Fast Parallel Algorithm For Unfolding Of Communities In Large Graphs

Charith Wickramaarachchi*, Marc Frincu†, Patrick Small* and Viktor K. Prasanna†
*Department of Computer Science
†Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, USA
Email: {cwickram, frincu, patrices, prasanna}@usc.edu

***Abstract*—Detecting community structures in graphs is a well studied problem in graph data analytics. Unprecedented growth in graph structured data due to the development of the world wide web and social networks in the past decade emphasizes the need for fast graph data analytics techniques. In this paper we present a simple yet efficient approach to detect communities in large scale graphs by modifying the sequential Louvain algorithm for community detection. The proposed distributed memory parallel algorithm targets the costly first iteration of the initial method by parallelizing it. Experimental results on a MPI setup with 128 parallel processes shows that up to ≈5x performance improvement is achieved as compared to the sequential version while not compromising the correctness of the final result.**

I. Introduction

Large graphs exhibit patterns that can be viewed as fairly independent compartments with distinct roles, i.e., communities that can be identified based on the graph structure or data clustering [1]. The advent of social networking and online marketing poses new challenges for community detection due to the large data size which can reach up to millions and even billions of vertices and edges. In addition to the graph size, the advent of cloud computing brought the reality of distributed big data to the picture. To cope with this scenario parallel and distributed community discovery algorithms need to be designed. This requires the graph to be initially partitioned across processors so that communication between them during graph processing is minimized. The dynamic evolving nature of these graphs represents another main challenge that emphasizes the need for fast graph analytics. Performing analytics on periodic graph snapshots is one of the proposed approaches to address this issue [2], [3]. However, due to the high velocities of data (e.g., Twitter reports more than 5,700 tweets per second on average [4]), performing analytics even on the snapshots needs to be completed fast for the results to be useful.

Graph partitioning and community detection can be seen as two sides of the same coin which try to achieve highly correlated goals. A major difference between the two is that for the former the number of partitions or partition size is known in advance. Most graph partitioning techniques were designed for parallel computing so that partitions are balanced among processors and communication between them is minimized [5]. Graph partitioning techniques are used in distributed graph analytics where they proved to give good performance for several classes of distributed graph algorithms [6][7]. For large graphs, partitioning is traditionally considered to be part of the data loading process to partition and load the graphs into the distributed storage. Graph algorithms and analytics are later performed on those partitions. Generally, partitioning to load balance and minimize edge cuts has been decoupled from the community detection with many algorithms using a random or hash based assignment of graph vertices to processors [8].

One of the main contributions of this work is to take advantage of the initial graph partitioning when performing parallel community detection in order to speed-up the process by minimizing the communication between processors. The design of several graph partitioning algorithms [9] work in favor of this idea as they try to minimize the cross partition edges between partitions. This reduces the chance for communities based on graph structural information to be spread across multiple partitions. While many community detection methods have been proposed [1] we focus in this paper on the recent Louvain community detection algorithm [10] which performs community detection by using a greedy modularity maximization approach [11]. The aim is to improve the performance of the algorithms's first iteration which takes on average 79% of the total algorithm time (cf. Section III).

We validate our approach by using an MPI implementation on a HPC cluster. To our knowledge this work is not only the first one to propose a distributed memory parallel extension to Louvain algorithm but also the first to take advantage of the graph partitioning to speed-up the community detection. The main contributions of this paper are the following:

1. We propose a distributed memory parallel algorithm extending the Louvain method by making its costly first iteration embarrassingly parallel without any noticeable loss in final modularity.
2. We show that by using different techniques for selecting the vertex traversal order of the Louvain method, we can further improve the performance.
3. We prove the efficiency of our algorithms by using a random community graphs ranging from 250K up to 16M vertices.

The rest of the paper is structured as follows: Section II presents some of the main results related to parallel algorithms for graph partitioning and Louvain community detection; Section III gives a brief overview on the Louvain method; Section IV outlines our proposed approach; Section V describes the

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

experimental setup and discusses the obtained results; and Section VI outlines the main achievements of this work.

II. Related Work

Graph partitioning is a technique which aims to split the graph k-ways such that the edge cuts are minimized while trying to balance the number of vertices in each partition. Most variants of the problem are $\mathcal{NP}$-hard [1]. Kernighan et al. [12] proposed one of the earliest partitioning algorithms. The algorithm tries to optimize a benefit function defined as the difference between the number of edges inside partitions and the ones between them. Existing techniques used in graph processing tools such as Apache Giraph [8] rely on simple hashing or vertex ordering based partitioning. A wide array of multi-level partitioners are implemented in the METIS library [9]. A multi-level parallel partitioning algorithm has been presented in [13].

Recently Kirmani et al. [14] proposed a parallel graph partitioning algorithm based on a geometric scheme that offers good edge cuts. The algorithm is shown to scale better than parallel METIS.

Efficient parallel partitioning algorithms for large social networks have been proposed as well [15]. The algorithm relies on label propagation and on a parallel evolutionary algorithm to obtain quality partitions and is shown to produce better results then parallel METIS.

In this work we utilized parallel METIS partitioner (PMETIS) to perform the initial graph partitioning due to its wide availability and ease of use.

Community detection has been widely studied with Fortunato [1] providing a thorough overview of the main approaches. Most of the work however focused on sequential algorithms. Among the first algorithms to exhibit near linear complexity was presented [16]. The proposed algorithm is based on a label propagation approach. We focus next on existing work that addressed parallel algorithms for community detection with emphasis on parallel Louvain algorithm.

Parallelizing the community detection based on the modularity metric has received an increasingly attention in the past few years. Riedy et al. [17] proposed the first shared memory algorithm for dedicated architectures (Cray XMT and OpenMP). More recently other papers parallelizing the Louvain community method based on OpenMP have been proposed [18], [19]. While all these papers show good performance and modularity results they all take the same approach of using a shared memory architecture. We argue that in order to enable large scale distributed community detection we need to break free of this constraint.

III. Louvian Method For Community Detection

The Louvain method for community detection is a greedy modularity maximization approach [10]. Modularity is a widely-used metric for determining the strength of detected communities [11] and is defined as:

$$Q = \sum_{c \in C} \left[\frac{m_c}{M} - \frac{d_c^2}{4M^2} \right] \quad (1)$$

# Vertices	% Time for 1st Iteration
250,000	90.48
500,000	88.61
1,000,000	87.45
2,000,000	71.08
4,000,000	76.67
8,000,000	72.67
16,000,000	69.75

TABLE I: % Run time spent on first iteration of the Louvain algorithm for different graph sizes.

where C represents the set of all communities, M represent total number of edges in the graph, m_c represent total number of edges inside community c and d_c represent total degree of vertices in c. The modularity metric mainly tries to quantify how many internal edges are in the communities than the expected. For the normalized modularity value for a graph $Q \in [-1, 1]$ a value of 1 represents the ideal scenario.

Louvain method is an iterative algorithm. Each iteration consists of two major steps. It starts by initializing each vertex with its own community. In the first step, it chooses a vertex traversal order to scan through the vertices, which in a random order in the the original sequential implementation [1]. Then for each vertex in the graph it considers its neighbors communities and checks whether or not removing the current vertex from its current community and inserting it to neighboring communities results in any modularity gain. The vertex is added to the community which gives the maximum positive gain in modularity. Community remains unchanged if no positive modularity gain can be achieved. The process is repeated multiple times until a local maximum modularity is reached. In the second step a new graph is created by collapsing the detected communities into vertices. Intra-community edges are collapsed into a single self loop edge and the weight of this edge is calculated by summing up the edge weights of all intra community edges in the community. Multiple edges between each two communities are collapsed into a single edge by summing up the edge weight. The process continues by repeating steps one and two until no improvement in modularity between two consecutive iterations is observed.

Blondel et al. [10] suggested the Louvain methods runtime scales linear with the graph size and the first iteration of the algorithm is the most costly iteration in terms of computation time. We instrumented the sequential version of the algorithm to log the time spent in each iteration and observed that on average 79.53% of the time is spent there for most the community graphs. Table I shows the percentage of the time spent in the first iteration compared to the overall execution time. This result provided us with a clue on what part of the algorithm needs to be optimized. A synthetic community graph generator was used to generate the depicted graphs. More details on the generated graphs will be given in Section V.

IV. Proposed Approach

We modify the Louvain method and parallelize the first iteration in order to significantly reduce execution time (cf.

[1] https://sites.google.com/site/findcommunities/

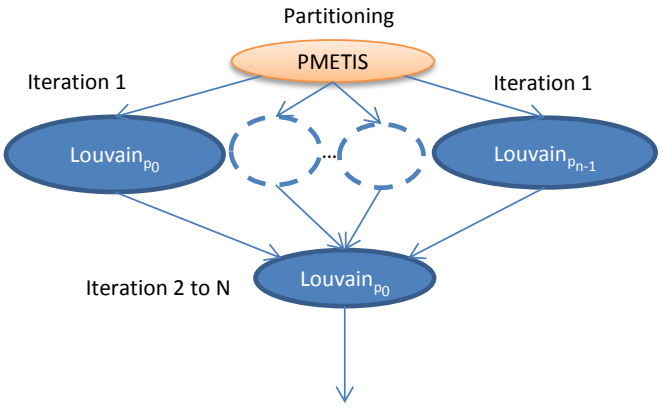

Fig. 1: Workflow of the proposed parallel Louvain algorithm.

Fig 1). For this we partition the graph using PMETIS. The parallel implementation was implemented a using GNU C++ message passing interface implementation where each MPI process picks up its partition based on the process id and runs the first iteration of the Louvain method within the process locally ignoring cross partition edges. Next, the graphs need to be merged together at the master node. As all individual partitions had local vertex numbering we need to renumber all vertices across partitions continuously. Thus one more step is required before sending the data to the master node. Through an *all gather* operation the total number of vertices in each process is sent to all other processes. Each process p_i will receive the total number of vertices in each partition $\{N_0, ., N_{i-1}, N_{i+1}, ..N_{P-1}\}$ where P is the total number of processes. It then renumbers its vertices such that the vertex numbers associated to its partition start from a value n_{start_i} computed based on the values of all processes p_j with $j < i$ as follows:

$$n_{start_i} = \sum_{0}^{i-1} N_j$$

After renumbering, each process sends its renumbered intermediate graphs to a master process which aggregates them into a single graph considering cross partition edges and computes the modularity value. The generated graph represents the first level in the hierarchical community. We must note that at this stage the size of the graph is reduced significantly. The algorithm then proceeds with the rest of the iterations by applying the sequential Louvain on this graph. The pseudocode for this process is shown in Algorithm 1.

As explained in Section III the original Louvain method uses a random vertex ordering at the start of each iteration to traverse the vertices. Because communities are created by a recursive exploration of the neighboring vertices we argue that a method in which the initial vertices are selected based on the edge degree may allow a faster convergence. To prove this we investigated three different vertex ordering strategies:

1 Ordering based on ascending order of edge degree.

2 Ordering based on descending order of edge degree.

3 High degree vertex neighbor order.

Algorithm 1 Pseudocode of the Parallel Extension for the Louvain Method Executed On Each MPI Process.

```
1: procedure PARALLEL LOUVAIN
2:     LOAD-PARTITION(process id)
3:     improvement ← LOUVAIN-ONE-LEVEL                ▷ Louvain iteration
4:     if ¬ improvement then
5:         EXIT
6:     end if
7:     OUTPUT-COMMUNITIES                              ▷ Output communities
8:     CREATE-NEW-GRAPH                ▷ Collapse communities into vertices
9:     RENUMBER-VERTICES      ▷ Renumber the vertices using all gather
   operation
10:    if process id = 0 then      ▷ If current process is the master process
11:        MERGE-GRAPHS
12:        improvement gets true
13:        while improvement do
14:            improvement ← LOUVAIN-ONE-LEVEL
15:            OUTPUT-COMMUNITIES
16:            CREATE-NEW-GRAPH
17:        end while
18:    else                          ▷ If current process is a worker process
       SEND-CURRENT-GRAPH                ▷ Send graph partition to master
19:    end if
20: end procedure
```

While the first two strategies are self explanatory the ordering of vertices for the third case is determined as follows:

1) Scan through vertices in the descending order of the edge degree.
2) For each vertex add all not visited neighbors to a queue and mark them as visited (neighbors of a given vertex are visited according to the ascending order of the vertex ids).
3) The vertex order of the queue is considered as the traversal order.

V. Experimental Results

A. Environment

The performance of our approach versus the sequential implementation of Louvain algorithm was evaluated by running a series of experiments on the University of Southern California's High Performance Computing Center (HPCC) cluster[2]. The HPCC cluster consists of heterogeneous computing nodes. All benchmarks were executed as single batch jobs of 16 compute nodes with 8 cores per node to address this issue. Experiments were conducted multiple times and the presented results are consistent with the results of all other rounds of experiments. Each node consists of two Quad-core AMD Opteron 2376 2.3 GHz processors.

B. Data Sets

Synthetic graph datasets were used in the experiments. The reason for choosing synthetic graphs is to have control over graph sizes to conduct scalability experiments over varying graph sizes. We used Fortunato's benchmarking suite for community detection [20] and generated 7 graphs (cf. Table II). This benchmarking package contains elaborate graph generation algorithms for undirected and unweighted graphs with community structures. In addition, the graph generation process allows fine control over many properties of the

[2]http://hpcc.usc.edu/

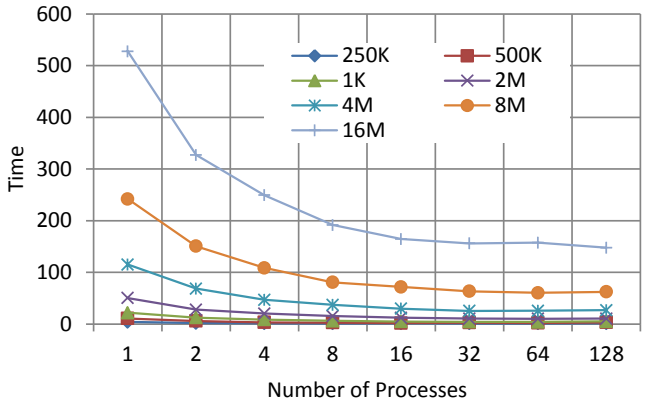

Fig. 2: Execution time when scaling up number of processors for different graph sizes.

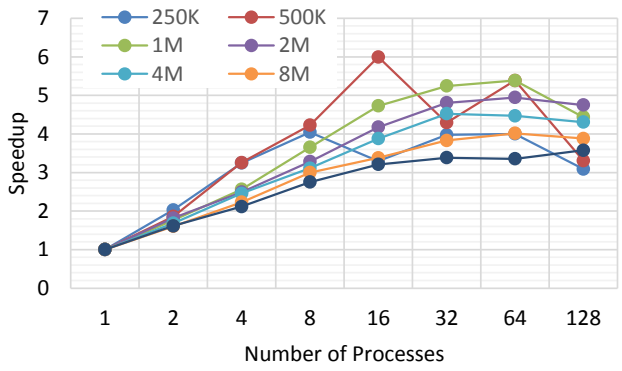

Fig. 3: Speedups compared to the sequential implementation when scaling up number of processors for different graph sizes.

produced synthetic graphs, including average and maximum degree distribution, and community overlap. But production of the largest graph in our dataset, e.g., 16M nodes, required a compute node with 64GB of memory at HPCC. That constrained us from generating larger graphs. A brief summery of data set properties are shown in Table II.

# Vertices	#Edges
250,000	936,948
500,000	1,874,686
1,000,000	3,751,912
2,000,000	7,503,410
4,000,000	15,009,838
8,000,000	30,024,411
16,000,000	60,067,971

TABLE II: Data sets statistics summary.

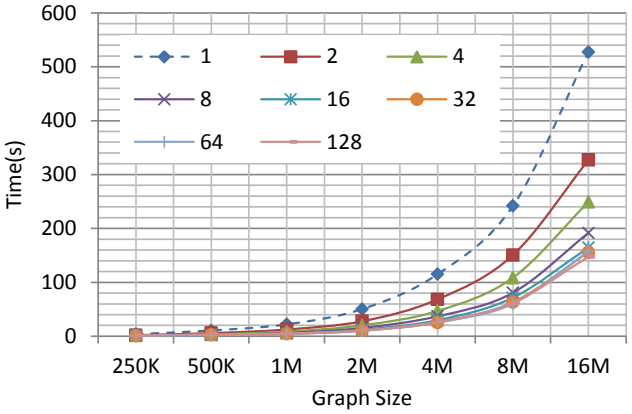

Fig. 4: The execution time when scaling up graph sizes with different number of processors.

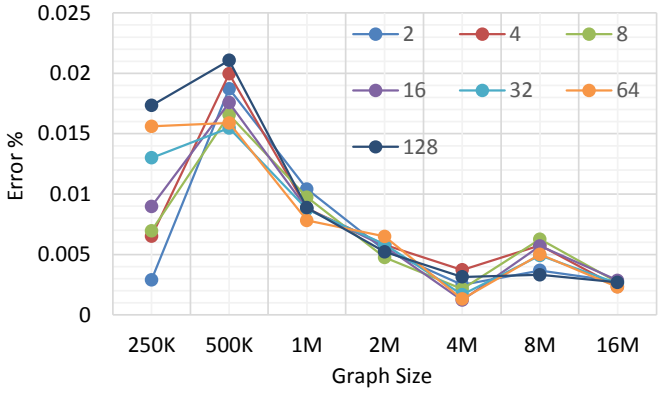

Fig. 5: Change in the percentage error of final modularity with graph sizes.

C. Results

We compared our parallel algorithm with the original Louvain algorithm implementation and also tried to compare our results with the shared memory implementation presented in [18]. However we found that the version used by authors of shared memory implementation does not scale beyond graphs larger than 10,000 vertices.

The main focus of this experiments was to analyze the scalability of our approach. We conducted series of experiments scaling the problem in two dimensions, namely with increasing number of processors and increasing graph sizes.

Figure 2 shows the change in total execution time when scaling up the number of processors for parallel implementation. We can see our approach outperforms the sequential version for all configurations. We also observed that the number of iterations of our algorithm remained the same as the sequential Louvain method. This is an indication that the partitioning of the graph for the first iteration did not split the communities in the first level(or had a very minimal impact). We notice however that the performance improvement decreases when scaling up number of processors for all graph sizes. The same behavior can be observed in Figure 3 where speedups start to flatten for most of graphs after scaling up to 16 - 32 processors. The reason behind this is due to the upper bound enforced from the Amdahl's law [21] and implementation overheads.

As shown in Figure 4 we can see the gap between the runtime of both sequential and parallel implementations increases with graph size. This is an indication of good

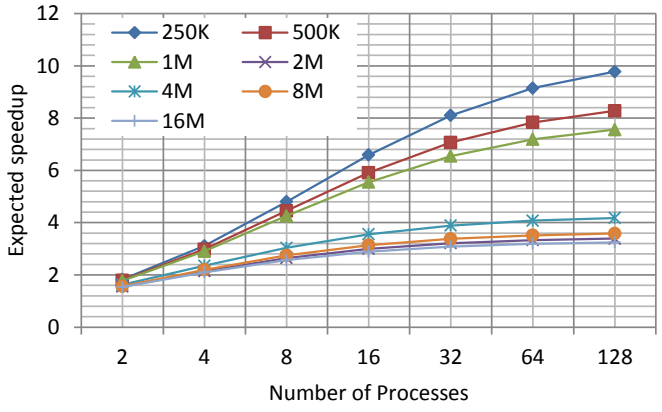

Fig. 6: Expected (theoretical) speedup for the parallel implementation when scaling up the number of processors for different graph sizes.

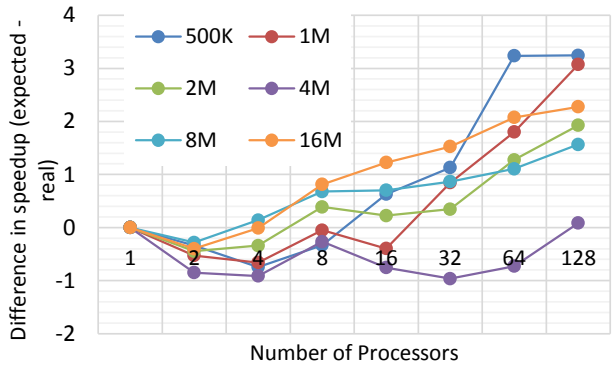

Fig. 7: Difference between expected speedup and actual speedup for parallel implementation when scaling up number of processors for various graph sizes.

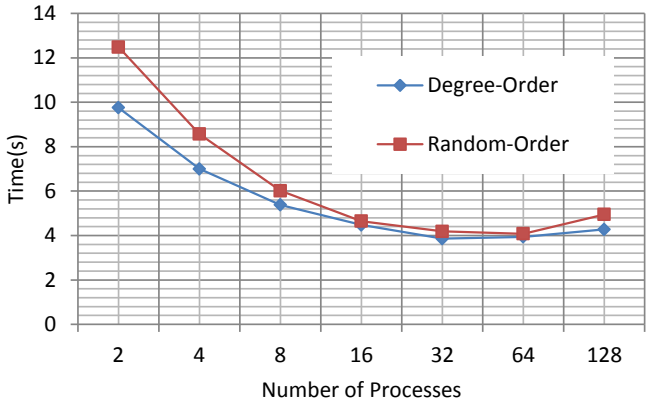

Fig. 8: Runtime behavior when using vertex ordering based on descending order of edge degree compared to random vertex order. Graph size: 1m vertices

scalability of our approach for larger graphs. But we can observe the improvement in runtime reduces when scaling up the processors up to 16-32 due to the same reasons explained before.

The correctness of the community detection algorithms is of a main concern when trying to gain parallel speedups. As an example if we ignore cross partition edges in community detection algorithms it might cost us in terms of final modularity due to incompleteness in detecting communities that span over multiple graph partitions. This poses the need for a fine balance between performance of algorithm and correctness of results. Figure 5 shows the behavior of the percentage error calculated using the difference in modularity between the sequential and parallel algorithms. Error E is calculated based on Relation 2:

$$E = abs(\frac{q_p - q_s}{q_s}) \quad (2)$$

Where q_s and q_p represents the final modularity obtained from sequential algorithm and our approach respectively. Results show that our approach finds communities with almost same quality as the sequential Louvain method (less than 0.025% error). One notable observation is that the percentage error reduces with the graph size. This is expected since the probability of partitioning communities into multiple graph partitions reduces for larger graphs since PMETIS tries to reduce the number cross partition edges between partitions. But for smaller graphs since PMETIS is constrained by number of partitions it will have to partition the graph into given number of partitions irrespective of increase in number of cross partition edges. This increases the chance of communities to be split across partitions during the partitioning process. Since our method ignore the cross partition edges in the first iteration the final community quality will be significantly effected for very small graphs.

As explained in Section IV our approach parallelize the first iteration of the Louvain method. Amdahl's law bounds our speedups based on the percentage execution time of this first iteration compared with the total execution time. To model the expected speedups for our approach timing results were collected for each iteration of the sequential implementation for different graph sizes. Figure 6 shows the predicted speedups for the parallel implementation and Figure 7 plots the change in difference between expected and actual speedups. Expected speedups were calculated in an optimistic manner ignoring the communication times and assuming linear run times with the graph size [10]. We can see that for all graph sizes the difference becomes larger with the graph size. This is expected due to the increase in overheads, but interestingly we see super linear speedups for almost all the graphs with initial small number of MPI processors. This is mainly due to the partitioning process where after partitioning the convergence rate (number of iterations required in the first step of Louvain to reach the local maximum modularity) of Louvain method iteration within the partition becomes higher than the expected. This is mainly due to the reduction in search space of the first step of Louvain method since we ignore cross partition edges in the first iteration.

We also evaluated the initial three vertex ordering strategies on the same graph data sets (cf. Section IV). Strategy 1 and 3 affected the overall run time in a negative way. Convergence rate of the algorithm reduced dramatically when first strategy was used. The computation complexity of the 3rd strategy

outperformed the improvement in convergence rage. As shown in Figure 8 we were able to get some marginal performance improvement when using the second strategy. But the improvement in convergence rate reduced when we increase the number of graph partitions.

We also conducted some preliminary experiments on a real world social network [3] with $\approx$ 16 million vertices and 30 million edges. Initial results indicated the same behavior with our approach, i.e., $\approx$ 2.6 speed up with 2 processors while final modularity of serial and parallel versions being 0.7162 and 0.7051 respectively.

VI. Conclusions And Future Work

Community detection in large graphs is receiving increasingly traction due to the unprecedented growth in internet and online social networks. Our work combines an existing graph partitioning technique which minimizes the cross partition edges, with the Louvain community detection method. We specifically target the costly first iteration of the sequential algorithm. Results showed our approach scales for large graphs giving significant performance improvements. We show that processing graph partitions independently in the first iteration ignoring cross partition edges does not impact to the quality of the final result.

Our evaluation on different vertex ordering strategies suggested that ordering the vertices in the descending order of edge degree can further improve the convergence rate of the Louvain method while giving the same final modularity.

We plan to further evaluate our approach along few directions. First, the final modularity of our method is highly dependent on the used graph partitioner. We plan to evaluate our method on few different graph partitioners and draw some conclusions that can help users decide on the best partitioning method they should use for Louvain. Second, we will explore strategies to improve the modularity in case there are communities that span across graph partitions by performing some communication between graph partitions.

In this work we did not perform extensive experiments on real world graphs. Even though preliminary results on real world graphs shows our approach holds, we would like to evaluate this approach on different classes of real world graphs with known ground truth communities [22].

Acknowledgment

This work was supported by a research grant from the DARPA XDATA grant no. FA8750-12-2-0319. Authors would like to thank Prof. Cauligi Raghavendra of University of Southern California, Jonathan Larson of Sotera Defence and Alok Kumbhare for their feedback.

References

[1] S. Fortunato, "Community detection in graphs," 2010, uRL: http://arxiv.org/abs/0906.0612 (accessed May 13, 2014).

[2] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 85–98.

[3] http://snap.stanford.edu/data/soc-pokec.html

[3] S. Asur, S. Parthasarathy, and D. Ucar, "An event-based framework for characterizing the evolutionary behavior of interaction graphs," *ACM Trans. Knowl. Discov. Data*, vol. 3, no. 4, pp. 16:1–16:36, Dec. 2009.

[4] twitter blog, "New tweets per second record, and how," 2013, uRL: https://blog.twitter.com/2013/new-tweets-per-second-record-and-how (accessed July 19, 2014).

[5] A. Pothen, "Graph partitioning algorithms with applications to scientific computing," in *Parallel Numerical Algorithms*. Springer, 1997, pp. 323–368.

[6] Y. Simmhan, A. G. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. S. Raghavendra, and V. K. Prasanna, "Goffish: A sub-graph centric framework for large-scale graph analytics," in *Proceedings of the 20th International Conference on Parallel Processing*, ser. EuroPar '14, 2014, p. accepted.

[7] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, ser. SSDBM. New York, NY, USA: ACM, 2013, pp. 22:1–22:12.

[8] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," in *Hadoop Summit*, 2011.

[9] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998.

[10] 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*, no. 10, p. P10008 (12pp), 2008.

[11] M. E. J. Newman, "Modularity and community structure in networks," *Proceedings of the National Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[12] 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.

[13] G. Karypis and V. Kumar, "Parallel multilevel graph partitioning," in *Proceedings of the 10th International Parallel Processing Symposium*, ser. IPPS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 314–319.

[14] S. Kirmani and P. Raghavan, "Scalable parallel graph partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 51:1–51:10.

[15] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," *CoRR*, vol. abs/1404.4797, 2014.

[16] 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.

[17] J. Riedy, D. A. Bader, and H. Meyerhenke, "Scalable multi-threaded community detection in social networks," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012, pp. 1619–1628.

[18] S. Bhowmick and S. Srinivasan, "A template for parallelizing the louvain method for modularity maximization," in *Dynamics On and Of Complex Networks, Volume 2*, ser. Modeling and Simulation in Science, Engineering and Technology. Springer New York, 2013, pp. 111–124.

[19] C. Staudt and H. Meyerhenke, "Engineering high-performance community detection heuristics for massive graphs," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct 2013, pp. 180–189.

[20] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 35, pp. 75 – 174, 2010.

[21] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[22] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, ser. MDS '12. New York, NY, USA: ACM, 2012, pp. 3:1–3:8.