Parallel Implementation of the Label Propagation Method for Community Detection on the GPU

Marko Mišić[1], Dražen Drašković[1], Lovro Šubelj[2], Marko Bajec[2]

[1]*University of Belgrade, School of Electrical Engineering*
[2]*University of Ljubljana, Faculty of Computer and Information Science*
E-mail: marko.misic@etf.bg.ac.rs

Abstract. *Community detection in social and other networks is one of the important problems for many applications. Label propagation has proven to be one of the most scalable methods for detecting communities in large complex networks. Rapid development of modern graphics processing units (GPUs) has allowed a significant increase in computing power. In this paper, we present a GPU implementation of synchronous label propagation method. We observed speedups over sequential implementation in the range of 3-50 times for small networks to 50-300 times for larger networks with thousands of nodes. The results of the analysis are briefly discussed with an emphasis on benefits and drawbacks of using GPUs for community detection.*

1 Introduction

Complex real-world systems in nature, society, and technology can be modeled with networks (graphs). Nodes of the network represent different entities, while edges represent connections between them. Different examples exist, such as web pages on the Web connected by hyperlinks, social networks consisting of people connected through various degree of friendship, co-authorship networks in scientific production, etc. [1]

Many of these networks consists of smaller modules with denser local structure, called communities [2]. Communities are intuitively defined as groups of nodes densely connected within and only loosely connected with the rest of the network. Community detection is significant for understanding of complex systems in sociology, biology, and computer science, as members of the community usually play similar role in the system [3]. Various algorithms for community detection have been proposed in the past. Rather comprehensive survey of those algorithms can be found in [3, 4]. On the other hand, due to scalability issues, only a minority of these algorithms can be applied to large real-world networks with millions of nodes and edges. For this reason, label propagation [5] and its variants [6] are often used for community detection.

The other significant issue in processing of large real-world networks is execution time. Execution time can be decreased using modern central processing units (CPUs) and graphics processing units (GPUs) which offer significant amount of parallelism [7]. Modern CPUs are multicore processors, consisting of dozens of cores that are able to execute program code concurrently, while contemporary GPUs are manycore accelerators with hundreds or thousands of cores available for execution. CPUs and GPUs are usually combined in a heterogeneous system, where compute-intensive parts are offloaded to the accelerator for fast execution. In that sense, extracting and mapping parallelism from existing problems becomes more and more important in different fields of research.

Computational efficiency and application of the community detection methods to very large networks have been studied in the open literature. Distributed community detection approaches have been presented in [8, 9], while experiences with community detection on multicore and GPU architectures can be found in [10].

In this paper, we have implemented and evaluated synchronous label propagation algorithm (LPA) on the GPU using NVIDIA CUDA programming model. Parallel implementation of LPA is tested and compared to its sequential counterpart with several artificial and real-world workloads. We observed significant speedups over sequential implementation and present our experiences with porting LPA on the GPU.

The rest of the paper is organized as follows. The second section presents a short overview of community detection algorithms with the emphasis on LPA. In the third section, we give a short overview of GPU architecture and CUDA programming model. Implementation details are given in the fourth section, while experimental results are discussed in the fifth section. Conclusion and directives for future work are given in the final section.

2 Community detection in graphs

Let the network be represented by a simple undirected graph $G(N,E)$, where N is the set of nodes and E is the set of edges. Depending on the specific application or analyzed system, different definitions of community exist. Stricter definitions, such as cliques, n-cliques, n-clans, etc. are mostly used in social network analysis [1]. Most of the definitions are based on the premise that members of the community are more densely connected within than with the rest of the network [3]. Communities can be defined as special case of clusters of nodes found in real networks. In this paper we use the terms community and cluster as synonyms.

Sometimes, intra-cluster and inter-cluster densities are used in the definition. Let C be the subgraph of G. Intra-cluster density of the subgraph C can be defined as the ratio between the number of internal edges of C and the number of all possible internal edges. Similarly, inter-cluster density of the subgraph C can be defined as the ratio between the number of edges running from the vertices of C to the rest of the graph and the maximum

number of inter-cluster edges possible. For C to be a community, intra-cluster density is expected to be appreciably higher than average density of the network. Many community detection algorithms try to find a good compromise between high intra-cluster and low inter-cluster density in order to find quality partitioning. Quality of cluster partitioning is often evaluated through modularity [11] and conductance [12] measures.

As mentioned before, numerous methods are used for community detection. Girwan and Newman [2] described an algorithm for hierarchical clustering. The algorithm iteratively removes the edge with the highest edge betweenness. Due to its high time complexity in the order of $O(N^2E)$, it is not practical for networks with more than several thousand nodes. Louvain method [13] is based on modularity maximization, but it has problems with quality of partitions.

Label propagation algorithm [5] is a simple, local-based method for community detection. Each node in the network is assigned a unique label. Communities are found by iteratively propagating labels among nodes in such way that each node adopts the label shared by most of its neighbors. Let w_{nm} be the weight of the edge between nodes n and m, c_n denote the community label of node n, and $N(n)$ denotes the set of its neighbors. In each iteration, the node adopts the label shared by most of its neighbors, takin into account edge weights:

$$c_n = argmax_l \sum_{m \in N^l(n)} w_{nm} \qquad (1)$$

$N^l(n)$ is the set of neighbors of n that share common label l. The labels are propagated quickly in the denser areas of the network, and densely connected sets of nodes form a consensus on some particular label after only a few iterations [5, 14]. Final communities are obtained when algorithm converges, as connected sets of nodes sharing the same label. Time complexity is nearly linear $O(N)$.

The main issues with LPA are convergence problems for some types of networks. The algorithm described above uses synchronous propagation (updating) of labels, where each node's label in iteration i is updated using labels of its neighbors in iteration $i-1$. For that reason, it can produce oscillation of labels for some network configurations, e.g. bipartite graphs. To ensure convergence, asynchronous updating of nodes is proposed in [5]. In such method, nodes are updated sequentially in some random order. On the other hand, introduction of randomness hampers the robustness of the algorithm, and consequently also the stability of the identified community structure. Some authors [15] propose semi-synchronous propagation, based on graph coloring and synchronous propagation, which eliminates convergence problems. However, graph coloring problem introduces non-trivial time complexity in the algorithm.

The other significant problem of LPA that affects convergence is the strategy for resolving majority label ties. Some of the strategies are: random label (ties are broken uniformly at random), label retention (label is retained if among majority labels), label priority (minimum or maximum priority label is taken), etc. In the end, there are different propagation criteria. Propagation can be terminated when each node's label equals the majority label, the label on the previous step, or the step before, or when the number of steps exceeds the defined threshold.

Synchronous and semi-synchronous propagation are suitable for parallel execution, while asynchronous is not, due to its inherent sequential nature. In this paper, we used synchronous propagation with maximum priority tie breaking. Propagation is terminated when each node's label equals the label on the previous step, or the step before, or the threshold is reached. This strategy is proposed in [5], as maximum priority tie breaking cannot produce oscillations with period longer than two, thus it is enough to store labels from two previous iterations.

3 GPU programming and CUDA

GPUs have been used for general-purpose computations for a decade. Nowadays, GPUs are powerful accelerators with manycore architecture, programmed through different low-level and high-level APIs. In this paper, we used NVIDIA Compute Unified Device Architecture (CUDA) [16] programing model, which allows GPU programming in languages such as C, C++, and many others.

GPUs consist of dozens of cores, thus code execution differs considerably from the execution on the CPU. CPU is used for I/O, management tasks, etc., while compute-intensive parts (kernels) are executed on the GPU. Kernel is executed by the large number of lightweight threads in SIMD fashion on the streaming multiprocessors (SMs). Kernel execution is organized as a grid of thread blocks, configured by the CPU. No synchronization between thread blocks is available during particular kernel execution.

GPU memory architecture is designed to support high throughput and execution of number of threads in parallel. There are several memories in the hierarchy that differ in speed and capacity: registers, on-chip (per SM) shared memory, read-only constant and texture memory. Since CPU and GPU operate in separate address spaces, data transfers between CPU and GPU are needed. In order to exploit all parallelism available and to maximize performance, the developer should be aware of resource constraints of the particular device architecture, memory hierarchy, SIMD nature of execution, etc. [17].

4 Implementation details

First, sequential implementation of LPA is written in C++, and it used as a basis for GPU implementation. Adjacency list is used for network representation. It follows LPA principles defined in Section 2. Hash map from STL library is used to count label frequencies.

Initial adjacency list representation is not suitable for GPUs, because of the irregular memory access patterns. Because of that, we adopted adjacency list with padding, where adjacency list for each node is padded with -1 for all missing edges, up to the maximum degree of the node in the network. Although, this might affect memory consumption, we assume that it is not a problem for modern GPUs with large global memory capacities. Also, this data layout is beneficial for memory reads, as they are coalesced, i.e. combined into one transaction.

Also, GPU implementation needed a slightly different approach for neighbors' label counting, as hash maps are not available on the GPU. For that reason, two different approaches have been used. The first approach (*lpa_1024*) is restricted to maximum degree of a node set to 1024, which is the maximum number of threads per block for used GPU architecture. The implementation uses global label counters, stored in fast, shared memory of the SM on the GPU. A kernel is implemented to perform one iteration of the algorithm.

Each thread is in charge of one neighbor in the adjacency list of a node. It reads neighbors' label and atomically updates the appropriate label counter in shared memory using *atomicAdd* intrinsic function. Still, atomic updates to the same memory locations are serialized, thus posing problems in the later iterations of the algorithm, when number of neighbors share the same label. The process iterates and kernel execution is repeated until propagation criteria is met.

To overcome the problem with limited maximum degree of a node, a second approach (*lpa*) is taken. In this approach, each thread is responsible for processing of one node and all of its neighbors. To force memory coalescing, initial data structure that represents the network is transposed with dedicated kernel. The other problem is related to label counting, since initial data structure representing counters would need the number of elements equal to maximum degree of a node in network. Shared memory has limited capacity, while global memory is slow. Moreover, search for label-counter pairs would additionally affect performance.

Solution was found with hierarchical mapping of counters to registers, shared, and global memory. The idea is to store the most frequent label and its counter to registers, six successive most frequent labels to shared memory, and the rest to global memory. Mapping to shared and global memory is performed using simple hashing with linear probing. After all neighbors of the node are processed, the most dominant label is stored in a register.

Although proposed hierarchical counting introduces branch divergence to the kernel code, due to the several conditional branches needed to update counters, it does not affect performance greatly. After few iterations, the number of labels significantly decreases. For that reason, remaining labels are stored in register or shared memory, and branching minimally affects performance.

5 Performance evaluation and discussion

Implemented solutions are evaluated on Intel Core i7 5820K 3.30GHz 6-core CPU with 16GB RAM using NVIDIA GTX Titan Black graphics card with 2880 CUDA cores and 6GB RAM under Ubuntu 14.04 OS. Implementations are tested with 14 different artificial networks and 7 different real networks, as shown in Table 1 and Table 2. Artificial networks were generated with the well-known tool described in [18]. Execution time is measured with available GPU timers and *nvprof* tool.

Table 1. Artifical test cases

Network	Nodes	Edges	Avg. degree	Max. degree
graph2k	2000	99120	99.12	200
graph3k_1	3000	148928	99.29	200
graph3k_2	3000	375198	250.13	499
graph3k_3	3000	750884	500.59	600
graph5k_1	5000	246825	100.00	600
graph5k_2	5000	1249990	499.99	600
graph5k_3	5000	1253402	501.36	1000
graph10k_1	10000	286012	57.20	460
graph10k_2	10000	1500837	300.17	460
graph20k	20000	2004428	200.44	1000
graph50k_1	50000	2458290	98.33	1000
graph50k_2	50000	1250020	50.00	90
graph100k	100000	2502007	50.04	90
graph200k	200000	5021393	50.21	100

Table 2. Real test cases

Network	Nodes	Edges	Avg. degree	Max. degree
karate	34	78	4.58	17
dolphins	62	159	5.13	12
books	105	441	8.40	25
football	115	616	10.71	13
jazz	198	2742	27.70	100
euroroad	1174	1469	2.50	10
netsci	1589	2742	3.45	34

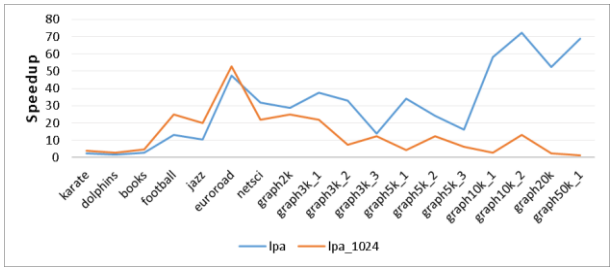

Figure 1. Speedup over sequential implementation for small and medium-sized networks

The results of our analysis are shown in Figure 1 and Figure 2, as speedups of GPU implementations over sequential implementation. The measurements were made only for discrete cases, thus the lines in the figures are merely a guide for the eye.

Figure 1 shows observed speedups over sequential implementation for small and medium-sized networks. For small networks, both implementations exhibit speedup, but *lpa_1024* is twice faster for some test cases. Still, observed speedups are relatively small, because nodes in the network have 5-10 neighbors on average. Parallel overheads are not negligible for both implementations.

Figure 1 shows the clear advantage of both implementations for medium-sized networks, as observed speedups are much higher, especially for *lpa* implementation. *Lpa_1024* exhibits only limited scalability with the number of nodes, as overheads imposed with *atomicAdd* operations become significant.

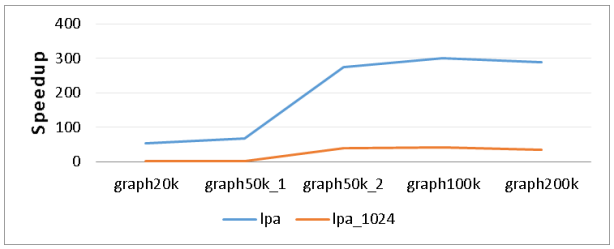

Figure 2. Speedup over sequential implementation for large networks

Figure 2 shows significant speedups of *lpa* implementation for large networks. Observed speedup remains constant with the size of the network for graphs with more than 50 thousand nodes. For that case, profiling showed that GPU resources, such as shared memory and number of blocks per SM, became saturated.

6 Conclusion

In this paper, we presented our experience with implementation of the synchronous label propagation algorithm on the GPU. We presented two different implementations of the algorithm. The first implementation showed better performance for small graphs, while the second approach showed its potential in large-scale networks. Observed speedups over the CPU implementation are significant for all test cases.

There are several directions for future work. First, we should concentrate on the quality of the partitions, since different strategies exist to improve the algorithm stability. Also, more experiments can be performed with hierarchical counting in order to optimize shared memory utilization. In the end, the choice of algorithm for parallel execution can be done depending on the input data analysis.

Acknowledgments

This work has been partially funded by the Ministry of Education and Science of the Republic of Serbia (III44009 and TR32047), and bilateral project Serbia-Slovenia “Open extraction of information for Slovene and Serbian languages”. The authors gratefully acknowledge the financial support.

References

[1] R. A. Hanneman and M. Riddle, "Introduction to social network methods," ed. Riverside, CA: University of California, Riverside, 2005.

[2] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences,* vol. 99, no. 12, pp. 7821-7826, 2002.

[3] S. Fortunato, "Community detection in graphs," *Physics reports,* vol. 486, no. 3, pp. 75-174, 2010.

[4] S. Fortunato and D. Hric, "Community detection in networks: A user guide," *Physics Reports,* vol. 659, pp. 1-44, 2016.

[5] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical review E,* vol. 76, no. 3, p. 036106, 2007.

[6] L. Šubelj and M. Bajec, "Robust network community detection using balanced propagation," *The European Physical Journal B-Condensed Matter and Complex Systems,* vol. 81, no. 3, pp. 353-362, 2011.

[7] M. J. Mišić, Đ. M. Đurđević, and M. V. Tomašević, "Evolution and trends in GPU computing," in *MIPRO, 2012 Proceedings of the 35th International Convention*, 2012, pp. 289-294: IEEE.

[8] N. Buzun *et al.*, "Egolp: Fast and distributed community detection in billion-node social networks," in *Data Mining Workshop (ICDMW), 2014 IEEE International Conference on*, 2014, pp. 533-540: IEEE.

[9] M. Ovelgönne, "Distributed community detection in web-scale networks," in *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2013, pp. 66-73: ACM.

[10] J. Soman and A. Narang, "Fast community detection algorithm with gpus and multicore architectures," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 568-579: IEEE.

[11] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E,* vol. 69, no. 2, p. 026113, 2004.

[12] B. Bollobás, *Modern graph theory*. Springer Science & Business Media, 2013.

[13] 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,* vol. 2008, no. 10, p. P10008, 2008.

[14] L. Šubelj and M. Bajec, "Unfolding communities in large complex networks: Combining defensive and offensive label propagation for core extraction," *Physical Review E,* vol. 83, no. 3, p. 036103, 2011.

[15] G. Cordasco and L. Gargano, "Community detection via semi-synchronous label propagation algorithms," in *Business Applications of Social Network Analysis (BASNA), 2010 IEEE International Workshop on*, 2010, pp. 1-8: IEEE.

[16] "CUDA C Programming Guide 7.5," ed: NVIDIA Corporation, 2016.

[17] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Morgan Kafumann, 2012.

[18] A. Lancichinetti and S. Fortunato, "Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities," *Physical Review E,* vol. 80, no. 1, p. 016118, 2009.