Overcoming MPI Communication Overhead for Distributed Community Detection

Naw Safrin Sattar(✉) and Shaikh Arifuzzaman(✉)

Department of Computer Science, University of New Orleans,
New Orleans, LA 70148, USA
{nsattar,smarifuz}@uno.edu

Abstract. Community detection is an important graph (network) analysis kernel used for discovering functional units and organization of a graph. Louvain method is an efficient algorithm for discovering communities. However, sequential Louvain method does not scale to the emerging large-scale network data. Parallel algorithms designed for modern high performance computing platforms are necessary to process such network big data. Although there are several shared memory based parallel algorithms for Louvain method, those do not scale to a large number of cores and to large networks. One existing Message Passing Interface (MPI) based distributed memory parallel implementation of Louvain algorithm has shown scalability to only 16 processors. In this work, first, we design a shared memory based algorithm using Open MultiProcessing (OpenMP), which shows a 4-fold speedup but is only limited to the physical cores available to our system. Our second algorithm is an MPI-based distributed memory parallel algorithm that scales to a moderate number of processors. We then implement a hybrid algorithm combining the merits from both shared and distributed memory-based approaches. Finally, we incorporate a parallel load balancing scheme, which leads to our final algorithm DPLAL (Distributed Parallel Louvain Algorithm with Load-balancing). DPLAL overcomes the performance bottleneck of the previous algorithms with improved load balancing. We present a comparative analysis of these parallel implementations of Louvain methods using several large real-world networks. DPLAL shows around 12-fold speedup and scales to a larger number of processors.

Keywords: Community detection · Louvain method · Parallel algorithms · MPI · OpenMP · Load balancing · Graph mining

1 Introduction

Parallel computing plays a crucial role in processing large-scale graph data [1,2,5,27]. The problem of community detection in graph data arises in many scientific domains [11], e.g., sociology, biology, online media, and transportation.

A. Majumdar and R. Arora (Eds.): SCEC 2018, CCIS 964, pp. 77–90, 2019.
https://doi.org/10.1007/978-981-13-7729-7_6

Due to the advancement of data and computing technologies, graph data is growing at an enormous rate. For example, the number of links in social networks [14,26] is growing every millisecond. Processing such graph big data requires the development of parallel algorithms [1–5]. Existing parallel algorithms are developed for both shared memory and distributed memory based systems. Each method has its own merits and demerits. Shared memory based systems are usually limited by the moderate number of available cores [18]. The increase in physical cores is restricted by the scalability of chip sizes. On the other hand, a large number of processing nodes can be used in distributed-memory systems. Although distributed memory based parallelism has the freedom of communicating among processing nodes through passing messages, an efficient communication scheme is required to overcome communication overhead. We present a comparative analysis of our shared and distributed memory based parallel Louvain algorithms, their merits and demerits. We also develop a hybrid parallel Louvain algorithm using the advantage of both shared and distributed memory based approaches. The hybrid algorithm gives us the scope to balance between both shared and distributed memory settings depending on available resources. Load balancing is crucial in parallel computing. A straight-forward distribution with an equal number of vertices per processor might not scale well [2]. We also find that load imbalance also contribute to a higher communication overhead for distributed memory algorithms [4]. A dynamic load balancing [3,25] approach can reduce the idle times of processors leading to increased speedup. Finding a suitable load balancing technique is a challenge in itself as it largely depends on the internal properties of a network and the applications [21]. We present DPLAL, an efficient algorithm for distributed memory setting based on a parallel load balancing scheme and graph partitioning.

2 Related Work

There exists a rich literature of community detection algorithms [6–8,15,16,20,24,27]. Louvain method [7] is found to be one of the most efficient sequential algorithms [15,16]. In recent years, several works have been done for paralleling Louvain algorithm and a majority of those are shared memory based implementations. These implementations demonstrate only a moderate scalability. One of the fastest shared memory implementations is Grappolo software package [12,17], which is able to process a network with 65.6M vertices using 20 compute cores. One of the MPI based parallel implementations [27] of Louvain method reported scaling for only 16 processors. Later, in [10] the authors could run large graphs with $1,000$ processing cores for their MPI implementation but did not provide a comprehensive speedup results. Their MPI+OpenMP implementation demonstrated about 7-fold speedup on $4,000$ processors. But the paper uses a higher threshold in lower levels in Louvain method to terminate the level earlier and thus minimized the time contributing to their higher speedup. The work also lacks on the emphasis on graph partitioning and balancing load among the processors. This is a clear contrast with our work where we focused on load balancing issue among others. Our work achieves comparable (or better in many cases) speedups using a significantly fewer number of processors than the work in [10].

3 Background

In this section, we present the Louvain algorithm in brief and discuss the computational model of our parallel algorithms. Note that we use the words *vertex* and *node* interchangeably in the subsequent discussions of this paper. The same is the case for the words *graph* and *network*.

3.1 Louvain Algorithm

Louvain algorithm [7] detects community based on modularity optimization. It demonstrates better performance than other community detection algorithms in terms of computation time and quality of the detected communities [15]. Modularity is calculated using Eq. 1.

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

Here,

Q = Modularity

A_{ij} = Link weight between nodes i and j

m = Total link weight in the network

k_i = Sum of the link weights attached to node i

c_i = Community to which node i is assigned

$\delta\left(c_i,\ c_j\right)$ = Kronecker delta Value is 1 when nodes i and j are assigned to the same community. Otherwise, the value is 0.

Louvain algorithm has 2 Phases:

- **Modularity Optimization:** This step looks for “small” communities by local optimization of modularity.
- **Community Aggregation:** This step aggregates nodes of the same community to form a super-node and thus create a new smaller network to work on in the next iteration.

Details on the above steps can be found in [7].

3.2 Computational Model

We develop our shared memory based parallel algorithm using Open Multi-Processing (OpenMP) library. Then, we develop our distributed memory based parallel algorithm using Message Passing Interface (MPI). Both MPI and OpenMP have been inscribed in our Hybrid Algorithm. At last, in DPLAL, along with MPI, we use the graph-partitioner METIS [13] to improve graph partitioning and load balancing.

4 Methodology

We present our parallel Louvain algorithms below. Note that we omitted some of the details of these algorithms for brevity. The pseudocode and functional description of our earlier implementation of shared and distributed memory algorithms can be found in [22].

4.1 Shared Memory Parallel Louvain Algorithm

In shared memory based algorithms, there is a shared address space and multiple threads share this common address space. This shared address space can be used efficiently using lock and other synchronization techniques. The main hindrance behind the shared memory based systems is the limited number of processing cores. We parallelize the Louvain algorithm by distributing the computational task among multiple threads using Open Multi-Processing (OpenMP) framework. (See a detailed description of this algorithm in [22].)

4.2 Distributed Memory Parallel Louvain Algorithm

Distributed memory based algorithms can exploit the power of large computing clusters that are widely available now-a-days. The compute nodes have different memory space. Processors exchange messages among themselves to share information. Such inter-processor communication introduces significant overhead, which needs to be minimized. Another crucial challenge is balancing load among processors. We use Message Passing Interface (MPI) for the implementation of distributed memory based parallel Louvain algorithm. In the first phase, we partition the entire network among the processors. Each processor gets a part of the network. In the second phase, each processor complete its computation independently and does communication with other processors whenever necessary. A particular processor is designated as the root or master. After each level of iteration, all processors communicate with the root processor to compute the modularity value of the full network. A detailed functional description of this approach can be found in [22].

4.3 Hybrid Parallel Louvain Algorithm

We use both MPI and OpenMP together to implement the Hybrid Parallel Louvain Algorithm. The hybrid version gives us the flexibility to balance between both shared and distributed memory system. We can tune between shared and distributed memory depending on available resources. In the multi-threading environment, a single thread works for communication among processors and other threads do the computation.

4.4 Distributed Parallel Louvain Algorithm with Load-Balancing

To implement DPLAL, we use the similar approach as described in Sect. 4.2. In the first phase, we have used well-known graph-partitioner METIS [19] to partition our input graph to distribute among the processors. Depending on METIS output, we adjust the number of processors because METIS does not always create same number of partitions as provided in input. We use both edge-cut and communication volume minimization approaches. An empirical comparison of these approaches is described later in Sect. 6. After partitioning, we distribute the input graph among the processors. For second phase, we follow the same flow as described in the Algorithm in [22]. But we have to recompute each function that has been calculated from input graph. Runtime analysis for each of these functions being used in MPI communication has been demonstrated in Sect. 6. Our incorporation of graph partitioning scheme helps minimize the communication overhead of MPI to a great extent and we get an optimized performance from DPLAL.

5 Experimental Setup and Dataset

We describe our experimental setup and datasets below. We use large-scale compute cluster for working on large real-world graph datasets.

5.1 Execution Environment

We use Louisiana Optical Network Infrastructure (LONI) QB2 [9] compute cluster to perform all the experiments. QB2 is a 1.5 Petaflop peak performance cluster containing 504 compute nodes with over 10,000 Intel Xeon processing cores of 2.8 GHz. We use at most 50 computing nodes with 1000 processors for our experiments.

5.2 Description of Datasets

We have used real-world networks from SNAP [23] depicted in Table 1. We have performed our experimentation on different types of network including social networks, internet, peer-to-peer networks, road networks, network with ground truth communities, and Wikipedia networks. All these networks show different structural and organizational properties. This gives us an opportunity to assess the performance of our algorithms for worst case inputs as well. The size of graphs used in our experiments ranges from several hundred thousands to millions of edges.

6 Results

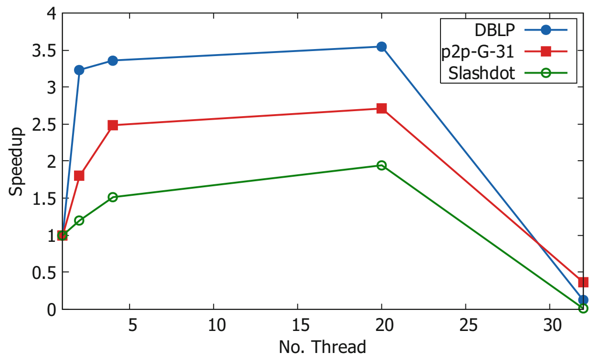
We present the scalability and runtime analysis of our algorithms below. We discuss the trade-offs and challenges alongside.

Table 1. Datasets used in our experimental evaluation.

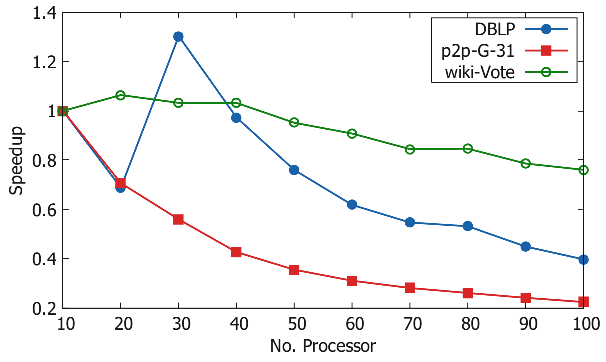
Network	Vertices	Edges	Description
email-Eu-core	1,005	25,571	Email network from a large European research institution
ego-Facebook	4,039	88,234	Social circles ('friends lists') from Facebook
wiki-Vote	7,115	103,689	Wikipedia who-votes-on-whom network
p2p-Gnutella08	6,301	20,777	A sequence of snapshots of the Gnutella peer-to-peer file sharing network for different dates of August 2002
p2p-Gnutella09	8,114	26,013	
p2p-Gnutella04	10,876	39,994	
p2p-Gnutella25	22,687	54,705	
p2p-Gnutella30	36,682	88,328	
p2p-Gnutella31	62,586	147,892	
soc-Slashdot0922	82,168	948,464	Slashdot social network from February 2009
com-DBLP	317,080	1,049,866	DBLP collaboration (co-authorship) network
roadNet-PA	1,088,092	1,541,898	Pennsylvania road network

Speedup Factors of Shared and Distributed Memory Algorithms. We design both shared and distributed memory based algorithms for Louvain methods. The speedup results are shown in Fig. 1a and b. Our shared memory and distributed memory based algorithms achieve speedups of around 4 and 1.5, respectively. The number of physical processing core available to our system is 20. Our shared memory algorithm scales well to this many cores. However, due to the unavailability of large shared memory system, we also design distributed memory algorithm. Further, shared memory algorithms show a limited scalability to large networks as discussed in [6]. Our distributed memory algorithm demonstrates only a minimal speedup for 30 processors. The inter-processor communication severely affects the speedup of this algorithm. We strive to overcome such communication bottleneck by designing hybrid algorithm.

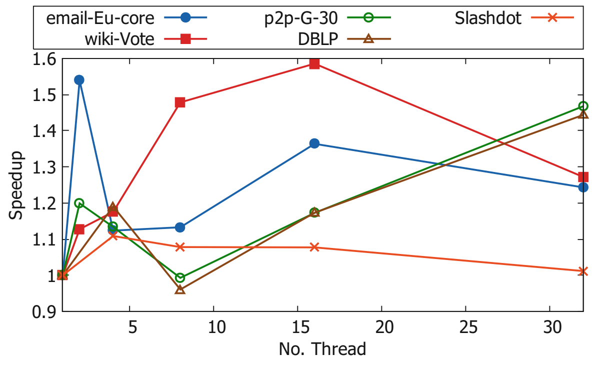
Speedup Factors of Our Hybrid Parallel Algorithm. Our hybrid algorithm tends to find a balance between the above two approaches, shared and distributed memory. As shown in Fig. 1c, we get a speedup of around 2 for the hybrid implementation of Louvain algorithm. The speedup is similar to the MPI implementation. It is evident that in multi-threading environment runtime will decrease as workload is distributed among the threads. But we observe that in some cases, both single and multiple threads take similar time. Even sometimes multiple threads take more time than a single thread. It indicates that hybrid implementation also suffers from the communication overhead problem



(a) Shared Memory Algorithm

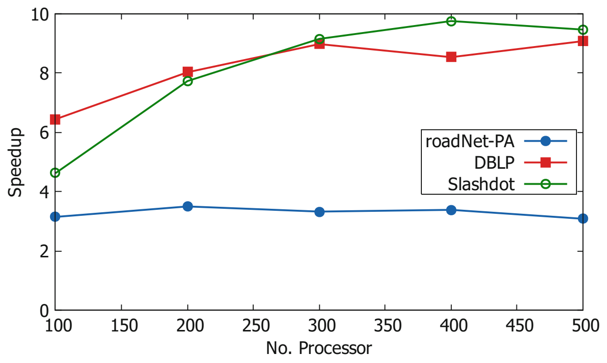


(b) Distributed Memory Algorithm

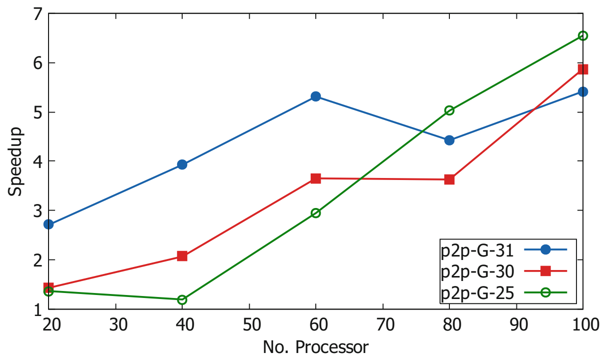


(c) Hybrid Algorithm

Fig. 1. Speedup factors of our parallel Louvain algorithms for different types of networks. Our hybrid algorithm strikes a balance between shared and distributed memory based algorithms.



(a) Speedup results for large graphs



(b) Speedup results for relatively small graphs

Fig. 2. Speedup factors of DPLAL algorithm for different types of networks. Larger networks scale to a larger number of processors.

alike MPI. Communication overhead of distributed memory setting limits the performance of hybrid algorithm as well.

Speedup Factors of Our Improved Parallel Algorithm DPLAL. Our final parallel implementation of Louvain algorithm is DPLAL. This algorithm achieves a speedup factor up-to 12. We reduce the communication overhead in message passing setting to a great extent by introducing a load balancing scheme during graph partitioning. The improved speedup for DPLAL is presented in Fig. 2. For larger networks, our algorithm scales to a larger number of processors. We are able to use around a thousand processors. For smaller networks, the algorithm scales to a couple of hundred processors. It is understandable that for smaller networks, the communication overhead gradually offsets the advantage obtained from parallel computation. However, since we want to use a larger number of processors to work on larger networks, our algorithm in fact has this desirable property. Overall, DPLAL algorithm scales well with the increase in the number of processors and to large networks.

Runtime Analysis: A Breakdown of Execution Times. We present a breakdown of executions times. Figure 3 shows the runtime analysis for our largest network RoadNet-PA. We observe that communication time for *gathering neighbor information* and *exchanging duality resolved community* decreases with increasing number of processors. Communication time for both *exchanging updated community* and *gathering updated community* increases up-to a certain number of processors and after decreasing, the time becomes almost constant. Among all these communications, time to gather communities at the root processor takes maximum time and contribute to the high runtime.

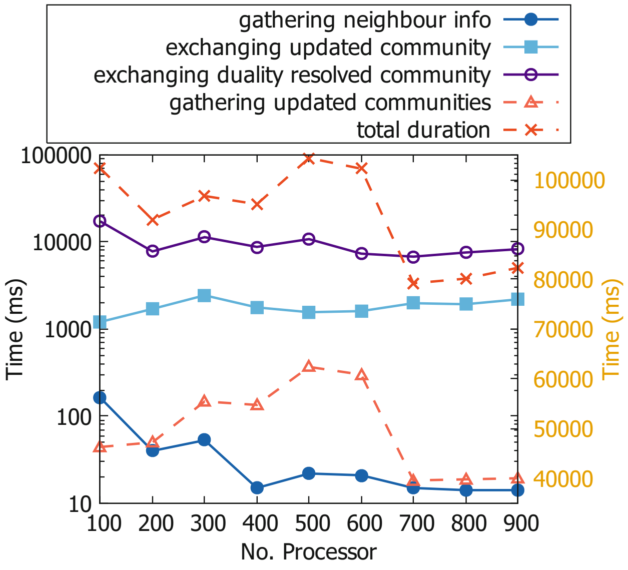


Fig. 3. Runtime analysis of RoadNet-PA graph with DPLAL algorithm for varied number of processors. We show a breakdown of execution times for different modules or functions in the algorithm. Time for *gathering updated communities* and *total duration* are plotted w.r.t the right y-axis.

Number of Processors Versus Execution Time. For many large networks that we experimented on (including the ones in Fig. 2a), we find that those can scale to up to ≈800 processors. We call this number as the *optimum number of processors* for those networks. This optimum number depends on network size. As our focus is on larger networks, to find out the relationship between runtime and network size, we keep the number of processor 800 fixed and run an experiment. As shown in Fig. 4, the communication time for *gathering neighbor info* decreases with growing network size whereas both time for *gathering updated communities* and *exchanging duality resolved community* increase. Communication time for *exchanging updated community* increases up-to a certain point and then starts decreasing afterwards. For larger networks ($>8K$), total runtime increases proportionately with growing network size. As smaller graphs do not scale to 800 processors, these do not follow the trend, but it can be inferred that these will behave the same way for their optimum number of processors.

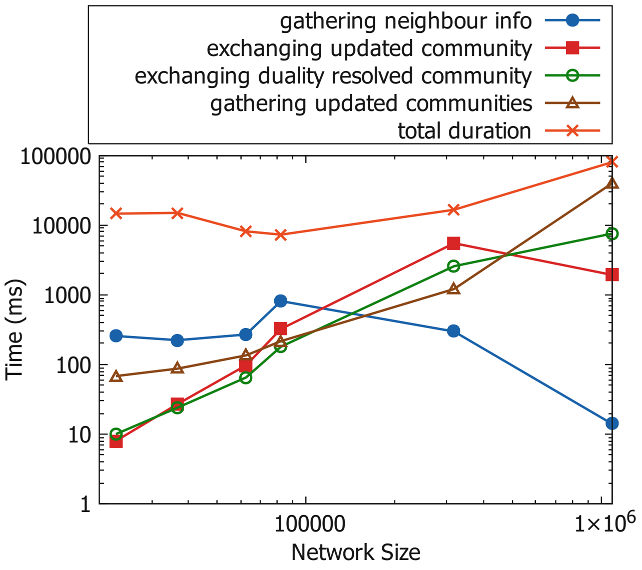


Fig. 4. Increase in runtime of DPLAL algorithm with an increase in the sizes of the graphs keeping the number of processors fixed.

METIS Partitioning Approaches. We also compare the METIS partitioning techniques, between edge-cut and communication volume minimization, to find out the efficient approach for our algorithm. Figure 5 shows the runtime comparison between edge-cut and communication volume minimization techniques. We find that the communication volume minimization approach always takes similar or higher time than that of edge-cut partitioning. So, in our subsequent experimentation, we have used edge-cut partitioning approach.

7 Performance Analysis

We present a comparative analysis of our algorithms, its sequential version, and another existing distributed memory algorithm.

7.1 Comparison with Other Parallel Algorithms

We compare the performance of DPLAL with another distributed memory parallel implementation of Louvain method given in Wickramaarachchi et al. [27].

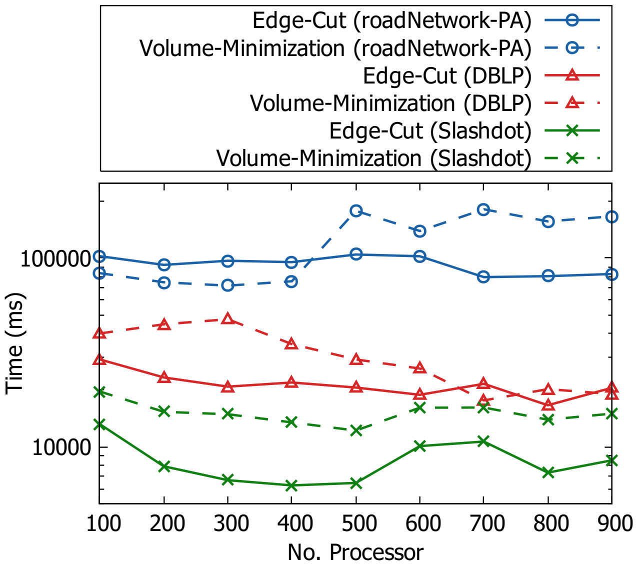


Fig. 5. Comparison of METIS partitioning approaches (edge-cut versus communication volume minimization) for several networks. The edge-cut approach achieves better runtime efficiency for the above real-world networks.

For a network with 500,000 nodes, Wickramaarachchi et al. achieved a maximum speedup of 6 whereas with DPLAL for a network with 317,080 nodes we get a speedup of 12 using 800 processors. The largest network processed by them has 8M nodes and achieved a speedup of 4. Our largest network achieves a comparable speedup (4-fold speedup with 1M nodes). The work in [27] did not report runtime results so we could not compare our runtime with theirs directly. Their work reported scalability to only 16 processors whereas our algorithm is able to scale to almost a thousand of processors.

7.2 Comparison with Sequential Algorithm

We have compared our algorithms with the sequential version [7] to analyze the accuracy of our implementations. Deviation of the number of communities between sequential and our implementations is represented in Table 2. The deviation is negligible compared to network size. The number of communities is not constant and they vary because of the randomization introduced in the Louvain algorithm. Table 2 gives an approximation of the communities.

Although shared memory based parallel Louvain has the least deviation, the speedup is not remarkable. Whereas, DPLAL shows a moderate deviation but its speedup is 3 times of that of shared parallel Louvain algorithm.

Table 2. Deviation of the number of communities for different parallel Louvain Algorithms from the sequential algorithm.

Algorithm	Network			
	com-DBLP		wiki-Vote	
	Comm. No.	Dev. (%)	Comm. No.	Dev. (%)
Sequential	109,104	–	1,213	–
Shared	109,102	.0006	1,213	0
Distributed	109,441	0.106	1,216	0.042
Hybrid	104,668	1.39	1,163	0.71
DPLAL	109,063	0.0129	1,210	0.042

8 Conclusion

Our parallel algorithms for Louvain method demonstrate good speedup on several types of real-world graphs. As instance, for DBLP graph with 0.3 million nodes, we get speedups of around 4, 1.5 and 2 for shared memory, distributed memory, and hybrid implementations, respectively. Among these three algorithms, shared memory parallel algorithm gives better speedup than others. However, shared memory system has limited number of physical cores and might not be able to process very large networks. A large network often requires distributed processing and each computing node stores and works with a part of the entire network. As we plan to work with networks with billions of nodes and edges, we work towards the improvement of the scalability of our algorithms by reducing the communication overhead. We have identified the problems for each implementation and come up with an optimized implementation **DPLAL**. With our improved algorithm DPLAL, community detection in DBLP network achieves a 12-fold speedup. Our largest network, roadNetwork-PA has 4-fold speedup for same number of processors. With increasing network size, number of processor also increases. We will work with larger networks increasing the number of processors in our future work. The optimum number of processor largely depends on the network size. We will also experiment with other load-balancing schemes to find an efficient load balancing scheme to make DPLAL more scalable. We also want to eliminate the effect of small communities that create misconception to understand the community structure and its properties. Further, we will explore the effect of node ordering (e.g., degree based ordering, random ordering) on the performance of parallel Louvain algorithms.

Acknowledgements. This work has been partially supported by Louisiana Board of Regents RCS Grant LEQSF(2017-20)-RDA- 25 and University of New Orleans ORSP Award CON000000002410. We also thank the anonymous reviewers for the helpful comments and suggestions to improve this paper.

References

1. Arifuzzaman, S., Khan, M.: Fast parallel conversion of edge list to adjacency list for large-scale graphs. In: 2015 Proceedings of the 23rd Symposium on High Performance Computing, pp. 17–24. Society for Computer Simulation International (2015)
2. Arifuzzaman, S., Khan, M., Marathe, M.: PATRIC: a parallel algorithm for counting triangles in massive networks. In: Proceedings of the 22nd ACM International Conference on Information & Knowledge Management, pp. 529–538. ACM (2013)
3. Arifuzzaman, S., Khan, M., Marathe, M.: A fast parallel algorithm for counting triangles in graphs using dynamic load balancing. In: 2015 IEEE International Conference on Big Data (Big Data), pp. 1839–1847. IEEE (2015)
4. Arifuzzaman, S., Khan, M., Marathe, M.: A space-efficient parallel algorithm for counting exact triangles in massive networks. In: 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), pp. 527–534. IEEE (2015)
5. Arifuzzaman, S., Pandey, B.: Scalable mining and analysis of protein-protein interaction networks. In: 3rd International Conference on Big Data Intelligence and Computing (DataCom 2017), pp. 1098–1105. IEEE (2017)
6. Bhowmick, S., Srinivasan, S.: A template for parallelizing the Louvain method for modularity maximization. In: Mukherjee, A., Choudhury, M., Peruani, F., Ganguly, N., Mitra, B. (eds.) Dynamics on and of Complex Networks, vol. 2, pp. 111–124. Springer, New York (2013). https://doi.org/10.1007/978-1-4614-6729-8_6
7. Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. J. Stat. Mech.: Theory Exp. **2008**(10), P10008 (2008)
8. Clauset, A., Newman, M.E., Moore, C.: Finding community structure in very large networks. Phys. Rev. E **70**(6), 066111 (2004)
9. Documentation—user guides—qb2. http://www.hpc.lsu.edu/docs/guides.php?system=QB2
10. Ghosh, S., et al.: Distributed Louvain algorithm for graph community detection. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 885–895. IEEE (2018)
11. Girvan, M., Newman, M.E.: Community structure in social and biological networks. Proc. Nat. Acad. Sci. **99**(12), 7821–7826 (2002)
12. Halappanavar, M., Lu, H., Kalyanaraman, A., Tumeo, A.: Scalable static and dynamic community detection using Grappolo. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6. IEEE (2017)
13. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
14. Kwak, H., Lee, C., Park, H., Moon, S.: What is twitter, a social network or a news media? In: Proceedings of the 19th International Conference on World Wide Web, pp. 591–600. ACM (2010)
15. Lancichinetti, A., Fortunato, S.: Community detection algorithms: a comparative analysis. Phys. Rev. E **80**(5), 056117 (2009)

16. Leskovec, J., Lang, K.J., Mahoney, M.: Empirical comparison of algorithms for network community detection. In: Proceedings of the 19th International Conference on World Wide Web, pp. 631–640. ACM (2010)
17. Lu, H., Halappanavar, M., Kalyanaraman, A.: Parallel heuristics for scalable community detection. Parallel Comput. **47**, 19–37 (2015)
18. McCalpin, J.D., et al.: Memory bandwidth and machine balance in current high performance computers. In: 1995 IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pp. 19–25 (1995)
19. Karypis Lab: METIS - serial graph partitioning and fill-reducing matrix ordering. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
20. Raghavan, U.N., Albert, R., Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. Phys. Rev. E **76**(3), 036106 (2007)
21. Raval, A., Nasre, R., Kumar, V., Vadhiyar, S., Pingali, K., et al.: Dynamic load balancing strategies for graph applications on GPUs. arXiv preprint arXiv:1711.00231 (2017)
22. Sattar, N., Arifuzzaman, S.: Parallelizing Louvain algorithm: distributed memory challenges. In: 2018 IEEE 16th International Conference on Dependable, Autonomic and Secure Computing (DASC 2018), pp. 695–701. IEEE (2018)
23. Stanford large network dataset collection. https://snap.stanford.edu/data/index.html
24. Staudt, C.L., Meyerhenke, H.: Engineering parallel algorithms for community detection in massive networks. IEEE Trans. Parallel Distrib. Syst. **27**(1), 171–184 (2016)
25. Talukder, N., Zaki, M.J.: Parallel graph mining with dynamic load balancing. In: 2016 IEEE International Conference on Big Data (Big Data), pp. 3352–3359. IEEE (2016)
26. Ugander, J., Karrer, B., Backstrom, L., Marlow, C.: The anatomy of the Facebook social graph. arXiv preprint arXiv:1111.4503 (2011)
27. Wickramaarachchi, C., Frincuy, M., Small, P., Prasannay, V.: Fast parallel algorithm for unfolding of communities in large graphs. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6. IEEE (2014)