An efficient implementation of Graph Clustering on GPU

Rudraksh Kashyap - 11840970
Dept. of EE
IIT Bhilai
Kota(Rajasthan), India
rudrakshk@iitbhilai.ac.in

Ashish Kumar Suraj - 11840230
Dept. of EECS
IIT Bhilai
Barabanki(UP), India
ashishs@iitbhilai.ac.in

***Abstract*—The objective of graph clustering is to identify the community structure of the graph, specifically the community membership for each node in the graph. Graph processing has always been a challenge, as there are inherent complexities in it. There are several implementations for parallel processing on the GPU systems. So the problem is to identify the optimal clusters in a larger graph using parallel programming. Although graph partition is a NP-hard problem, but existing relaxation methods provide reasonable approximate solutions in reasonable time. But the scalability with increasing graph size of such solutions remains a challenge. In this project, we try to find new techniques to speed up the stochastic block partition algorithm.**

I. Introduction

Graph clustering refers to clustering of data in the form of graphs. Finding the clusters in a large graph is important to understand the underlying relationships between the nodes. Clustering helps in understanding the natural grouping in a dataset. Graph data has come to play a critical role in a diverse array of applications, particularly when looking for hidden or complex relationship structures and activities. But performance and scalability remain challenging issues because the computational complexity of many traditional algorithms, including graph partitioning (also known as community detection) is NP- hard. For graph partitioning, the development of approximation algorithms [1] has been critical in finding feasible solutions for real-world datasets. Unfortunately, there are major scalability challenges even for approximation algorithms. These challenges are made even more difficult for graph partitioning when the number of communities is not known a priori and there are no a priori cues about the membership of some of the nodes.

There are a lot of clustering algorithms for the data to be compiled. Such as based on Topology algorithms, Edge Betweenness Clustering, Biconnected Components Clustering, k-means Clustering, Hirarchic Clustering and so on. Also there are lot of papers out there which are written in the same field.

Previous IEEE HPEC GraphChallenge works have used a shared memory Louvain implementation and spectral clustering methods to achieve substantial speedups in partitioning performance over the baseline stochastic block partition algorithm. In this proposal, we focus on improving the algorithmic performance of stochastic block partition.

The contributions of our work are as follows:

———TODO———————

The rest of the proposal is outlined as follows. Section II provides an overview of the stochastic block partition algorithm. Details of our optimization technique are provided in Section III. Section IV presents a performance evaluation of the optimization techniques. Finally, related work, their limitations and how our approach is different is mentioned in Section V.

II. Approach

In this project we are planning to efficiently parllize **Stochastic block partition algorithm** , which forms the GraphChallenge baseline, uses a generative statistical model based on work by Peixoto [6], [7], [8] and builds on the work from Karrer and Newman [9]. Implementation in any language of the baseline algorithm has excellent single-threaded performance on small graphs, but scaling this performance is difficult. The algorithm has two challenges: the optimal number of blocks is not known a priori, and neither is the assignment of each node to a block. To overcome these challenges, the static algorithm uses an entropy measurement function which measures the quality of a partition for a given number of blocks. Once it finds the optimal partitioning at a particular number of target blocks and the associated entropy, it can compare that entropy against future partitions at a different number of target blocks.

In particular, the block phases of the algorithm work as follows, where N refers to the number of nodes in the graph. It first finds optimal partitions and entropies for partition sizes $\frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \ldots,$ until a valley is found where further reductions in block size no longer produce decreases in entropy. When this minimum entropy is bracketed between two partition sizes, the algorithm switches to a Golden section search to find the final partition and entropy. To find the optimal partition for a given number of blocks, the algorithm goes from a larger partition size to a smaller one. It does this using two distinct program operations – *agglomerative merging* and *nodal movements*.

During an agglomerative merge, two existing candidate blocks are found by greedily picking the merge with the lowest resulting entropy. Each agglomerative merge takes two communities and puts them together as one, resulting in one fewer number of blocks. Once the algorithm reaches the target number of blocks, it switches to performing nodal moves that can reassign a vertex from one block to another. Movements that result in large decreases in entropy are likely to be accepted, and movements that do not are unlikely to be accepted. The overall algorithm proceeds in this manner, alternating between agglomerative merges and nodal movements at each block phase. When the overall change in entropy stabilizes for a particular number of blocks, the algorithm stops movements for that partition size and goes to the next target block number. [2], [3] The overall complexity of the baseline algorithm is $O(E \log^2 E)$ [4]. One advantage of stochastic approaches is that more complex rules that govern the goodness of communities in different application domains can be easily adapted to a stochastic method.

Parallelism in above algorithm
Unlike agglomerative merges, nodal movements affect one node at a time rather than entire communities. Such behavior makes them write-intensive and difficult to parallelize. Furthermore, the quality of nodal updates for one worker depends on the timely incorporation of updates from other workers. Our baseline implementation will avoids expensive global updates to the shared interblock edge count matrix by having each worker thread maintain its own copy of the data structure. We will try to designed our implementation to carefully manage the granularity of messages, both from the workers to the main aggregator and back to each worker, to minimize the amount of data transferred. This parallel nodal movement design uses a pool of worker processes, each responsible for a group of vertices, which finds nodal movement proposals for that group, computes the associated acceptance probability, and ultimately accepts or rejects the proposals. Each worker reports the changes in the state back to the main aggregator for efficient distribution to the other workers.

We will try to program this approach to for a configurable number of move threads and merge threads and will try to vary one while holding the other constant and then note the time taken.

III. Evaluation Methodology

For the evaluation of the techniques, we are planning to start with our fast parallel stochastic block partition implementation written in Python. Our implementation might be using NumPy[16] for low-level array operations, and on Python multiprocessing parallel operation. We are going to use latest available version of Python and Numby. Most of the times we shall use kaggle or google colab for conducting our experiments and we may as well use our machine(if possible) for light weight datasets. We may also use Graph-tool which is an efficient Python module for manipulation and statistical analysis of graphs.

One straightforward measure of algorithm performance is the amount of time used to perform partitioning. We will aim to instrumented our code in such a way that it only will measure just the time spent during computation, ignoring overheads such as graph load time from disk.

Input data set analysis can be of different types as interval, ordinal or categorical. We shall start our experiment with graphs having less number of nodes and then gradually we shall increase the complexity of the graph. Dataset for the Streaming Graph Challenge: Stochastic Block Partition can be found here.

IV. Related Work

Previous IEEE HPEC GraphChallenge works have used a shared memory Louvain implementation and spectral clustering methods to achieve substantial speedups in partitioning performance over the baseline stochastic block partition algorithm.
Other recent work shows the advantage of using sampling compared to modularity minimization [14]. Our work will be complementary to this approach since the core data structure and algorithmic operations remain the same.

References

[1] Y. Jin and J. F. Jaja, "A high performance implementation of spectral clustering on cpu-gpu platforms," in Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International. IEEE, 2016, pp. 825–834.

[2] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," The journal of chemical physics, vol. 21, no. 6, pp. 1087–1092, 1953.

[3] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," Biometrika, vol. 57, no. 1, pp. 97–109, 1970.

[4] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song, and et al., "Streaming graph challenge: Stochastic block partition," 2017 IEEE High Performance Extreme Computing Conference (HPEC), Sep 2017. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2017.8091040