Final Exam
Master of Science in Computer Science (Non-thesis)

By: Reet Barik

November 20, 2019

Abstract

For most parallel graph-theoretic applications, data movement has become a primary bottleneck toward achieving scale in computing. One of the primary contributors to data movement is how the graph is stored or represented in memory. To this end, various vertex ordering schemes have been studied in the past. There are generally two schools of approaches here i.e. conduct a heavy-weight preprocessing step to reorder vertices with a goal to subsequently reduce or optimize data movement during the processing of the graph; or alternatively, perform a lighter weight reordering depending on the end-application characteristics.

1 Introduction

Studying data movement in database systems is one of many applications for which graph processing algorithms are used. Path planning, social network analysis, recommender systems etc are some of them. In database systems, CPU cache performance is plagued by efficiency issues, so much so that almost half the processing time comprises of cache miss latency [7]. The problem of poor data locality is caused by irregular memory access. Real world graphs present an opportunity to improve the locality of graph applications because of the presence of clusters or hubs.

The first paper [28] that this report looks at aims to increase CPU speedup for graph computing by reducing the cache miss ratio for different graph algorithms with the help of a general solution which is target algorithm agnostic. The approach is to come up with something called ‘graph ordering’ which is an optimal permutation among all nodes wherein, all nodes that are frequently accessed together, are stored locally in the memory. This paper shows that the problem of finding the ‘graph ordering’ of a given graph is NP-hard and proposes a basic algorithm with a bounded approximation because of the nature of the problem. It also goes on to propose a new algorithm to improve the time complexity of the basic algorithm with the help of a new optimization technique based on a new data structure (priority queue). The proposed approach is then evaluated against 9 possible graph orderings using 8 real large graphs on 9 representative graph algorithms. The result shows high performance by reduction in CPU cache miss ratio (best speedup is > 2).

Sophisticated reordering techniques such as the one mentioned above have an extremely large overhead to come up with a ‘graph ordering’. Some graph processing applications use the input graph enough times for the locality benefits to far outweigh the reordering overhead incurred. But for those applications which do not use the same input graph multiple times, or takes input graphs which are dynamic in nature (like Page ranks in dynamically changing social networks [32] or or tracking changes in the diameter of an evolving graph [12]), the reordering overhead is far too high for the reordering techniques to be viable. This begs the question of what properties are exhibited by graph applications and input graphs for which graph reordering is an optimization. This is the main query that the second paper [33] that this report focuses on, tries to answer. Firstly, it identifies lightweight reordering techniques that improves performance despite the reordering overhead. These are evaluated across a multiple applications (like PageRank (PR-G and PR-L) [31], Radii Estimation (Radii-L), SSSP-Delta Stepping [8] etc.) to empirically identify the types of applications that benefit most from this approach. These lightweight techniques are not generalizable over all types of input graphs. Hence, this paper further investigates the link between the speedup observed through experiments and the structural properties of the input graphs. This dependence is exploited and consequently a low-overhead technique is proposed which determines whether a given input graph is a good candidate for reordering. The results show that this ’selective’ lightweight reordering gives a maximum end-to-end speedup of up to 1.75x and never causes a slowdown beyond 0.1%.

The task for this exam is to read the two papers cited above, critically evaluate both papers, contrast and compare them (including any similarities and differences in their approaches and any tradeoffs), and present own views and ideas (if any) on the problem.

2 Related Works

There are numerous graph ordering techniques that have existed in the literature for some time. PT-Scotch is a tool [23] for efficient parallel graph ordering. It extends Scotch [26] and its graph ordering capabilities in the parallel domain. The purpose of Scotch is to apply graph theory, with a divide and conquer approach, to scientific computing problems. A lightweight reordering technique is Rabbit Order [24] which aims to achieve high locality and fast reordering. There are two main approaches by which it accomplishes that. The first one is hierarchical community based ordering. And the second one is parallel incremental aggregation. Some of the works deal directly with cache. Cache-Guided Scheduling [5] is one of those where the underlying concept is that the cache inherently knows which vertices are stored in the cache and hence it is best positioned to come up with a schedule that maximizes locality. 'Scheduling' here is analogous to 'ordering' and means the same thing as coming up with a way to determine the relative ordering in which vertices are processed. Cagra [17] is a cache-optimized in-memory graph framework that uses a new technique called 'CSR segmenting'.
There are approaches like Park [20] and Then [30] which are applicable for specific graph applications only. Banerjee [6] improves the efficiency of the DAG traversal by proposing a node ordering scheme by the children-depth-first traversal method. Mendelzon [11] minimizes the number of I/Os needed to access the graph by proposing a heuristic method for clustering nodes in the local area in the same disk page. Auroux [25] reorders the node set by BFS [13]. Kang [4] removes the hub nodes iteratively from the graph using a graph decomposition approach. Boldi [22] computes the clusters of the graph by label propagation and graph compression techniques. Metis [18] is a set of programs that implement multilevel graph

partitioning algorithms. Petit [10] attempts to approach this problem in the Minimum Linear Arrangement (minLA) setting. Karantasis [21] attempts to parallelize two popular reordering algorithms, RCM and Sloan.

3 Algorithm Description

3.1 Background

This section will provide an overview of concepts and terms from both papers under consideration.

- The first paper uses the following terminology:

  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 and edge between them.
  6. Siblings: two nodes are sibling nodes if they share a common in-neighbor.

- The second paper introduces a metric called the 'Packing Factor'. The idea is to quantify the decrease in sparsity of the hub vertices from the point of view of cache lines. It is essentially the ratio of the original graph's hub working set to the minimum number of cache lines in which the graph's hubs can fit, based solely on cache line capacity. The algorithm for computing 'Packing Factor' is shown below:

Algorithm 2 Computing the Packing Factor of a graph

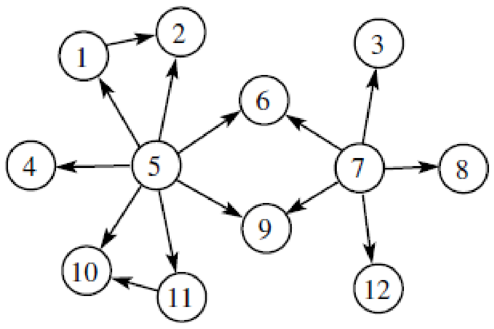
```
1:  procedure COMPUTEPACKINGFACTOR($G$)
2:      $numHubs \leftarrow 0$
3:      $hubWSet\_Original \leftarrow 0$
4:      for $CacheLine$ in $vDataLines$ do
5:          $containsHub \leftarrow False$
6:          for $vtx$ in $CacheLine$ do
7:              if ISHUB($vtx$) then
8:                  $numHubs$ += 1
9:                  $containsHub \leftarrow True$
10:         if $containsHub = True$ then
11:             $hubWSet\_Original$ += 1
12:     $hubWSet\_Sorted \leftarrow$ CEIL($numHubs/VtxPerLine$)
13:     $PackingFactor \leftarrow hubWSet\_Original/hubWSet\_Sorted$
    return $PackingFactor$
```

The next two subsections describes the main contributions of the two papers under consideration.

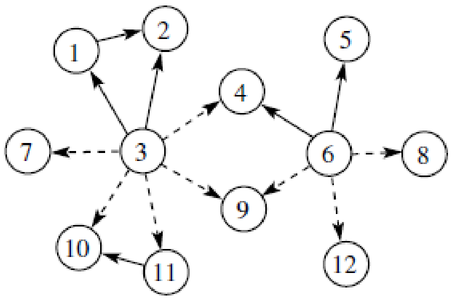
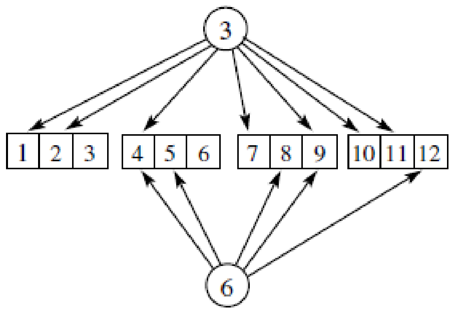
3.2 Hao et al. : Speedup Graph Processing by Graph Ordering

Graph partition algorithms are possible solutions to the problem of reducing the cache miss ratio. Such algorithms divide the graph into partitions such that edge-cut is minimized (edge-cut is the number of edges that cross different partitions). But this approach has got two primary drawbacks. Firstly, real graphs don't have good edge cuts as shown in [29] due to the power-law degree distribution and existence of huge nodes. Secondly, it is difficult to determine partition size since the cache line size is small and fixed. Moreover the way of data alignment in memory may be different from the partitions allocated. Graph ordering offers better performance over partitioning as can be seen from the following example:

Consider the input graph shown in the following figure:

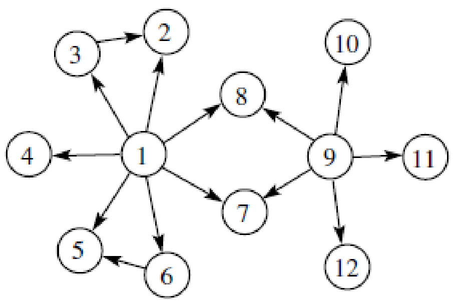
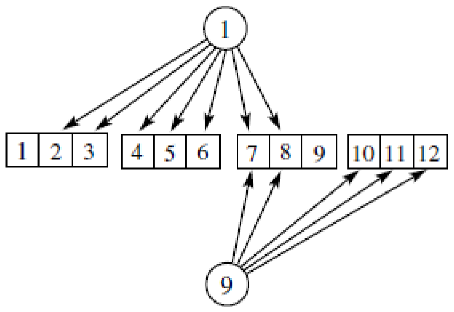


After graph partitioning, the optimal partition looks like the following 2 figures:

In the figure on the left, the edges shown as dashed lines are the edges that cross two partitions to be cut. The figure on the right shows the four resulting partitions. It can also be seen that if an algorithm accesses the out-neighbors of node 3, it needs to access all four partitions which are stored across four CPU cache lines. Similarly, for node 6, there is a need to access three partitions stored across three cache lines.

After graph reordering, the optimal representation looks like the following 2 figures:

In case of reordering, if an algorithm tries to access the out-neighbors of node 1 (topologically the same node as node 3 from the partitioning case), it needs to access three partitions which are stored across three CPU cache lines. Similarly, for node 9 (topologically the same node as node 6 from the partitioning case), there is a need to access just two partitions stored across two cache lines. The reduction in latency from partitioning to ordering is quite apparent.

The following is a detailed description of the algorithm:
A common pattern that can be found in graph computing algorithms is as follows:

```
1: for each node $v \in N_O(u)$ do
2:     the program segment to compute/access $v$
```

In the above statement, it can be observed that both the neighbor and sibling type of relationships need to be taken into account. For the purpose of the algorithm, a metric is defined that measures the closeness of two nodes in terms of locality. For two nodes u and v, the scoring function is given by:

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

Here, $S_s(u,v)$ is the number of the times that u and v co-exist in sibling relationships, which is the number of their common in-neighbors. And $S_n(u,v)$ is the number of times that u and v are neighbors, which is either 0, 1, or 2.
The sliding window model which is used in the algorithm can be understood as follows: 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.

Now that the groundwork is laid, the problem statement boils down to the following: Find the optimal graph ordering $\phi()$, that maximizes *Gscore* (the sum of locality score), F(·), based on a sliding window model with a window size w, where

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

It can be observed that the above problem with a window size of 1 reduces to the maximum traveling salesman problem [2], henceforth abbreviated as maxTSP. In general, the problem can be thought of as a variant of the maxTSP problem with a window size w instead of

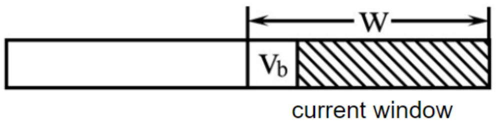
1. The maxTSP-w problem is solved by constructing an edge-weighted complete undirected graph G_w from the original graph G where the vertex set of G_w is the same as G and since it is a complete graph, there is an edge between every pair of nodes in G_w. The weight of an edge in G_w is the score of the 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. Here, instead of the construction of G_w, the best-neighbor heuristic idea that is used to solve maxTSP is used to solve maxTSP-w. In the basic algorithm proposed, the 3 inputs are the graph G, the window size w and the scoring function S.

The algorithm GO (Graph Ordering) can be summarized as follows:

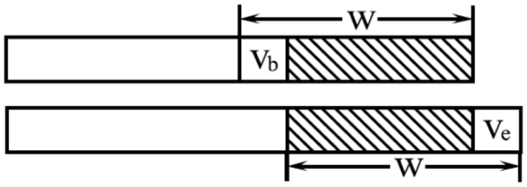
1. Start with a random node and insert it into the permutation.
2. In the i-th iteration, for node v that hasn't yet been inserted, the score is given by

$$k_v = \sum_{j=max\{1,i-w\}}^{i-1} S(v_j, v)$$

where v_j is the node inside the current window set.

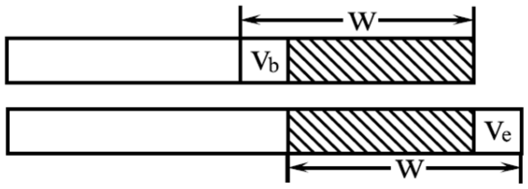


3. Choose the node v_{max} (v_e in the figure below) having the largest locality score with the window set. Append it at the end of permutation.
4. Slide the window set.
5. Repeat the above two steps iteratively until every node has been chosen.



The best-neighbor heuristic that the algorithm adopts instead of constructing the complete graph can give 0.5 – approximation for maxTSP. It is shown that the above algorithm can give $1/2w$ – approximation for graph ordering with window size w.

The time complexity of the GO algorithm is $O(wd_{max}n^2)$ which is very high. To remedy this a variant named GO-PQ using a priority queue was proposed. The algorithm can be summarized as follows:

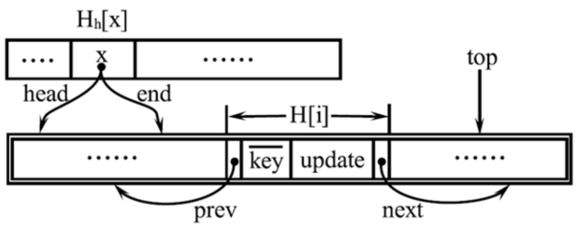


1. Use a priority queue to keep the locality score for every node.

2. Update the locality score incrementally (in $O(log(n))$ time by traditional priority queue).

3. Pop the node with the largest locality score.

Due to the use of this particular data structure, access is fast, that is, Node $v_i s$ locality score is stored in the i-th position of the array, can be accessed in $O(1)$ time. A doubly linked list with decreasing key by prev and next pointer. The node position is adjusted by prev and next in $O(1)$ time. A top pointer points to the node having largest key as shown in the figure below:



The data structure is maintained as followed,

$$\begin{aligned} \mathsf{update}(\mathsf{top}) &= 0 \\ \mathsf{update}(v_i) &\leq 0 \quad \text{for } v_i \neq \mathsf{top} \\ \overline{\mathsf{key}}(\mathsf{top}) &\geq \overline{\mathsf{key}}(v_i) \end{aligned}$$

The above three conditions make sure the top node is the one having the largest locality score. That is,

$$\overline{\mathsf{key}}(\mathsf{top}) + \mathsf{update}(\mathsf{top}) \geq \overline{\mathsf{key}}(v_i) + \mathsf{update}(v_i)$$

The priority queue based algorithm can compute graph ordering in $O(\sum_{u \in V}(d_o(u))^2)$ time complexity with space consumption linear with the size of node set, |V(G)|.

Evaluation:

1. The GO-PQ algorithm was pitted against the following graph orderings: Original, MINLA [10], MLOGA, RCM, DegSort, CHDFS, SlashBurn [27], LDG, and METIS [18].

2. The graph algorithms being tested are Neighbors Query (NQ), Breadth-First Search (BFS) [13], Depth-First Search (DFS) [13], Strongly Connected Component (SCC) detection [1], Shortest Paths (SP) by the Bellman-Ford algorithm [14], PageRank (PR) [31], Dominating Set (DS) [9], graph decomposition (Kcore) [3] and graph diameter (Diam).

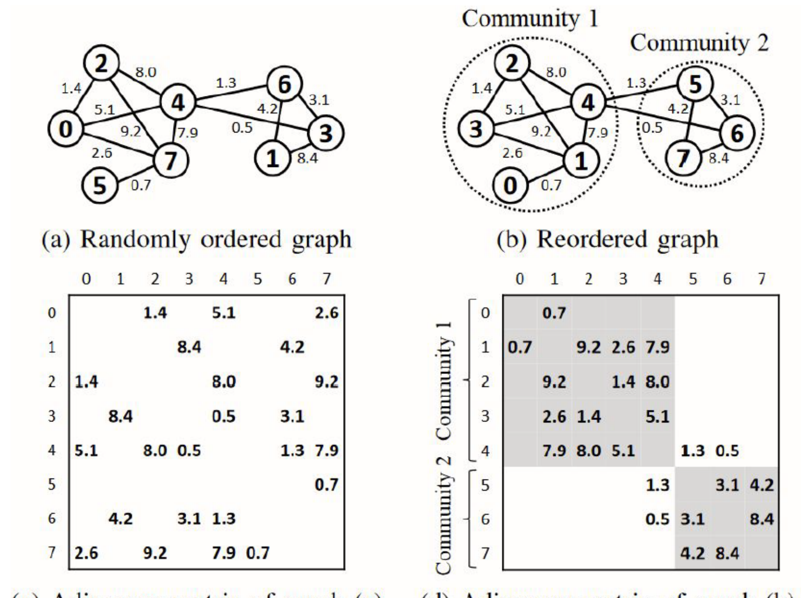
3. The experiments were carried on 8 real graph datasets (like Pokec, Flickr, Google+, Twitter etc).

3.3 Balaji et al. : When is Graph Reordering an Optimization?

This paper attempts to answer the above question by looking at the search space described by 4 graph ordering techniques (Gorder [28], Rabbit Ordering [24], Frequency based clustering (or Hub Sorting [19], and Hub Clustering applied on 8 real graphs as input used by 11 applications from the GAP [16] and Ligra [15] benchmark suites.

The Lightweight Graph ordering techniques that are considered in this paper are briefly described below:

- Rabbit Order: This approach does fast community detection using incremental aggregation by providing a mapping between the hierarchically dense communities in graphs and different levels of the cache hierarchy; wherein, the smaller, denser communities are mapped to caches closer to the processor. This is shown in the example below:



(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		

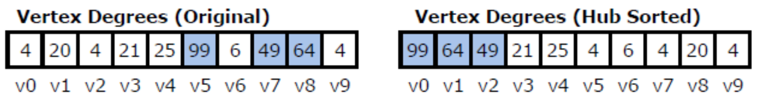
(c) Adjacency matrix of graph (a)

		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)

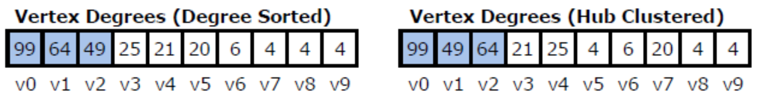
The complexity is given by $O(|E| - ck|V|)$ where c is the clustering coefficient and k is the average degree of the vertices.

- Hub Sorting: Here, 'hub vertices' (vertices with degree more than the average degree) are relabeled in descending order of degrees while keeping the vertex id assignment of most non-hub vertices intact. The spatial and temporal locality of power law graphs are improved as can be seen from the following example:

  
  Vertex Degrees (Original)
4 20 4 21 25 99 6 49 64 4
v0 v1 v2 v3 v4 v5 v6 v7 v8 v9
Vertex Degrees (Hub Sorted)
99 64 49 21 25 4 6 4 20 4
v0 v1 v2 v3 v4 v5 v6 v7 v8 v9

  The complexity is given by $O(|V|.logV)$.

- Hub Clustering: This lightweight reordering technique is the author's variant of the Hub Sorting technique where hub vertices are assigned contiguous range of ids. The difference from Hub sorting is that the ids are not assigned by decreasing order of degree. Hence, this provides improvement in just temporal locality because high-reuse hub vertex data is tightly packed. Since it doesn't sort the hub vertices, its overhead is lower than Hub sorting. An example is shown below:

  
  Vertex Degrees (Degree Sorted)
99 64 49 25 21 20 6 4 4 4
v0 v1 v2 v3 v4 v5 v6 v7 v8 v9
Vertex Degrees (Hub Clustered)
99 49 64 21 25 4 6 20 4 4
v0 v1 v2 v3 v4 v5 v6 v7 v8 v9

  The complexity is given by $O(|V|)$.

The findings of the exercise carried out to identify the type of application that benefits from Lightweight reordering are as follows:

1. Applications that process a large fraction of edges were shown to be the biggest beneficiary of Lightweight reordering.

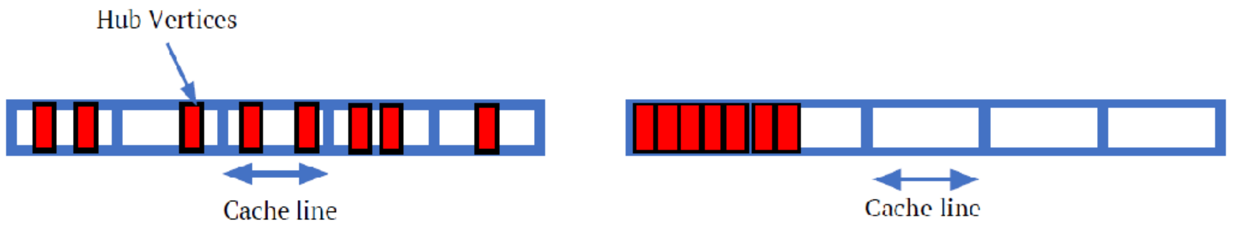
2. The trend for symmetric bipartite graphs show that Hubsort causes a net slowdown. This is because, vertices from different parts are assigned consecutive IDs which increases the range of irregular access to the neighbouring vertex u from a vertex v.

3. Push style applications, (that is, applications where a vertex is processed first and then the vertices in its out-neighbor set are processed) or those that process a small fraction of edges per iteration are inappropriate for lightweight reordering because of false sharing.

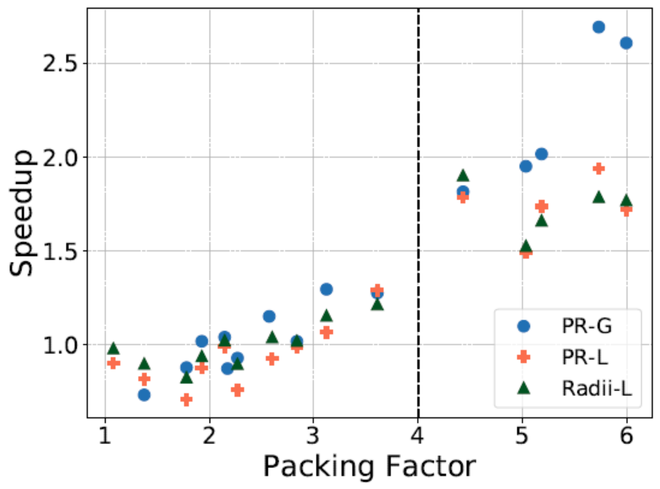
From the experiments, it was also observed that whenever Hub Sorting is deemed effective for some application, its benefits is dependent on the input graph. This results in the next question being asked to be: Which input graphs benefit from reordering? The answer to this question is provided by 'selectively' applying Hubsort reordering. This process of Selective Lightweight Graph Reordering is as follows:

1. The properties exhibited by graphs that make them suitable for hub sorting are:
   - Skew in degree distribution: This indicates the presence of hubs.
   - The presence of hub vertices in the graph should be sparse.

   Such an example of a graph that benefits from hub sorting is shown below where the figure on the left shows the layout of the hub vertices in the original ordering and the one on the right shows the ordering after hub sorting. It can be observed that the hub vertices now are distributed across a significantly lesser number of cache lines.



2. In order to identify graphs that exhibit the above mentioned properties, Packing Factor is used as the indicator. In order to identify the range of Packing Factor for which Hubsort is beneficial, the Speedup observed from the experiments is plotted against the Packing Factor. It is observed that there is a string correlation (more than 90%) between the two. The empirically obtained conclusion is that graphs with a Packing Factor less than 4 doesn't show significant speedup from Hubsort as can be seen in the figure below:



3. Depending on the Packing Factor (empirically obtained threshold is 4), the graph is either Hub sorted before processing, or processed directly. It is observed that Selective Reordering avoids slowdown and computing Packing Factor doesn't degrade performance.

4 Compare and Contrast

This sections aims to identify the key similarities and dissimilarities between the two approaches. This is followed by a critical evaluation of the two methods (drawbacks and strengths) and some suggestions for improvement.

4.1 Similarities

The similarities between the two approaches are looked at below:

- Both approaches try to maximize locality by decreasing cache miss ratio for graph applications.
- Another similarity between the two approaches is that both of them try to exploit the topology of the graphs (power-law degree distribution) in order to achieve the goal.
- The intuition for both approaches is too preserve the spatial locality of the vertices. Simply put, the tendency is to keep the neighbors of a vertex close, while storing in

the memory.

- Both approaches attempts to reassign IDs to the vertices, thereby producing a linear ordering which is then used while storing them in memory.

4.2 Differences

The primary differences between the two approaches are as follows:

- The metric/score that is the main focus of Gorder, takes into account the neighbor and sibling relationship between two vertices. Packing Factor, on the other hand, tries to assign hub vertices to cache lines based on their sparsity.
- Gorder is application agnostic and also input graph independent. This is completely opposite for the other approach because it is application-specific and also dependent on the nature of the input-graph.
- Gorder uses the priority queue data structure to increase efficiency while the other tries to exploit the hierarchical nature of cache.
- The first one tries to approximately solve an NP-hard problem and evaluate the results empirically while the second method itself is developed empirically.

4.3 Critical Evaluation

An evaluation of the two methodologies are as follows:

- Gorder fails to take advantage of the various characteristics of the input graphs as well the applications which those graphs as input. On one hand, this ensures that it generalizes well. But the overhead cost limits its application to only those cases where the input graphs are reused enough to amortize the overhead cost.
- Selective Graph ordering triumphs where Gorder fails. But, since the whole methodology was empirically derived, there is a chance it might overfit to the type of applications and input graphs that were used in the experiments.

Suggested improvement and future scope:

- Gorder might benefit from using a different scoring function which is more descriptive and is complex enough to capture either the characteristics of the graph processing application or the input graph or both.
- For Selective Ordering, Packing Factor seems like a good indicator of whether the input graph will benefit from lightweight reordering. But this is valid only for the set of input graphs that was considered for the experiments. There might exist other metrics that generalize better to a larger set of input graphs. In other words, for unseen input graphs, Packing Factor might misclassify it as unsuitable for lightweight reordering and vice versa.
- Loosening the bound given by the Gorder method thereby making it computationally lightweight (The 'loosening' will come from not computing the score for every pair of nodes in the sliding window).

5 Conclusion

This report takes a look at two approaches to vertex ordering that aims at minimizing graph processing time by reducing cache miss ratio. To that end, it gives a brief description of the two methodologies and identifies some pros and cons for both. Additionally, it suggests some scope for further research which might result in addressing the disadvantages identified.

References

[1] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. Computers & Mathematics with Applications, 7(1), 1981.

[2] A. I. Serdyukov. An algorithm with an estimate for the traveling salesman problem of the maximum. Upravlyaemye Sistemy, 25:80–86, 1984.

[3] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. CoRR, cs.DS/0310049, 2003.

[4] U. Kang and C. Faloutsos. Beyond ‘caveman communities’: Hubs and spokes for graph compression and mining. In Proc. of ICDM’11, 2011.

[5] A. Mukkara, N. Beckmann, D. Sanchez, ”Cache-Guided Scheduling: Exploiting caches to maximize locality in graph processing”, AGP’ 17, 2017.

[6] J. Banerjee, W. Kim, S. Kim, and J. F. Garza. Clustering a DAG for CAD databases. IEEE Trans. Software Eng., 1988.

[7] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In Proc. of VLDB’99, 1999.

[8] U. Meyer and P. Sanders, “Delta-stepping: A parallel single source shortest path algorithm,” in Proceedings of the 6th Annual European Symposium on Algorithms, ESA ’98, (London, UK, UK), pp. 393–404, Springer-Verlag, 1998.

[9] E. Cockayne. Domination of undirected graphs - a survey. In Theory and Applications of Graphs, pages 141–147. Springer, 1978.

[10] J. Petit. Experiments on the minimum linear arrangement problem. Journal of Experimental Algorithmics (JEA), 2003.

[11] A. O. Mendelzon and C. G. Mendioroz. Graph clustering and caching. In Computer Science 2. Springer, 1994.

[12] J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution: Densification and shrinking diameters,” ACM Trans. Knowl. Discov. Data, vol. 1, Mar. 2007.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms. MIT press Cambridge, 2 edition, 2001.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms. MIT press Cambridge, 2 edition, 2001.

[15] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory,” in ACM Sigplan Notices, vol. 48, pp. 135–146, ACM, 2013.

[16] S. Beamer, K. Asanovic, and D. Patterson, “Locality exists in graph processing: Workload characterization on an ivy bridge server,” in Workload Characterization (IISWC), 2015 IEEE International Symposium on, pp. 56–65, IEEE, 2015.

[17] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. ”Making caches work for graph analytics”. In 2017 IEEE International Conference on Big Data (Big Data). 293–302

[18] metis G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. J. Parallel Distrib. Comput., 48(1), 1998.

[19] Y. Zhang, V. Kiriansky, C. Mendis, M. Zaharia, and S. Amarasinghe, “Optimizing cache performance for graph analytics,” arXiv preprint arXiv:1608.01362, 2016.

[20] J. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. IEEE Trans. Parallel Distrib. Syst., 15(9), 2004.

[21] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzar´an, and K. Pingali, “Parallelization of reordering algorithms for bandwidth and wavefront reduction,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 921– 932, IEEE Press, 2014.

[22] P. Boldi, M. Santini, and S. Vigna. Permuting web graphs. In Proc. of WAW’09, 2009.

[23] Chevalier, Cédric, and François Pellegrini. ”PT-Scotch: A tool for efficient parallel graph ordering.” Parallel computing 34, no. 6-8 (2008): 318-331.

[24] Arai, Junya, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. ”Rabbit order: Just-in-time parallel reordering for fast graph analysis.” In 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 22-31. IEEE, 2016.

[25] L. Auroux, M. Burelle, and R. Erra. Reordering very large graphs for fun & profit. In International Symposium on Web AlGorithms, 2015.

[26] Scotch: Static mapping, graph partitioning, and sparse matrix block ordering package, http://www.labri.fr/ pelegrin/scotch/.

[27] Y. Lim, U. Kang, and C. Faloutsos, “Slashburn: Graph compression and mining beyond caveman communities,” IEEE Transactions on Knowledge and Data Engineering, vol. 26, no. 12, pp. 3077–3089, 2014.

[28] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In Proceedings of the 2016 International Conference on Management of Data,Journal of molecular biology, pp. 1813-1828. ACM, 2016.

[29] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In Proc. of WWW’08, 2008.

[30] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. PVLDB, 8(4), 2014.

[31] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[32] H. Kwak, C. Lee, H. Park, and S. Moon, “What is Twitter, a social network or a news media?,” in WWW ’10: Proceedings of the 19th international conference on World wide web, (New York, NY, USA), pp. 591–600, ACM, 2010.

[33] V. Balaji, B. Lucia. When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs.In 2018 IEEE International Symposium on Workload Characterization (IISWC), pp. 203-214. IEEE, 2018.