Pokec
- LiveJournal
- Flickr
- wikilink
- Google+
- twitter
- PLD
- SD1

Experimental Results

CPU Cache Miss Ratio:

- METIS fails to compute the graph partitions for the other 5 larger graph except Pokec, Flickr and LiveJournal, due to its excessive memory consumption. Hence, not shown in second table.
- The cache statistics are collected by the 'perf' tool.

Order	L1-ref	L1-mr	L3-ref	L3-r	Cache-mr
Original	11,109M	52.1%	2,195M	19.7%	5.1%
MINLA	11,110M	58.1%	2,121M	19.0%	4.5%
MLOGA	11,119M	53.1%	1,685M	15.1%	4.1%
RCM	11,102M	49.8%	1,834M	16.5%	4.1%
DegSort	11,121M	58.3%	2,597M	23.3%	5.3%
CHDFS	11,107M	49.9%	1,850M	16.7%	4.4%
SlashBurn	11,096M	55.0%	2,466M	22.2%	4.3%
LDG	11,112M	52.9%	2,256M	20.3%	5.4%
METIS	11,105M	50.3%	2,235M	20.1%	5.2%
Gorder	11,101M	**37.9%**	**1,280M**	**11.5%**	**3.4%**

Cache Statistics by *PR* over Flickr (M = Millions)

Order	L1-ref	L1-mr	L3-ref	L3-r	Cache-mr
Original	623.9B	58.4%	180.0B	28.8%	18.6%
MINLA	628.8B	62.5%	196.6B	31.2%	14.8%
MLOGA	620.0B	62.1%	189.6B	30.5%	14.3%
RCM	628.9B	44.9%	103.8B	16.5%	10.2%
DegSort	632.2B	55.1%	149.5B	23.6%	15.9%
CHDFS	630.3B	38.0%	101.2B	16.1%	10.9%
SlashBurn	628.8B	44.5%	121.0B	19.3%	13.7%
LDG	637.9B	58.4%	186.2B	29.2%	18.6%
Gorder	620.3B	**31.5%**	**79.5B**	**12.8%**	**8.2%**

Cache Statistics by *PR* over sd1-arc (B = Billions)

Experimental Results

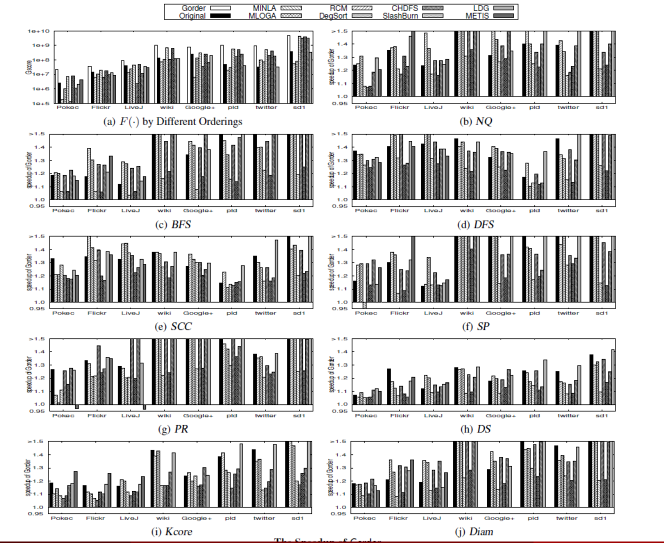
Running time of Gorder:

- Adjacent table shows running time of Gorder for reference.
- The speedup of Gorder over another ordering X is shown as the relative difference of $T(X)/T(Gorder)$ (in the next slide)

Order	*NQ*	*BFS*	*DFS*	*SCC*	*SP*	*PR*	*DS*	*Kcore*	*Diam*
Pokec	8.7	2.0	2.5	5.2	1.3	12.3	10.4	6.6	1,003
Flickr	5.1	1.5	1.8	3.7	1.0	9.1	8.6	5.3	620
LiveJ	19.4	4.9	5.9	12.1	4.6	26.4	24.0	16.8	2,556
wikilink	56.1	10.0	14.3	28.5	35.3	81.9	85.7	50.0	5,932
Google+	134	35.0	43.3	87.6	28.6	210	183	131	17,936
pld-arc	199	45.2	55.7	115	40.4	305	251	177	14,389
twitter	467	79.2	80.9	158	74.4	819	535	378	32,808
sd1-arc	492	83.7	104	218	120	665	587	430	30,202

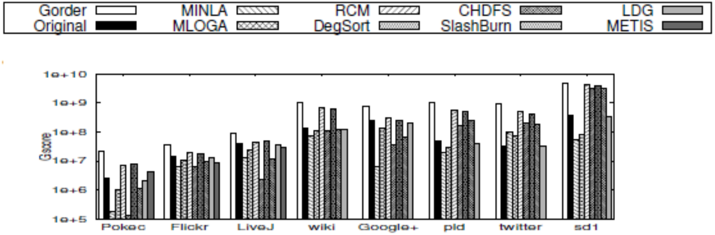
Running time by Gorder (in second)

Experimental Results



Experimental Results

'Gscore' by different orderings:



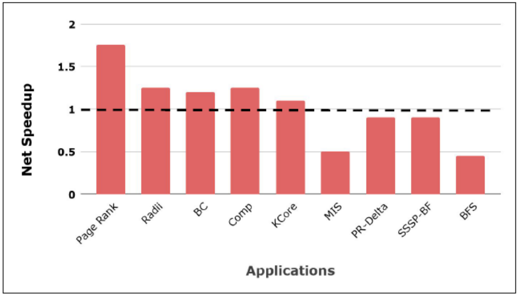
(a) $F(\cdot)$ by Different Orderings

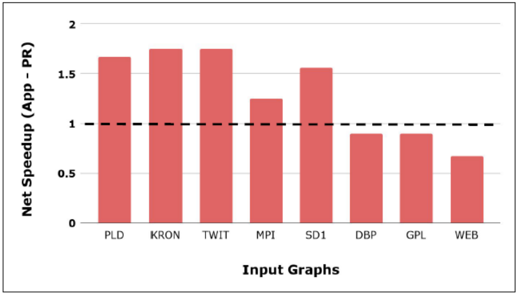
Lightweight Reordering

Need for Lightweight Reordering

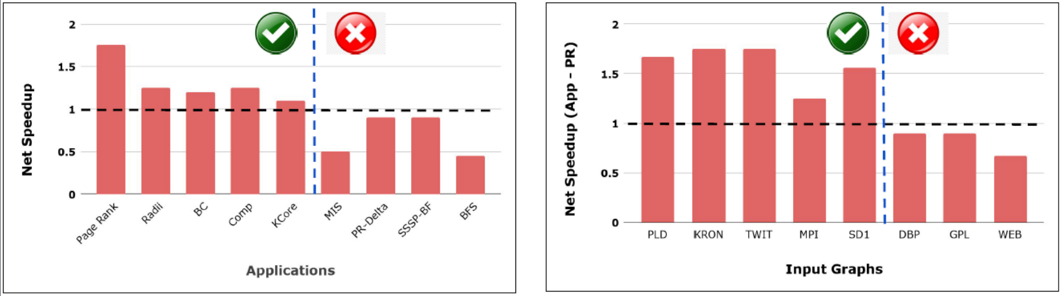
$$Speedup = \frac{T_{Original}}{T_{Reordered} + ReorderingTime}$$

Need for Lightweight Reordering





Need for Lightweight Reordering



Experimental 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)

Experimental Space

Three Graph Ordering Techniques:

- Rabbit Ordering
- Hub-Sorting
- Hub-Clustering

Experimental Space

Fifteen Graph Applications from the from the GAP and Ligra benchmark suites:

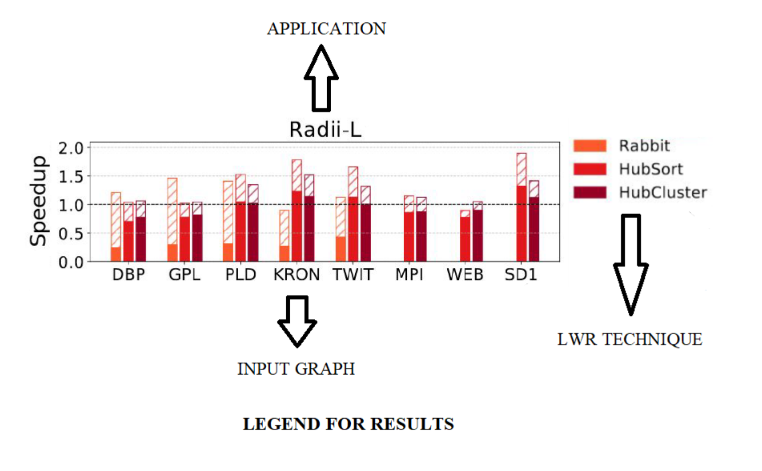
- Page Rank (PR-G)
- Page Rank (PR-L)
- Radii Estimation (Radii-L)
- Collaborative Filtering (CF-L)
- Connected Components (Comp-G)
- Connected Components (Comp-L)
- Maximal Independent Set (MIS-L)
- Page Rank-Delta (PR-Delta-L)
- SSSP-Bellman Ford (SSSP-L)
- Betweenness Centrality (BC-G)
- Betweenness Centrality (BC-L)
- SSSP-Delta Stepping (SSSP-G)
- Breadth First Search (BFS-G)
- Breadth First Search (BFS-L)
- K-core Decomposition (KCore-L)

Experimental Space

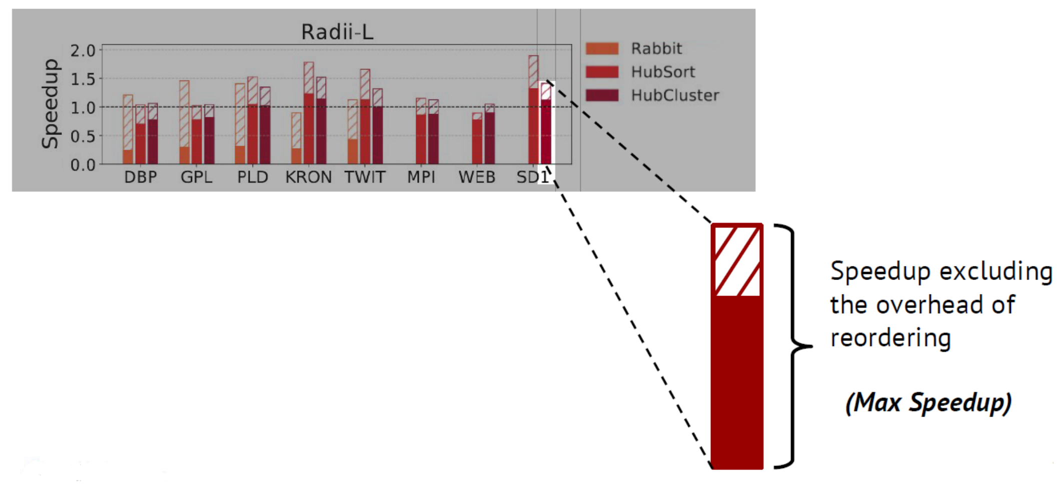
Eight real input graphs:

- DBP
- GPL
- PLD
- KRON
- TWIT
- MPI
- WEB
- SD1

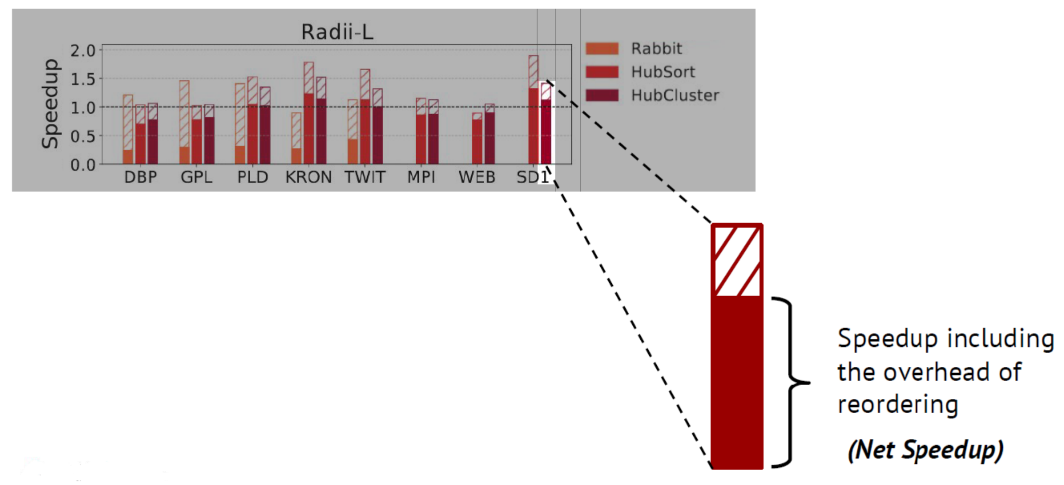
Experimental Results



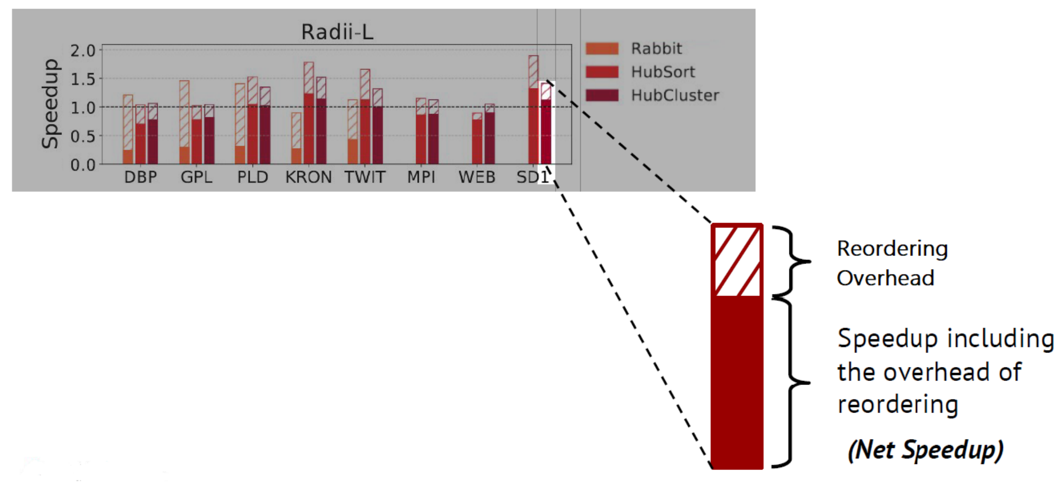
Experimental Results



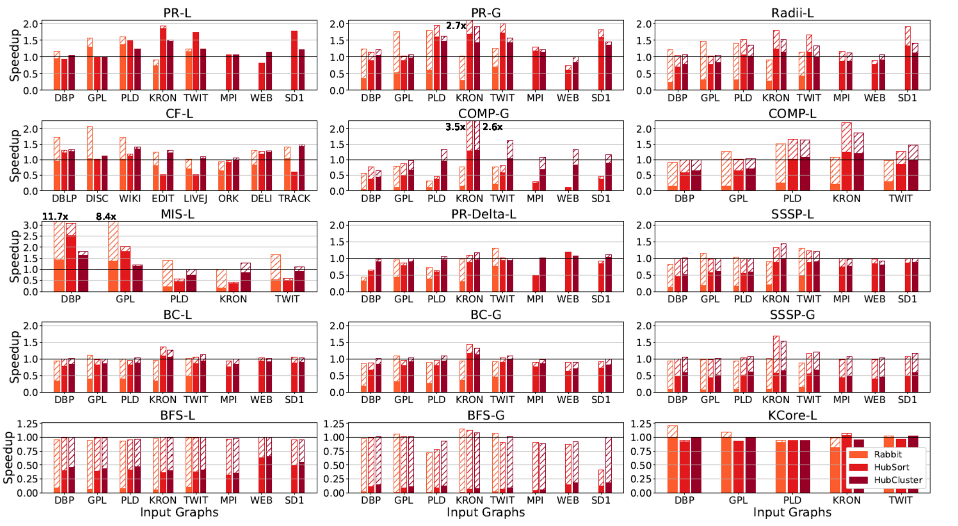
Experimental Results



Experimental Results



Experimental Results



Experimental Results

Points to be noted:

- The baseline is an execution on the input graph as originally ordered by the publishers of the graph datasets.
- Data for Rabbit ordering on MPI, WEB, and SD1 was omitted because the machine's 64GB of memory was exhausted.
- Same as above for COMP-L, MIS-L, and KCore-L for the undirected versions of the same graphs.
- CF-L was tried on a different set of input graphs: DBLP, DISC, WIKI, EDIT, LIVEJ, ORK, DELI, and TRACK (not mentioned in the paper why).

Experimental Findings

The following are the takeaways from the application point of view:

- Applications like Page Rank and Radii which process a large fraction of edges in each iteration are most suitable for LWR.
- Symmetric bipartite graphs are poor candidates for LWR (require bi-partiteness aware reordering).
- Reordering affects convergence for applications with ID-dependent computations.
- Push-style applications (like Page Rank Delta and SSSP-Bellman Ford) or those that process very few edges per iteration (like BC, SSSP-Delta Stepping, BFS, and KCore) do not benefit from LWR. *This is because of 'False Sharing' and limited reuse of $vtxData$ and NOT because of reordering overhead.*
- For those applications where Hubsort is effective, the benefits are input graph dependent (some input graphs can cause no speedup or a net slowdown due to overhead).

Experimental Findings

The following are the takeaways from the application point of view:

- Applications like Page Rank and Radii which process a large fraction of edges in each iteration are most suitable for LWR.
- Symmetric bipartite graphs are poor candidates for LWR (require bi-partiteness aware reordering).
- Reordering affects convergence for applications with ID-dependent computations.
- Push-style applications (like Page Rank Delta and SSSP-Bellman Ford) or those that process very few edges per iteration (like BC, SSSP-Delta Stepping, BFS, and KCore) do not benefit from LWR. *This is because of 'False Sharing' and limited reuse of $vtxData$ and NOT because of reordering overhead.*
- For those applications where Hubsort is effective, the benefits are input graph dependent (some input graphs can cause no speedup or a net slowdown due to overhead).

Experimental Findings

The following are the takeaways from the application point of view:

- Applications like Page Rank and Radii which process a large fraction of edges in each iteration are most suitable for LWR.
- Symmetric bipartite graphs are poor candidates for LWR (require bi-partiteness aware reordering).
- Reordering affects convergence for applications with ID-dependent computations.
- Push-style applications (like Page Rank Delta and SSSP-Bellman Ford) or those that process very few edges per iteration (like BC, SSSP-Delta Stepping, BFS, and KCore) do not benefit from LWR. *This is because of 'False Sharing' and limited reuse of $vtxData$ and NOT because of reordering overhead.*
- For those applications where Hubsort is effective, the benefits are input graph dependent (some input graphs can cause no speedup or a net slowdown due to overhead).

Experimental Findings

The following are the takeaways from the application point of view:

- Applications like Page Rank and Radii which process a large fraction of edges in each iteration are most suitable for LWR.
- Symmetric bipartite graphs are poor candidates for LWR (require bi-partiteness aware reordering).
- Reordering affects convergence for applications with ID-dependent computations.
- Push-style applications (like Page Rank Delta and SSSP-Bellman Ford) or those that process very few edges per iteration (like BC, SSSP-Delta Stepping, BFS, and KCore) do not benefit from LWR. *This is because of 'False Sharing' and limited reuse of $vtxData$ and NOT because of reordering overhead.*
- For those applications where Hubsort is effective, the benefits are input graph dependent (some input graphs can cause no speedup or a net slowdown due to overhead).

Experimental Findings

The following are the takeaways from the application point of view:

- Applications like Page Rank and Radii which process a large fraction of edges in each iteration are most suitable for LWR.
- Symmetric bipartite graphs are poor candidates for LWR (require bi-partiteness aware reordering).
- Reordering affects convergence for applications with ID-dependent computations.
- Push-style applications (like Page Rank Delta and SSSP-Bellman Ford) or those that process very few edges per iteration (like BC, SSSP-Delta Stepping, BFS, and KCore) do not benefit from LWR. *This is because of 'False Sharing' and limited reuse of $vtxData$ and NOT because of reordering overhead.*
- For those applications where Hubsort is effective, the benefits are input graph dependent (some input graphs can cause no speedup or a net slowdown due to overhead).

Experimental Findings

From the input-graph point of view, it was observed that Hubsort was most effective when the graphs had the following properties:

- There is a skew in degree distribution indicating the presence of hubs.
- The hub vertices are sparsely distributed which is an indication of the quality of the original ordering.

Experimental Findings

From the input-graph point of view, it was observed that Hubsort was most effective when the graphs had the following properties:

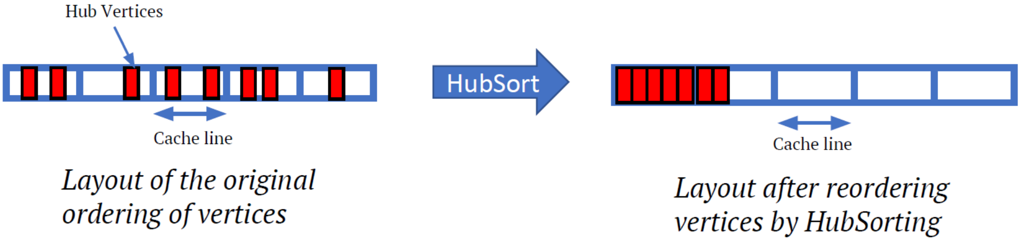
- There is a skew in degree distribution indicating the presence of hubs.
- The hub vertices are sparsely distributed which is an indication of the quality of the original ordering.

Experimental Findings

'Packing Factor' metric was used to capture those two properties. It is a measure of how densely the hubs will be packed after Hubsorting.

Experimental Findings

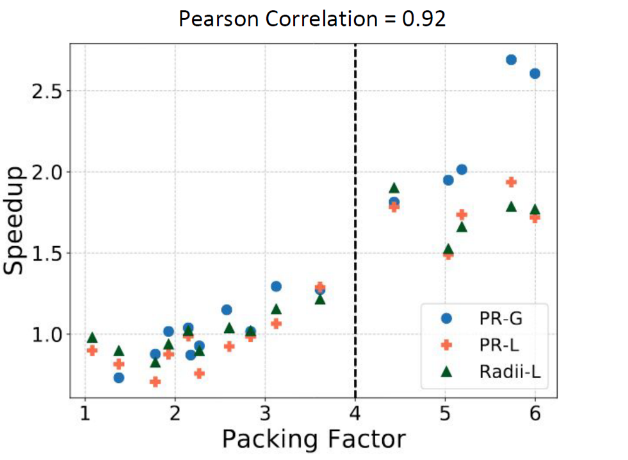
'Packing Factor' metric was used to capture those two properties. It is a measure of how densely the hubs will be packed after Hubsorting.



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

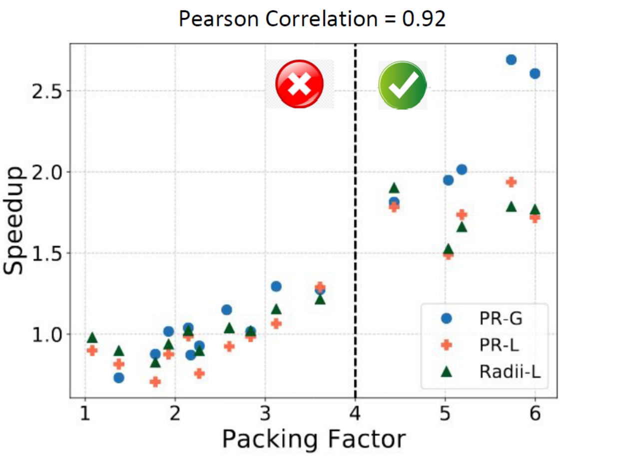
Experimental Findings

'Packing Factor' is a good indicator of Speed-up from Hubsorting:



Experimental Findings

'Packing Factor' is a good indicator of Speed-up from Hubsorting:



Experimental Conclusion

The following pseudocode can be followed:

$PF \leftarrow computePF(G)$
if $PF \geq 4$ **then**
  $G' \leftarrow HubSort(G)$
  Process(G')
else
  Process(G)
end if

When the original graph is processed (for $PF \leq 4$), there is no net speed-up. But, net slowdown is prevented.

Acknowledgments

The author is extremely thankful to Prof. Ananth Kalyanaraman for the opportunity to present on this interesting topic.

The End