RLP: Random Label Propagation Graph Partitioning

Aadam
Ghulam Ishaq Khan Institute
Topi, Pakistan
Email: aadimator@gmail.com

Engr. Obaidullah
Ghulam Ishaq Khan Institute
Topi, Pakistan
Email: obaidul024@gmail.com

***Abstract*—Graph partitioning is an NP-hard problem where our goal is to partition the graph into approximately equal sizes while minimizing the number of edges between different partitions. In this paper, we propose a novel randomized linear-time algorithm for graph partitioning, called Random Label Partitioning (RLP), inspired from the label propagation algorithm for community detection. We run RLP on several graphs and show that it gives a relatively good modularity score while keeping the running time very low. The source code is available at https://github.com/aadimator/label-graph-partitioning.**

I. Introduction

Graphs are prevalent structures in computer science for representing a number of real-world systems, including but not limited to, social, biological, and information networks. The scale of such networks continues to grow with each passing day, especially in the domain of social networks. If we take a look at the modern World Wide Web, it hosts tens of billions of web-pages (nodes) with trillions of links (edges) between them. Facebook alone serves billions of monthly active users, plus hundreds of millions of pages, events, and groups, all interacting with each other through network structures. Similarly, Twitter sees hundreds of millions of monthly active users interact by sharing and liking each others content. In all these examples, graph-wide computations are central to the core functions of many products and services [1].

Unfortunately, large-scale computations are quite expensive and require a large amount of storage and computation resources. The graphs may even account for terabytes of compressed data and most computations over such large datasets are unmanageable for a single machine to perform in a reasonable amount of time. To solve this problem, large graphs are partitioned into smaller graphs, thus reducing the amount of load onto a single machine and enabling their parallel computing and storage. The problem of graph partitioning is in general defined as the partitioning of a graph into k groups of approximately equal sizes, minimizing the number of edges between groups [2].

This problem is NP-hard and several efficient heuristic methods have been developed over years to solve the problem. Some use greedy optimization techniques [3], [4], while others use approximate optimization techniques, like Spectral Clustering [5], [6] or Multi-commodity flow [7], some use heuristic algorithms like Multilevel graph partitioning (METIS) [8] while others try Randomized algorithms [9] to solve this NP-hard problem.

II. Case Study

Graph partitioning has been widely applied in task scheduling, cloud computing, data centers, hardware/software co-design, sparse linear system solving, virtual machine scheduling, and VLSI circuit design, etc [10]. The problem of partitioning graphs into equal-sized components while minimizing the number of edges between different components is extremely important in parallel computing [11]. We need to assign data/partitions equally among processors, while minimizing the communication overhead in order to take full benefit of the parallelization. Given a graph $G = (V, E)$, where nodes V represent data or tasks and edges E represent communication, our goal is to divide V into equal-sized parts $V_1, ..., V_n$ while minimizing the capacity of edges cut, where n is the number of available processors.

There have been numerous endeavors at parallelizing graph partitioning for large scale graphs on shared memory machines. These attempts have been ill-fated as these algorithms cannot be directly applied on distributed systems.

There are however some challenges to graph partitioning algorithms in parallel computing. First, traditional graph partitioning methods cannot well-preserve the global structural information of a graph on a single processor, thus it can reduce the quality and accuracy of the accumulated results. Secondly, for real world graphs and in particular scale free graphs, that are, graphs with common higher degree vertices than the average (they are characteristically robust and fault tolerant), it is difficult to create a balanced edge partitioning. Traditional methods assign all the edges of a vertices falling upon it to one processor, thereby severely affecting workload balance among processors, and can impair scalability.[12]

III. Our Solution

Throughout this paper we will consider a graph G such that $G = (V, E)$ is an undirected and unweighted graph. This graph has vertex set $V = v_1, ..., v_n$ and an edge set E with m undirected edges.

Our solution, Random Label Partitioning (RLP), is inspired from the label propagation algorithm [2] which was introduced for community detection in graphs. Although graph partitioning and community detection problems have a lot in common, one of the main differences between the two is that in community detection, we have to find k ourselves such that it reduces the number of cuts between different communities/partitions, while in graph partitioning problem,

k is given to us beforehand and we have to partition the graph into approximately equal sizes while minimizing the number of cuts between different partitions.

The approach that we took to solve the graph partitioning problem in Random Label Partitioning (RLP) algorithm is described in detail below. First, create a set of all the vertices/nodes in the graph, called unlabeled nodes $S = s_1, ..., s_n$. Then, select k random nodes from the unlabeled set, and label them from $p_1, ..., p_k$ respectively. Next, move these labeled nodes into a new set, labeled nodes P (i.e. remove the nodes from unlabeled set and add them into the labeled set). Then process the nodes in the unlabeled nodes set until it is empty, such that for every node p_i in the labeled set, select a random unlabeled neighbor s_i of that node, label it the same as that of the selected node i.e. p_i, and then move the newly labeled neighbor to the labeled set P. If all the neighbors of the node p_i are already labeled, move it to the finished set (we remove this node from the labeled set to reduce the running time and increase the efficiency of the algorithm). When the unlabeled set S is empty, it indicates complete traversal and labeling of all the nodes present in the graph. Once the loop finishes, return the labels of the nodes, specifying the partition, $p_1, ..., p_k$, that each node belongs to.

Algorithm 1: LabelPartition

```
input : A graph G, number of required partitions (k)
output: k Partitions of the graph
unlabeled ← Set(Vertices(G));
labeled ← Set();
finished ← Set();
labels ← Set(size(Vertices(G)));
k_vertices ← sample random k vertices from
 unlabeled;
for (i, v) in enumerate(k_vertices) do
    labels [v] = i;
    Move v to labeled;
end
while !empty(unlabeled) do
    for each node in labeled do
        neighbors ← unlabeled neighbors of node;
        if isempty(neighbors) then
            Move node to finished;
        else
            neighbor ← Random node from neighbors;
            labels [neighbor] ← labels [node];
            Move the neighbor to labeled;
        end
    end
end
Return labels;
```

Due to the embedded randomness in our approach, as described in LabelPartition 1, our algorithm does not guarantee global optimum, and we can't guarantee an optimization of any specific measure or function. Moreover, it lacks reproducibility as well, because of the inherent randomness present in the algorithm. It depends on the initial state (first random picks) and if the initial picks are not optimal, then the result shall also not be optimal.

In order to try to mitigate the sub par random picks and to have a better chance of selecting the partitions that minimize the number of cuts, the label partitioning algorithm, as described in Algorithm 1 is called c times, each time returning a new random partition of the graph. These random partitions are evaluated using the modularity metric and the highest scoring modularity metric partition among all the partitions is selected as the final output partition. The partitions that maximize the modularity are chosen as final output, as described in Algorithm 2.

Algorithm 2: GraphPartition

```
input : A graph G, number of required partitions (k)
output: k Partitions of the graph
best_partition ← empty();
for i ← 1 to c do
    partitions ← LabelPartition(G, k);
    if Modularity(G, best_partition) <
     Modularity(G, partitions) then
        best_partition ← partitions;
    end
end
return best_partition;
```

Modularity, as defined in [13], over Graph $G(E, V)$ with partitions $c = c_1, ..., c_k$, is defined as:

$$Q = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j)$$

where m is the number of edges, A is the adjacency matrix of G, k_i is the degree of i and $\delta(c_i, c_j)$ is 1 if i and j are in the same community and 0 otherwise.

IV. Asymptotic Analysis

In Algorithm 1, LabelPartition, first few statements (initializing the sets) take constant time, $O(1)$. After that, labeling k random vertices takes $O(k)$ time and the $while$ loop takes $O(n)$ time where n is the number of vertices present in the graph, as we only process a specific node until it has an unlabeled neighbor, and remove the node from processing queue if all of its neighbors are labeled. The time complexity of accessing the neighbors of the node depends on the data structure being used to represent the graph. In this case, as our algorithm mostly deals with accessing neighbors, we are using Adjacency List for graph representation, and we are assuming the time complexity of accessing the neighbors to be constant, $O(1)$. So, the time complexity of Algorithm 1 is $O(1) + O(k) + O(n + 1)$. Since k represents the number of partitions to be made, k will always be less than or equal to n. In most of the cases, k will be very small as compared to

n. So the overall time complexity of Algorithm 1 comes down to $O(n)$ i.e. a linear time complexity.

In Algorithm 2, we simply call the Algorithm 1 c times, where c is a user-defined constant, specifying the number of iterations to be run to get a better and more modular partition. So, the running time complexity of Algorithm 2 is $O(cn)$, where c is usually quite small.

V. Results

To test the effectiveness of Random Label Partitioning algorithm, we ran it on some of the most widely used graph datasets in the literature. We ran RLP on Karate Club [14], Air Traffic Control [15], Facebook combined [16], Dolphins [17], Lesmis [18], and Mcldata [19], as shown in Table I, where k represents the number of partitions and c represents the number of iterations that Algorithm 1 was executed for. RLP gives a good modularity value while keeping the running time very low as compared to other algorithms for graph partitioning.

TABLE I: Results by Random Label Partitioning algorithm

Graph	Nodes	Edges	k	c	Modularity	Time (ms)
Karateclub	34	78	4	50	0.4198	6.92
AirTC	1226	2615	8	25	0.6039	273
Facebook	4037	87933	6	25	0.6748	6120
Dolphins	62	159	3	25	0.4609	4.85
Lesmis	77	254	5	50	0.5403	33.2
Mcldata	200	2500	2	25	0.3221	47.2

VI. Conclusion

Our algorithm is naive in it's approach but given the linear time complexity, it is a good starting point in improving upon the existing method and incorporating latest methods and approaches documented in the literature. Furthermore, it can be used as it is, in real-time scenarios where we have strict constraints over time and we want a quick solution rather than an optimal one.

References

[1] A. Awadelkarim and J. Ugander, "Prioritized Restreaming Algorithms for Balanced Graph Partitioning," 7 2020. [Online]. Available: http://arxiv.org/abs/2007.03131

[2] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, vol. 76, no. 3, pp. 1–11, 2007.

[3] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[4] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proceedings - Design Automation Conference*. Institute of Electrical and Electronics Engineers Inc., 1 1982, pp. 175–181.

[5] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning Sparse Matrices with Eigenvectors of Graphs," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 3, pp. 430–452, 7 1990.

[6] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 888–905, 2000.

[7] T. Leighton, S. Rao, and A. Srinivasan, "Multicommodity flow and circuit switching," in *Proceedings of the Hawaii International Conference on System Sciences*, vol. 7. Institute of Electrical and Electronics Engineers Computer Society, 1998, pp. 459–465.

[8] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal of Scientific Computing*, vol. 20, no. 1, pp. 359–392, 7 1998.

[9] D. R. Karger, "Global Min-cuts in RN C , and Other Ramications of a Simple Min-Cut Algorithm," 1992.

[10] J. Wu, G. Jiang, L. Zheng, and S. Zhou, "Algorithms for balanced graph bi-partitioning," *Proceedings - 16th IEEE International Conference on High Performance Computing and Communications, HPCC 2014, 11th IEEE International Conference on Embedded Software and Systems, ICESS 2014 and 6th International Symposium on Cyberspace Safety and Security*, pp. 185–188, 2014.

[11] K. Andreev and H. Racke, "Balanced Graph Partitioning," *Theory of Computing Systems*, vol. 39, pp. 929–939, 2006.

[12] J. Zeng and H. Yu, "A study of graph partitioning schemes for parallel graph community detection," *Parallel Computing*, vol. 58, pp. 131–139, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2016.05.008

[13] M. Newman, *Networks: An Introduction*. Oxford Scholarship Online, 2010. [Online]. Available: https://www.oxfordscholarship.com/view/10.1093/acprof:oso/9780199206650.001.0001/acprof-9780199206650#

[14] W. W. Zachary, "An Information Flow Model for Conflict and Fission in Small Groups," *Journal of Anthropological Research*, vol. 33, no. 4, pp. 452–473, 12 1977.

[15] J. Kunegis, "KONECT," in *Proceedings of the 22nd International Conference on World Wide Web - WWW '13 Companion*. New York, New York, USA: Association for Computing Machinery (ACM), 2013, pp. 1343–1350. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2487788.2488173

[16] J. M. Stanford, "Learning to Discover Social Circles in Ego Networks," Tech. Rep., 2012. [Online]. Available: http://snap.stanford.edu/data/

[17] D. Lusseau, K. Schneider, O. J. Boisseau, P. Haase, E. Slooten, and S. M. Dawson, "The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations," *Behavioral Ecology and Sociobiology*, vol. 54, no. 4, pp. 396–405, 9 2003. [Online]. Available: https://link.springer.com/article/10.1007/s00265-003-0651-y

[18] D. E. Knuth, *Stanford GraphBase: A Platform for Combinatorial Computing*, 1st ed. Addison-Wesley, 1993. [Online]. Available: https://www.informit.com/store/stanford-graphbase-a-platform-for-combinatorial-computing-9780321606327

[19] H. Wang, W. Gan, S. Hu, J. Y. Lin, L. Jin, L. Song, P. Wang, I. Katsavounidis, A. Aaron, and C. C. Kuo, "MCL-JCV: A JND-based H.264/AVC video quality assessment dataset," in *Proceedings - International Conference on Image Processing, ICIP*, vol. 2016-Augus. IEEE Computer Society, 8 2016, pp. 1509–1513.