Graph Ordering (or reordering): A survey
CptS 575: Data Science

Reet Barik, WSU ID: 11630142

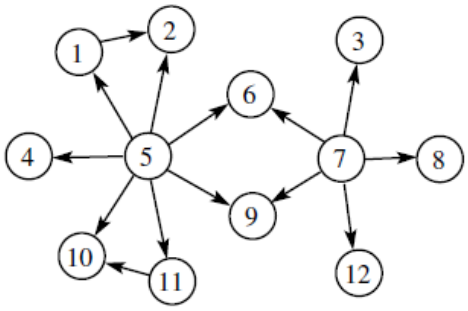
Abstract

For most parallel graph-theoretic applications, data movement has become a primary bottleneck towards 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.

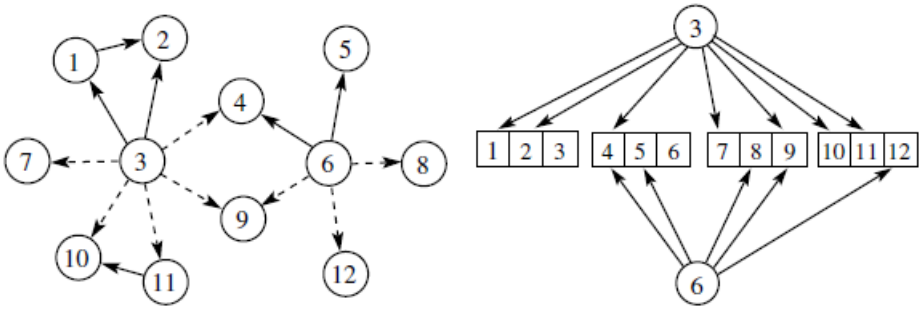
I. Introduction

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 [9]. 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*. This survey aims to look at different graph ordering techniques from the point of view of preserving the locality of vertices, thereby reducing the cache miss ratio.

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 [36] 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:

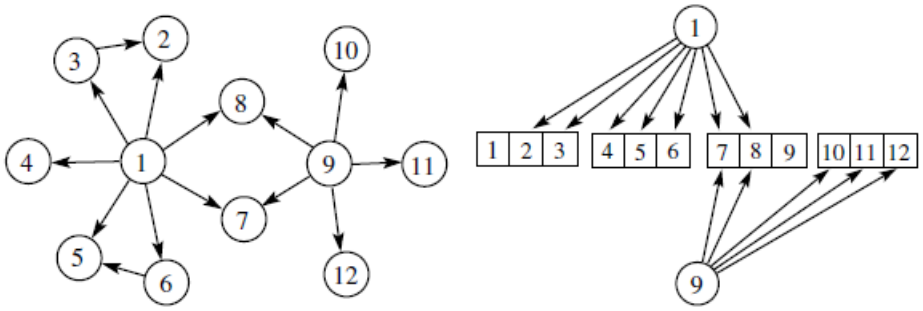


After graph partitioning, the optimal partition looks like the following 2 figures:



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.

Graph ordering techniques can be broadly classified into two categories. The primary approach, like 'Gorder' [35], aims to increase CPU speedup for graph computing by reducing the cache miss ratio for different graph algorithms. 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.

Sophisticated reordering techniques such as Gorder, 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 take input graphs which are dynamic in nature (like Page ranks in dynamically changing social networks [39] or tracking changes in the diameter of an evolving graph [15]), the reordering overhead is far too high for the reordering techniques to be viable. This begs the question of the type of properties that are

exhibited by graph applications and input graphs for which graph reordering is an optimization. Hence, there is also a need for a second approach where the reordering step is computationally lightweight at the cost of not being agnostic in terms of input graphs or end applications.

II. Related Works

Apart from Gorder, there are numerous graph ordering techniques that have existed in the literature for some time. PT-Scotch is a tool [29] for efficient parallel graph ordering. It extends Scotch [33] 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 [30] 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 [6] 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 [20] is a cache-optimized in-memory graph framework that uses a new technique called 'CSR segmenting'.

There are approaches like Park [26] and Then [37] which are applicable for specific graph applications only. Banerjee [7] improves the efficiency of the DAG traversal by proposing a node ordering scheme by the children-depth-first traversal method. Mendelzon [14] 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 [32] reorders the node set by BFS [16]. Kang [5] removes the hub nodes iteratively from the graph using a graph decomposition approach. Boldi [28] computes the clusters of the graph by label propagation and graph compression techniques. Metis [21] is a set of programs that implement multilevel graph partitioning algorithms. Karantasis [27] attempts to parallelize two popular reordering algorithms, RCM and Sloan. Petit [13] attempts to empirically approach this problem in the Minimum Linear Arrangement (minLA) setting.

III. Literature Review

The paper *Petit: Experiments on the Minimum Linear Arrangement Problem* [13], looks at the graph ordering problem in the Minimum Linear Arrangement setting from the experimental point of view. It attempts to empirically derive the upper and lower bounds of this problem by comparing the performance of several algorithms that can be reduced to the minLA problem.

Several methods for obtaining the lower bound of the minLA problem are reviewed: Path method [22], Gomory–Hu Tree method [23], and Juvan–Mohar method [22]. Several methods are also proposed by the author: Edges method, Degree method, and Mesh method. Approximate solutions (and hence the upper bound) for the minLA problem are also investigated empirically. This is done using several 'successive augmentation heuristics' such as Spectral Sequencing [22], Local Search heuristics [8], Hillclimbing, Metropolis heuristic [12], and Simulated Annealing [24]. The lower and upper bounding techniques thus obtained have been implemented in the "llsh toolkit" which is essentially a library of methods for the minLA problem with a C++ core.

While the above paper investigated the bounds of this particular NP-Hard problem, there are numerous other works that came up with algorithmic contributions to approximately solve the problem. The rest of this sections aims to provide a brief description of some of those works and contribution in the field of graph ordering. They are as follows:

A. *Hao et al. : Speedup Graph Processing by Graph Ordering [35]*

A common pattern that can be found in graph computing algorithms is as follows:

```
1: for each node v ∈ 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($\cdot$), based on a sliding window model with a window size w, where

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

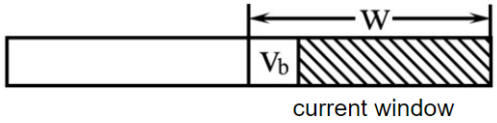
It can be observed that the above problem with a window size of 1 reduces to the maximum traveling salesman problem [3], henceforth abbreviated as maxTSP. In general, the problem can be thought of as a variant of the maxTSP problem with a window size w instead of 1. The maxTSP-w problem is solved by 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.

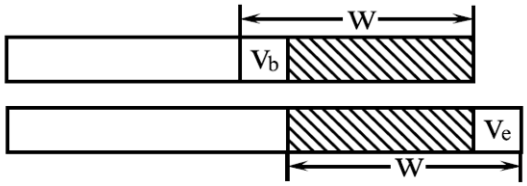
2. In the i-th iteration, for node v that hasn't yet been inserted, the score is given by

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

where v_j is the node inside the current window set.

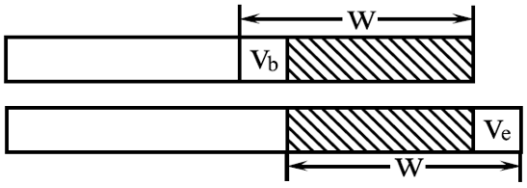


3. Choose the node v_{max} (v_e in the figure below) having the largest locality score with the window set. Append it at the end of permutation.
4. Slide the window set.
5. Repeat the above two steps iteratively until every node has been chosen.



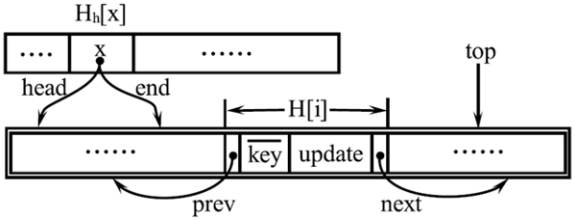
The best-neighbor heuristic that the algorithm adopts instead of constructing the complete graph can give 0.5 – approximation for maxTSP. It is shown that the above algorithm can give $1/2w$ – approximation for graph ordering with window size w.

The time complexity of the GO algorithm is $O(w.d_{max}.n^2)$ 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 shown below,

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

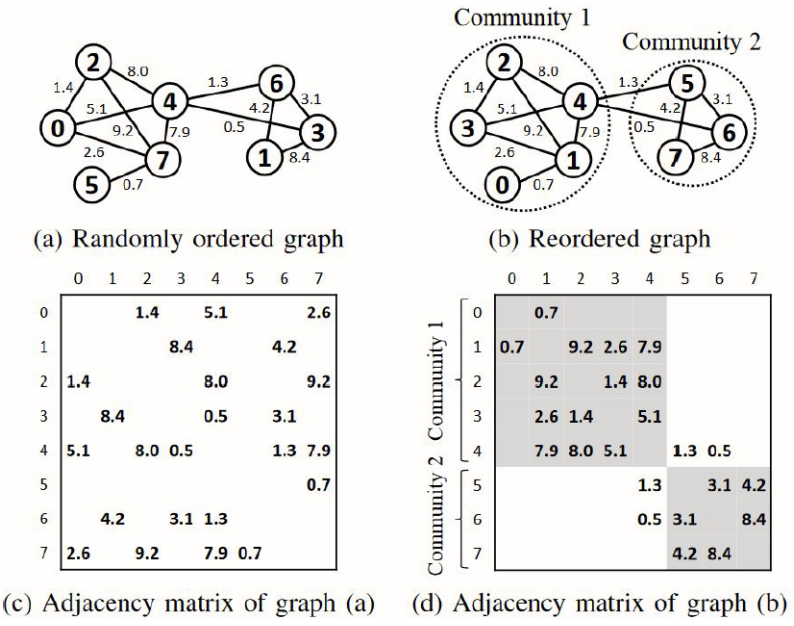
The above three conditions make sure the top node is the one having the largest locality score. That is,

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

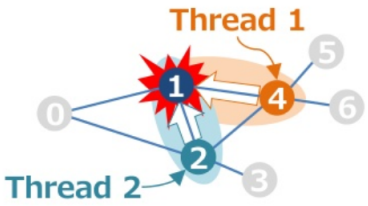
The priority queue based algorithm can compute graph ordering in $O(\sum_{u \in V}(d_o(u))^2)$ time complexity with space consumption linear with the size of node set, $|V(G)|$.

B. Arai et al. : Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis [30]

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:

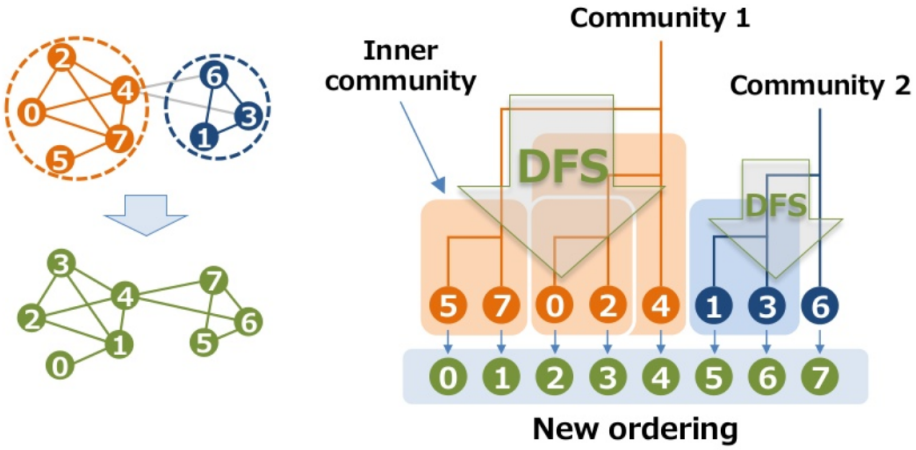


The hierarchical community structure detection is done by 'Parallel implementation aggregation'. This means that vertex pairs are merged, thereby extracting hierarchical communities. This process is fast since it rapidly coarsens the graph but is sequential by nature. Naive parallel implementations result in conflicts as shown below:



Such conflicts are solved by lazy aggregation where lightweight concurrency control is done by atomic operations. Merges are delayed until the merged vertex is required. As shown in the example below, all the vertices in a community are labelled as a member of that community. After all the vertices have been labelled, the ones with the same labels are merged.

After this is done, a dendogram is constructed while extracting communities. The vertices are then reordered using DFS visit order to generate the final ordering as shown:



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

C. Balaji et al. : When is Graph Reordering an Optimization? [40]

This paper attempts to answer the question in the title by looking at the search space described by 4 graph ordering techniques (Gorder [35], Rabbit Ordering [30], Frequency based clustering (or Hub Sorting [25]), and Hub Clustering) applied on 8 real graphs as input used by 11 applications from the GAP [19] and Ligra [18] benchmark suites.

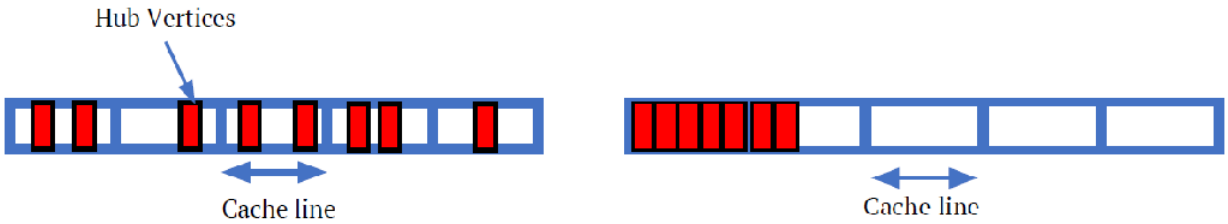
The findings of the exercise carried out to identify the type of application that benefits from Lightweight reordering are as follows:

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.

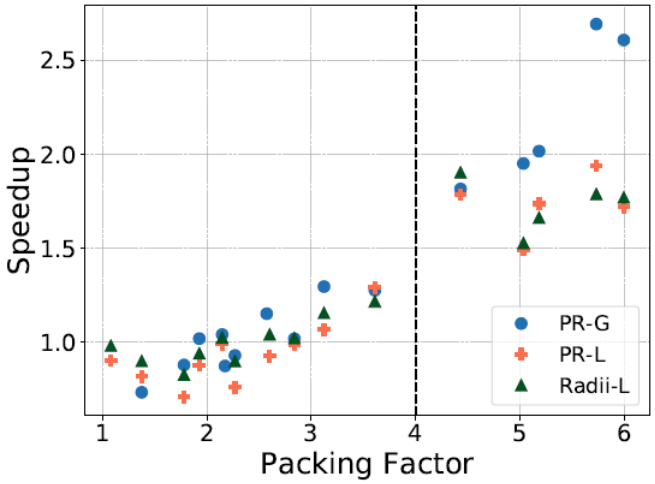


2) In order to identify graphs that exhibit the above mentioned properties, a metric called the 'Packing Factor' is used as the indicator. The idea is to quantify the decrease in sparsity of the hub vertices from the point of view of cache lines. It is essentially the ratio of the original graph's hub 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 ← 0
3:      hubWSet_Original ← 0
4:      for CacheLine in vDataLines do
5:          containsHub ← False
6:          for vtx in CacheLine do
7:              if ISHUB(vtx) then
8:                  numHubs += 1
9:                  containsHub ← True
10:         if containsHub = True then
11:             hubWSet_Original += 1
12:     hubWSet_Sorted ← CEIL(numHubs/VtxPerLine)
13:     PackingFactor ← hubWSet_Original/hubWSet_Sorted
    return PackingFactor
```

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:



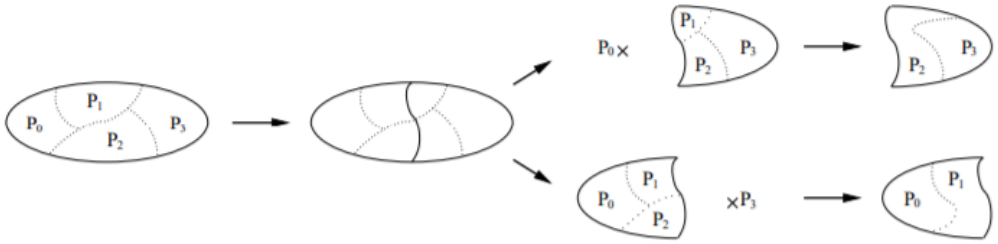
3) Depending on the Packing Factor (empirically obtained threshold is 4), the graph is either Hub sorted before processing, or processed directly. It is observed that Selective Reordering avoids slowdown and computing Packing Factor doesn't degrade performance.

D. Chevalier et al. : PT-Scotch: A tool for efficient parallel graph ordering [29]

Efficient parallel reordering is done in three different levels of concurrency. The three steps are described below:

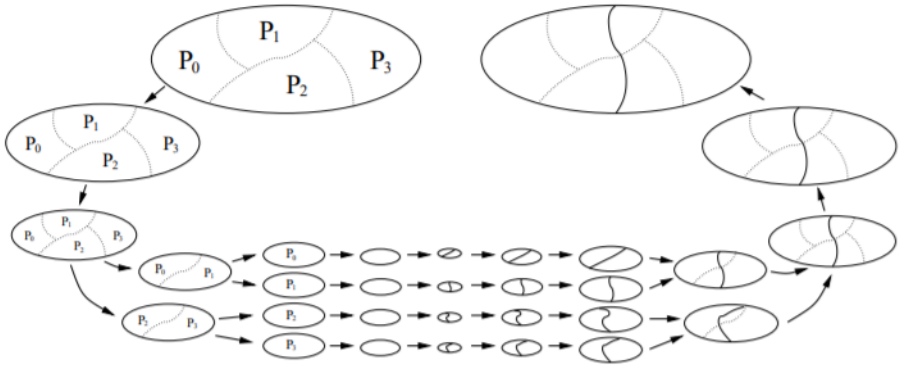
1) Nested dissection: The initial graph is arbitrarily distributed across process (preferably balanced in terms of vertices). A distributed induced subgraph is built once a separator has been computed in parallel corresponding to the first separated part. For example, a

nested dissection step for a (sub-)graph distributed across four processes is shown below:

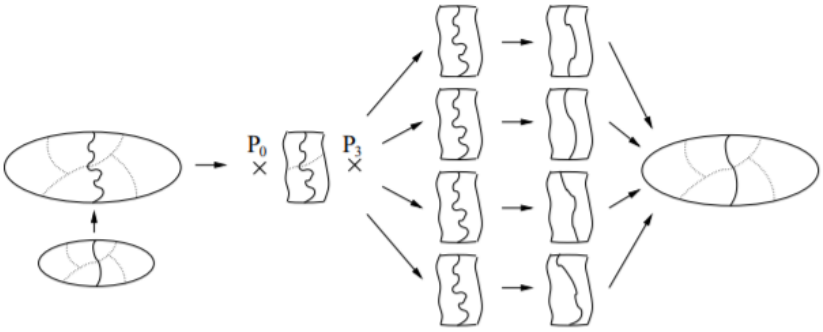


These induced subgraphs are then folded onto each other such that, at the end of the folding phase, every process has a sub graph fragments of one of the two folded subgraphs and the nested dissection algorithm can recursively proceed independently on each subgroup of processes until each subgroup is reduced to a single process.

2) The second level of concurrency deals with the computation of separators. A set of increasingly coarser but topologically similar versions of a graph are put through the computation of collapsing edges and vertices in order to obtain coarser and coarser graphs until the coarsest graph obtained is no larger than a few hundred vertices. A vertex spearator on this graph is computed and is projected back on to the finer graphs untill it is finally projected on to the original graph. A variation of the Fiduccia-Mattheyses [1] is used during the uncoarsening phase. A diagram of the parallel computation of a graph across four processes is shown below:



3) Band refinement is the third level of concurrency which deals with improving the projected separators. This is done by creation of a distributed band graph from a projected partition. Once the separator of the distributed band is refined, it is projected back on to the original graph. A diagram showing the above process executed across four processes is shown below:



E. Karantasis et al. : Parallelization of Reordering Algorithms for Bandwidth and Wavefront Reduction [27]

This paper proposes a parallel version of the Reverse Cuthill-McKee [31] algorithm which is used to produce a permutation ordering of nodes in a graph. RCM is very challenging to parallelize because it places additional restrictions on valid permutations. This challenge has been surmounted in two ways:

1) Leveled Reverse Cuthill-McKee: This first approach is an incremental one, wherein, at each major step of the algorithm, the permutation for the nodes seen so far is calculated and does not change in subsequent steps. The 'expansion step' expands a list of nodes in parallel and these 'parent's examines their neighbor to find their 'child' nodes. The 'reduction' step computes the number of children for each parent. The 'prefix sum' step computes an index in the permutation array for each child while the 'placement' step actually places the child in the permutation array. The algorithm is shown below:

Algorithm 5 Leveled RCM

```
1   Graph G = input() // Read in graph
2   P[0] = source;
3   while (P.size < G.size){
4     // Expansion
5     List children;
6     foreach (Node parent: P[parent1:parentN]){
7       for (Node nb: parent.neighbors) {
8         if (nb.level <= parent.level)
9           continue;
10        //Atomic check with compare_and_swap
11        if (nb.level > parent.level + 1){
12          atomic nb.level = parent.level + 1;
13          children.push(child);
14        }
15        if (parent.order < child.parent.order)
16          atomic child.parent = parent;
17    } }
18    // Reduction
19    foreach (Node child: children) {
20      atomic child.parent.chnum++;
21    }
22    // Prefix Sum
23    foreach (int thread: threads) {
24      // Prefix sum of parent.chnum into parent.index
25    }
26    // Placement
27    foreach (Node child: children) {
28      atomic index = child.parent.index++;
29      P[index] = child;
30      if (child == child.parent.lastChild)
31        sort(P[children1:childrenM]);
32    } }
```

2) Unordered Reverse Cuthill-McKee: This approach uses an a-posteriori approach that builds an RCM-valid permutation right after a complete level structure is computed. It computes BFS levels, counts the number of nodes at each level, and computes the prefix sum of levels. The placement phase is more strict in this approach. A single thread is assigned to each level giving it a pipeline structure. As a result, a thread receives the nodes in the correct order for its own level before passing that order on to the next level. The algorithm is shown below:

Algorithm 6 Unordered RCM

```
1   Graph G = input() // Read in graph
2   Unordered_BFS(Graph G, source)
3   counts = Count(G) // Same as for BFS
4   sums[1:] = prefixSum(counts)
5   sums[0] = 0
6   place(G, source, max_dist, sums);
7
8   void place(Graph G, Node source, int max_dist, Array sums) {
9     read_offset = sums;
10    write_offset = sums;
11    perm[0] = source;
12    write_offset[0] = 1;
13
14    foreach (int thread: threads) {
15      for (int l = thread; l < max_dist; l += threads.size()) {
16        while (read_offset[l] != sums[l + 1]) {
17          while (read_offset[l] ==
18                  write_offset[l]) { } // Spin
19          Node N = perm[read_offset[l]];
20          ++read_offset[l];
21          children = // Edges of N with dist == l + 1
22          sort(children) // Sort children by degree
23          for (Node c: children) {
24            perm[write_offset[l+1]] = c;
25            ++write_offset[l+1];
26  } } } } }
```

IV. Conclusion

A summary review of the works from the previous section and the broad approaches in general is as follows:

- The paper by Petit attempts to come up with bounds for the NP-hard minLA problem by empirically evaluating the performance of different algorithms that can be reduced to the minLA setting. The library, which was the end result of this work serves as a good benchmark to evaluate future algorithmic contributions.
- Both heavyweight (Gorder) and lightweight (Rabbit Order, Hubsort etc.) 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 to 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.
- The algorithm Gorder fails to take advantage of the various characteristics of the input graphs as well as the applications, which take 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.
- Rabbit Order and PT-Scotch offers a way to increase the scalability by the virtue of the algorithm being parallelizable. But the tradeoff in the former comes at the expense of degrading community quality and in the long run also the reordering quality.
- 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.
- The paper by Karantasis et al. deals with the parallelization of the algorithm RCM and Sloan only. Those two algorithms are best suited for graphs which can be represented as a sparse matrix with a symmetric sparsity pattern and often times not feasible for real world graphs.

Some improvements and scope for future research are suggested as follows:

- The minLA library could be updated to incorporate parallelized versions of the existing algorithms. This could be used as a benchmark for future contributions in the field of parallelized graph ordering algorithms.
- 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.
  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).
- 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.
- Instead of going about trying to solve the problem from the algorithm point of view, one might investigate exploring different data layouts (new data structures) that can take advantage of existing reordering algorithms better than traditional ones.

ACKNOWLEDGMENT

The author would like to thank Professor Assefaw Gebremedhin (Instructor, CPTS 575 Data Science) for the opportunity to work on an interesting topic as graph ordering. The author is also grateful for the guidance and inputs of Professor Ananth Kalyanaraman (Master's Non-thesis Advisor) during the course of writing the report.

REFERENCES

[1] C. M. Fiduccia, R. M. Mattheyses, A linear-time heuristic for improving network partitions, in: Proc. 19th Design Automation Conference, IEEE, 1982, pp. 175–181.
[2] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. Computers & Mathematics with Applications, 7(1), 1981.
[3] A. I. Serdyukov. An algorithm with an estimate for the traveling salesman problem of the maximum. Upravlyaemye Sistemy, 25:80–86, 1984.
[4] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. CoRR, cs.DS/0310049, 2003.

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

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

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

[8] C. Papadimitrou and K. Steiglitz. Combinatorial Optimization, Algorithms and Complexity. Prentice Hall, 1982.

[9] 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.

[10] 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.

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

[12] W. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of State Calculations by Fast computing machines. The Journal of Chemical Physics, 21(6):1087–1092, 1953.

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

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

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

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

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

[18] 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.

[19] 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.

[20] 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

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

[22] M. Juvan and B. Mohar. Optimal linear labelings and eigenvalues of graphs. Discrete Applied Mathematics, 36(2):153–168, 1992.

[23] D. Adolphson and T. C. Hu. Optimal linear ordering. SIAM Journal on Applied Mathematics, 25(3):403–423, Nov. 1973.

[24] S. Kirkpatrik, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. Science, 220:671–680, May 1983.

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

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

[27] 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.

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

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

[30] 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.

[31] E. Cuthill and J. McKee. Reducing the Bandwidth of Sparse SymmetricMatrices.24th National Conference, 1969

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.