CS 5395 Independent Study Project Report

Parallel Computation of Community Structures in Graphs

Texas State University

Department of Computer Science

Sirichi Bobby Srisan

Advisor: Dr. Martin Burtscher

Spring 2020

Abstract

In this project, I study the parallelization of the Louvain method and its modularity optimization algorithm to rapidly detect communities within large graph structures [1]. Community discovery is useful to find varying granularities of embedded groupings of nodes. For example, hierarchies of social communities may be found at local, city, or national levels and beyond. The problem of community detection requires partitioning a graph into communities of densely connected nodes with sparse interconnections between communities. Precise formulations of this optimization problem are known to be computationally intractable, so approximation algorithms are necessary when dealing with large graphs. A fast method for community discovery is beneficial when decisions must be made quickly. I explore parallel processing using GPUs with the goal of reducing overall computation time to retrieve community partitions. I evaluate the performance of and discuss findings related to my implementation.

1. Introduction

Social network analysis is the process of studying behaviors of social structures through networks and graph theory. Complex systems of information are often represented as nodes with links. Nodes are individual actors that are linked through some relationship or interaction. Social networks, mobile phone networks, collaboration networks are some examples of such structures, which can reach sizes of billions of nodes and links. A key part of social network analysis is partitioning networks into subunits, also known as communities, to explore relationships in more detail. Communities are sets of nodes with relatively high interconnections. An ideal partitioning creates groups wherein elements share a common attribute or interact with one another more so than with members of other communities. The quality of the partitions is often measured by modularity, a value calculated as:

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

Equation 1.

where A_{ij} is the weight of link between *i* and *j*; k_i is the sum of link weights attached to node *i*; c_i is the community of node *i*; m is the sum of link weights in the graph; and $\delta(c_i, c_j)$ is a Kronecker function that evaluates to 1 if c_i and c_j are equal, and 0 otherwise. In effect, modularity measures the relative density of links inside communities to links outside communities.

Most large networks contain within them several natural organization levels. For example, hierarchies of social communities occur at local, city, national levels, and beyond. Thus, it is possible and desirable to find community partitions at different levels of resolution. A maximum Q theoretically results in the best possible grouping of the nodes, however, going through all possible iterations of the nodes into groups is impractical for very large graphs, so heuristic algorithms are used.

1.1 Louvain Method of Community Detection

The Louvain method for community detection is an approach to extract communities from large networks. The method uses a greedy algorithm to find hierarchical node clusters while optimizing modularity as it progresses.

The algorithm has two phases that repeat iteratively. Given a weighted network of *N* nodes, we initially assign each node to a community of its own. So, in this initial partition there are as many communities as there are nodes, and we have a starting modularity value. Then, in the first phase, we perform a local maxima search for modularity; for each node *i*, we consider a neighbor, *j*, and we evaluate the increase of modularity that would take place by removing *i* from its community and by placing it in the community of *j*. All neighbors of each node are considered. We reassign a node to the neighboring community that produced the largest modularity gain. This first phase stops when all nodes have been visited at least once and when no individual move can improve the modularity. The second phase defines a coarser network in terms of communities found in the previous state. In this network, each community becomes a node of itself. The weights of the links between the new nodes are given by the sum of the weight of links between nodes in the corresponding communities. Additionally, weights of links strictly inside communities become self-loops of the new nodes. Once the new network is defined, we apply the local search as described in phase one. The two phases are repeated until no further modularity-increasing reassignment of communities is possible.

When comparing modularity optimization methods, two measures of importance are the speed and the resulting modularity value. A higher modularity value indicates better-defined communities. An approach that rapidly finds results shows efficiency in resource usage. In Figure 1, we see that while the Louvain method is indeed faster than others, graphs with millions of nodes will run for minutes to hours as the graph size increases [1].

Graph name	*Karate*	*Arxiv*	*Internet*	*Web nd.edu*	*Phone*	*Web uk-2005*	*Web WebBase 2001*
Nodes /links	34 / 77	9k / 24k	70k / 351k	325k / 1M	2.6M / 6.3M	39M / 783M	118M / 1B
Clauset, Newman,Moore	.38 / 0s	.772 / 3.6s	.692 / 799s	.927 / 5034s	-/-	-/-	-/-
Pons, Latapy	.42 / 0s	.757 / 3.3s	.729 / 575s	.895 / 6666s	-/-	-/-	-/-
Wakita, Tsurumi	.42 / 0s	.761 / 0.7s	.667 / 62s	.898 / 248s	.56 / 464s	-/-	-/-
Louvain	**.42 / 0s**	**.813 / 0s**	**.781 / 1s**	**.935 / 3s**	**.769 / 134s**	**.979 / 738s**	**.984 / 152min**

Table 1: Performance comparison of modularity optimization methods. Main entries are formatted as modularity value/run time. -/- indicates the method took over 24hrs to run.

2. Parallelizing the Louvain Method

The goal of this project is to explore parallel processing using GPUs to reduce overall computation time to retrieve community partitions via the Louvain method. Real world networks contain millions of datapoints, so, we seek rapid results. With a broad understanding of the algorithm and original reference source code, we can study the authors' original reference implementation to identify critical functions, data structures, and other calculations.

Using the *Python* reference implementation and a program profiler, results show that much of the run time occurs on code corresponding approximately to the first phase of the algorithm. That is, there is a significant amount of time spent on iterating through all nodes to search for community reassignments that produce positive gains in modularity. Therefore, this section of code and accompanying functions are candidates for parallelization.

```
      1741849860 function calls (1740434396 primitive calls) in 2173.463 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.004    0.004 2173.463 2173.463 {built-in method builtins.exec}
     1    0.093    0.093 2173.458 2173.458 <string>:1(<module>)
     1    0.175    0.175 2173.365 2173.365 community_louvain.py:161(best_partition)
     1    2.714    2.714 2171.409 2171.409 community_louvain.py:253(generate_dendrogram)
     6  433.219   72.203 2059.197  343.200 community_louvain.py:463(__one_level)
```

Listing 1: Timing profile of Python Louvain serial implementation

Consider that for a graph of many nodes, this function generally runs for a longer time. Furthermore, in the serial iterations, the ordering can influence the computation time. An approach, then, is to use the multithreading capability of a GPU to compute all nodes in graph at once. The main idea is that each thread is assigned to do the neighborhood search for one node. Once the individual computations are done, the threads will perform a global consensus to decide which node should be assigned to which community that will give the highest modularity gain among all nodes, given the present community memberships.

To aid in the computation, an equation tells us relative gain obtained by moving an isolated node *i* into a community *C*:

$$\Delta Q = \left[\frac{\Sigma_{in} + \mathrm{k}_{i,in}}{2m} - \left(\frac{\Sigma_{tot} + \mathrm{k}_i}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2 - \left(\frac{\mathrm{k}_i}{2m}\right)^2\right]$$

Equation 2.

where Σ_{in} is sum of link weights strictly inside *C*; $\mathrm{k}_{i,in}$ is the sum of link weights from *i* to nodes in *C*; Σ_{tot} is the sum of link weights incident to C; k_i is the sum of link weights to node *i*; and m is the sum of all link weights in entire graph. In practice, the change of modularity is evaluated by removing *i* from its community and then by moving it into a neighboring community. It should also be noted that this equation can resolve down to terms with k_i and Σ_{in} removed.

2.1 Design

Since GPUs have high performance in coalesced reading and writing of arrays, my design uses a series of arrays to keep track of the information used in Equation 2. These data arrays are copied over to the GPU and shared among the threads. They include:

- current node-to-community memberships: indexed by node number, these values are each nodes' respective community number assignments
- sum of link weights strictly inside each community: indexed by community number
- sum of link weights incident to each community: indexed by community number
- sum of link weights incident to each node: indexed by node number
- compressed sparse row representation of the entire graph's adjacency matrix

The sum of link weights of nodes and the compressed sparse row (CSR) arrays will not change throughout the life of the graph. The CSR arrays are constructed in the standard format. For the other arrays, the number of nodes of the graph determines their maximum size, and they are equal in length. Since all communities are singular nodes on the outset, the arrays holding per-community sums have a length equal to the number of nodes. I use an array to hold precomputed sums of link weights to nodes to speed up the modularity calculation. On each visit to a community, the sum of link weights from node *i* to nodes in community *C* (represented by $k_{i,in}$ in Equation 2) is recomputed rather than being stored in an array. Right before the second phase, the current partition is pushed onto a stack to retain the community hierarchy. All arrays except for the fixed arrays are reallocated in the second phase.

A concurrency problem arises for one pass of nodes in the graph because each node must consider itself removed from its community in an intermediate step. Recall that the strategy is to use a thread of perform computations for a node using shared data about the current state of communities. For example, consider a scenario where a thread changes the sum of incident link weights to a community as its assigned node has been removed, as the intermediate step in algorithm necessitates. Concurrently, another thread is calculating the change in modularity if another node were to move into that same community. This changes the sum of links incident to the community. Employing locks and a critical section while a community is being considered degrades efficiency because many threads will become serialized. A more straightforward solution that removes data dependencies between threads is to account for self-community removal with a local value. That way, the shared memory values do not change in the middle of computation for other threads.

2.2 Implementation

The implementation for this project had three main components:

Python: In data analysis, *Python* provides comprehensive scientific toolkits to analyze and explore datasets. *Python* also provides functional scripting, which is beneficial when writing complete low-level functions is not a concern. Additionally, *NetworkX* is a useful *Python* library for studying graphs and networks.

CUDA: *CUDA* (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) created by *NVIDIA*. It allows general purpose computing on GPUs. The *CUDA* API is accessed in *C*/*C++*.

Cython: *Cython* is a programming language designed to give *C*-like performance with *Python* code. With *Cython*, we can write *Python* code that calls back and forth from and to *C*/*C++* code natively at any point. A *Cython* wrapper interfaces between the main *Python* module and the written *C++* module, which manages CPU-to-GPU flow.

A *Python* module reads in network graph data and generates the relevant arrays from it. It uses the *Cython* wrapper to initialize a *C++* class object that will reference the arrays. In turn, the *C++* object also allocates memory for the arrays in the GPU device and copies over data. The arrays referenced in the *Python* and *C++* processes are the same memory allocations. As needed by the respective *Python* code, *CUDA* kernels are launched to do the parallel computation. The pair of best node and best community is copied back to the CPU for one round

of community moving. After development, the *Cython* module is compiled and imported as a *Python* module.

3. Test Setup

For test and development, I used an *NVIDA Tesla K20* GPU. The GPU has a maximum of 2048 threads per SM. The test data I used comes from a dataset known as *Zachary's Karate Club*. It is commonly used to demonstrate community structure in graphs. It has 34 nodes and 78 links with an average degree of 5 and maximum degree of 16. I ran the serial code five times, randomizing the node iterations each time. For my GPU-enabled code, I ran the data set five times as well.

3.1 Analysis of Results

Graph	Louvain run time	GPU run time	Louvain modularity	GPU modularity
karate	0.0044 s	0.0033 s	0.417	0.394

Table 2: Comparison run of serial code and GPU-enable code. Values are the average of 5 runs.

Initial results indicate some speedup. The original algorithm's code computed one level of partitioning at 0.0044 seconds, whereas the GPU completed at 0.0033 seconds. Due to time constraints, tests and results from larger graphs were not obtained. So, firm conclusions on speedup results will require more data.

Referring to the modularity, the original Louvain code gives a higher value, which means it found better community partitions. The GPU consistently returns the same modularity because it performs a greedy optimization among all nodes at once, meaning that the order in which nodes are assigned is always the same, whereas the serial code arrives at different modularity values depending on the starting node, which is randomized. For larger graphs, and depending on the links, the GPU-enabled code will enter phase 2 sooner because it reaches a state of maximum modularity sooner.

It is also interesting to note that the resulting community assignment set of the GPU is one that belongs in the set of possible communities that the original method can produce. This was verified by running the serial code through many iterations until the modularity matched and the community set matched.

For larger graphs, there is overhead in generating the arrays that keep track of sums and node assignments. This memory consumption in the GPU code may be the same as the serial code as it also maintains similar lists of values.

4. Conclusion and Future Work

In future work, we can add more parallelism by simultaneously assigning nodes of non-adjacent communities because they will have independent operations. Considering that not all nodes have neighboring communities, instead of assigning one node at a time after finding the local maximum delta, we can assign more if the threads are not reading and writing in the same shared array locations. Furthermore, for adjacent communities, we can perform concurrent assignments, but cautiously. Here we may create critical sections to handle cases such as when two or more nodes are entering, leaving, or swapping adjacent communities.

The goal of this project was to find a way to improve the Louvain method in terms of speedup. Through parallel processing using a GPU, I modified a process-heavy step in the original algorithm such that neighborhood computation is done for all nodes at once. While this may improve speed for large number of nodes, larger datasets are needed to confirm this. Additionally, the modularity score of this approach is not yet optimal as compared to the original implementation. A globally greedy assignment, while more consistent, will likely result in lower modularity. A heuristic using simulated annealing or hill climbing is another possible improvement to optimizing modularity.

References

[1] V. D. Blondel, J.-L. Guillaume, R. Lambiotte and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment,* 2008.