2018 IEEE 16th Int. Conf. on Dependable, Autonomic & Secure Comp., 16th Int. Conf. on Pervasive Intelligence & Comp., 4th Int. Conf. on Big Data Intelligence & Comp., and 3rd Cyber Sci. & Tech. Cong.

Parallelizing Louvain Algorithm: Distributed Memory Challenges

Naw Safrin Sattar and Shaikh Arifuzzaman
Department of Computer Science
University of New Orleans, New Orleans, LA 70148 USA
Email: {nsattar, smarifuz}@uno.edu

***Abstract*—Louvain algorithm is a well-known and efficient method for detecting communities or clusters in social and information networks (graphs). The emergence of large network data necessitates parallelization of this algorithms for high performance computing platforms. There exist several shared-memory based parallel algorithms for Louvain method. However, those algorithms do not scale to a large number of cores and large networks. Distributed memory systems are widely available nowadays, which offer a large number of processing nodes. However, the existing only MPI (message passing interface) based distributed-memory parallel implementation of Louvain algorithm has shown scalability to only 16 processors. In this paper, we implement both shared- and distributed-memory based parallel algorithms and identify issues that hinder scalability. In our shared-memory based algorithm using OpenMP, we get 4-fold speedup for several real-world networks. However, this speedup is limited only by the physical cores available to our system. We then design a distributed-memory based parallel algorithms using message passing interface. Our results demonstrate an scalability to a moderate number of processors. We also provide an empirical analysis that shows how communication overhead poses the most crucial threat for deisgning scalable parallel Louvain algorithm in a distributed-memory setting.**

***Keywords*-community detection; Louvain algorithm; parallel algorithm; distributed-memory; MPI; graph mining; social networks**

I. Introduction

A community or cluster in a network is a subset of nodes where there are more inside connections than outside. Community detection [2], [6], [4] in networks is an important problem in network (graph) mining. Large networks emerging from online social media and other scientific disciplines make the problem of community deteciton very challenging. For example, social networks Facebook and Twitter have billions of users [17], [18]. In the modern era of big data, dealing with such large networks require parallel algorithms [12], [13], [11], [14]. There exist both shared- and distributed-memory based parallel algorithms. Scalability with shared-memory based systems is usually limited by the moderate number of available cores, whereas with distributed-memory sytems, it is possible to utilize a large number of processing nodes [1]. Conventional multi-core processors can exploit the advantages of shared-memory based parallel programming. Increasing physical cores gets limited due to the scalable chip size restriction. Shared global address space size is also limited because of memory constraint. On the contrary, distributed-memory based parallelism has the freedom of communication among processing nodes through passing messages. Implementation of distributed-memory based algorithms is challenging considering the need for an efficent communication scheme. In this paper, first, we figure out the limitations of shared-memory based parallelism of Louvain algorithm for community detection. We then develop an MPI based parallel Louvain algorithm and identify the challenges to scalability. Finally, we propose a novel solution to increase speedup factors for parallel Louvain algorithm in a distributed setting.

II. Related Works

The problem of detecting communities in networks has a long history [15], [2], [4], [6], [16], [19], [20]. As identifed in [15], Louvain algorithm [2] is one of the efficient sequential algorithms. However, sequential algorithms are unsuitable to process large networks with millions to billions of nodes and edges in a reasonable amount of time [13]. Thus, in recent years, there has been a shift of focus towards the development of parallel algorithms. There exist several parallel algorithms based on Louvain method [3], [5], [6]– most of those algorithms are based on shared-memory systems. Not many work has been done for distributed-memory systems.

A template has been proposed to parallelize the Louvain method for modularity maximization with a shared-memory parallel algorithm in [3] using OpenMP. Maximum modularity has been found by parallel reduction. The paper combines communities to supervertices using parallel mergesort. The authors run their experimental setup on two sets of LFR benchmarks of 8,000 and 10,000 vertices which are small numbers compared to large real-world networks [10]. In [5], another shared-memory implementation is provided using a hierarchical clustering method with adaptive parallel thread assignment. The authors have showed the granularity of threads could be obtained adaptively at run-time based on the information to get the maximal acceleration in the parallelization of Louvain algorithm. They have computed the gained modularity of adding a neighbor node to the community by assigning some threads in parallel. Dynamic thread assignment of OpenMP has been disabled to let the algorithm adaptively choose the number of threads. For upto

32 cores, the speedup is not significant compared to the previous implementations, e.g., PLM [6] [7] and CADS [8].

To the best of our knowledge, the only MPI based distributed-memory algorithm is proposed in [4]. The network is partitioned using PMETIS . The work parallelizes only the first level of Louvain algorithm and peforms sequential execution for the later levels. Each MPI process locally ignores cross-partition edges, which decreases the quality of the detected communities. The authors use three different vertex ordering strategies but none of those are effective enough in terms of improving the performance of the algorithm. Although the paper demonstrates scalability up to only 16 to 32 processors.

III. Preliminaries

Notations, definitions, and computational model used in this paper are described below.

A. Notation

The network is denoted by $G(V, E)$, where V and E are the sets of vertices and edges, respectively. Vertices are labeled as $V_0, V_1, \dots, V_{n-1}$. We use the words node and vertex interchangeably, same for links and edges. P is the number of processors used in the computation, denoted by $P_0, P_1, \dots, P_{N-1}$, where $0, 1, 2, \dots, N-1$ refers to the rank of a processor. Terms frequently used throughout the paper are enlisted in Table I.

Table I: Terminologies used in the paper

Symbol	Meaning
$G(V, E)$	Graph network with V = set of vertices and E = set of edges
in [c]	Sum of the weights of the links inside community c
tot [c]	Sum of the weights of the links incident to vertices in community c
n2c [i]	Community label of vertex i
d(i,c)	Number of links from vertex i to community c
N	Total number of processors (World size)
$n = \|V\|$	Total number of vertices (Network size)

B. Louvain Algorithm for Community Detection

Louvain is a simple, heuristic method to extract the community structure of large networks based on modularity optimization [2]. It outperforms all other known community detection method in terms of computation time and quality of the detected communities [15]. Modularity Q is calculated using Equation 1, where $-1 < Q < 1$. The meanings for different quantities are described in Table II. Louvain algorithm is divided in two phases, which are iteratively repeated .

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

Table II: Symbols used for calculating Modularity in Equation 1 and Equation 2

Symbol	Meaning
Q	Modularity
A	Usual adjacency matrix
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
$\frac{k_i}{2m}$	Average fraction of weight that would be assigned to node j, if node i assigned its link weight randomly to other nodes in proportion to their own link weights
$A_{ij} - \frac{k_i k_j}{2m}$	How strongly nodes i and j are connected in the real network, compared to how strongly connected we would expect them to be in a random network
c_i	Community to which node i is assigned
$\delta(c_i,\ c_j)$	Kronecker delta. Value is 1 when nodes i and j are assigned to the same community. Otherwise, the value is 0
ΔQ	Gain in Modularity
$\sum_{in}$	Sum of the weights of the links inside community C
$\sum_{tot}$	Sum of the weights of the links incident to nodes in community C
$k_{i,in}$	sum of the weights of the links from node i to nodes in C

1) First Phase : For each node i the neighbours j of i are considered and the gain in modularity, ΔQ that would take place by removing i from its community and by placing it in the community of j is evaluated. ΔQ obtained by moving an isolated node i into a community C is computed using Equation 2.

$$\Delta Q = \left[\frac{\sum_{in} + k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m} \right)^2 \right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m} \right)^2 - \left(\frac{k_i}{2m} \right)^2 \right] \tag{2}$$

The symbolic meanings are given in Table II. When i is removed from its community C a similar equation is used. This continues repeatedly until no further improvement is achieved. This phase comes to an end when a local maxima of the modularity is attained.

2) Second Phase: In this phase, a new network is formed whose nodes are the communities found during the first phase. The weights of the links between two new nodes are given by the sum of the weight of the links between nodes in the corresponding two communities. Links between nodes of the same community lead to self-loops for this community in the new network.

After completion of second phase, the first phase of the algorithm is reapplied to the resulting weighted network and iteration continues as long as positive gain in modularity is achieved.

C. Computational Model

At first, we develop shared-memory based parallel algorithm using Open Multi-Processing (OpenMP) to eliminate the limitations of [3] with a different approach. Shared-memory based algorithms have some own limitations due to limited number of physical processing cores hindering high speedup. Therefore, we develop parallel algorithm for message passing interface (MPI) based distributed-memory parallel systems, where each processor has its own local memory. The processors do not have any shared memory, one processor cannot directly access the local memory of another processor, and the processors communicate via exchanging messages using MPI.

IV. Methodology

A. Overview of the Shared-Memory Algorithm

In our shared memory implementation, we parallelize the Louvain algorithm computational task-wise in a straight-forward approach. Whenever there is a need to iterate over the full network or even the neighbors of a node, considering the large network size, the work is done by multiple threads to minimize the workload and do the computation faster.

B. Overview of the Distributed-Memory Algorithm

Our aim is to compute the communities of the full network in a distributed manner. To serve the purpose, we distribute the full network among the processors in a balanced way, so that each processor can do its computation in a reasonable time and no one should wait for another. It is necessary because after each level of computation, they have to communicate with the root processor to generate the modularity of the full network. After each level of iteration, the network size decreases gradually and when the network size is considerably small, the final level is computed by a single processor which acts similar to the Louvain Sequential Algorithm.

1) Graph Partitioning : Let N be the number of processors used in the computation. The network is partitioned into N partitions, and each processor, P_i is assigned one such partition $G_i(V_i, E_i)$. Processor P_i performs computation on its partition G_i. The network data is given as input in a single disk file. While partitioning $G(V, E)$, if the vertex-id does not start with 0, then the vertices are renumbered from 0 to $n_i - 1$. The network is partitioned depending on the number of vertices of the network such that each processor gets equal number of vertices. N can vary depending on the system configuration and available resources. n is also of varied range. So, we cannot divide the vertices equally among processors when $n \mod N! = 0$ We distribute the remaining nodes starting with processor P_0 and continue up to processor $P_k(0 <= k < N)$, as long as the remainder lasts. Each processor has its own part of the network necessary to compute the communities in the partial network. It is a very naive partitioning technique.

2) Community Detection: We have parallelized the sequential algorithm in such a way that each processor can compute its partial network‘s community with minimized communication among the processors. The following information are needed for each processor to complete its part of computation.

- Degree of each vertex within the partition
- Neighbors of each vertex
- Weight associated with each neighbor

In first phase, each processor scans through all neighboring vertices and identifies those in different processors. It then gathers the mentioned information by message passing among those processors. Then each processor locally computes the modularity of the partial network and does community detection. After computation, it sends information of each vertexs community to the processor acting as the root. The Root processor needs the value of *in*, *tot* arrays and total weight of the network to compute modularity of the full network. This full process is iterated several times as long as there is increase in modularity.

The output is stored as (node,community) for each level of iteration. It is also stored in an adjacency matrix format by which the graph can be visualized using Python‘s **networkx** library.

Algorithm 1 represents the pseudo-code of our approach. All the steps from Section IV-B2a to IV-B2f are done for each level of computation and repeated as long as we get increase in modularity value.

Algorithm 1: Parallel Louvain using MPI

```
Data: Input Graph G(V,E)
Result: (Vertex, Community) Pair
1  while increase in modularity do
2      G (V, E) is divided into p processes;
3      Each graph_i.bin contains $\lceil \frac{n}{p} \rceil$ vertices and
         corresponding edges in adjacency list format;
4      for Each processor $P_i$ (executing in parallel) do
5          Gather_Neighbour_Info();
6          Compute_Community();
7          Exchange_Updated_Community();
8          Resolve_Community _Duality();
9          Exchange_Duality _Resolved _Community();
10         Find_Unique_Communities();
11         Compute_Modularity();
12         Generate_NextLevel_Graph();
13         if $number\_of\_communities < i$ then
14             $i \leftarrow \frac{number\_of\_communities}{2}$;
15         end
16     end
17 end
```

a) Collection of Neighbour Information: Each processor P_i reads the input graph and initialize the arrays *tot*, *in* and *n2c*. P_i then scans through the neighbor list of all vertices and finds out the neighbors which do not belong to current processor. So, each processor communicates with other processors to collect degree and neighbor list with weight of each vertex in the neighbor list for calculation in later steps.

b) Local Community Update: Update of community of each vertex locally is done in this step. Each processor, P_i completes this step individually and locally updates the community of the vertices belonging to it and does not require any communication with other processors. A random vertex, v is chosen from the list of vertices. Then, the set of neighboring communities of v is computed with the information from Section IV-B2a. The number of links from v to all its neighboring community is computed and stored. Next, v is removed from its current community, C_{old}. Now, the modularity gain is calculated for all of its neighboring communities. If gain is positive, and the gain is maximum for community, C, then v's community is updated to C. Otherwise, v gets back to C_{old}. These remove() and insert() operations update the *in* and *tot* arrays implicitly. As, each processor is locally updating the community, we need to keep trace if C belongs to processor P_i or not. If C belongs to another processor, P_j, we simply store $v_{comm} = C$ and $v_{link} = d_{v,C}$ where $d_{v,C}$= number of links from vertex v to community C for using in section IV-B2c. Selection of vertex continues until all vertices from the list of vertices are covered.

c) Global Community Update: In this step, the update of community is done globally among all processors. Now, Processor, P_i sends Processor, P_j the following data:

- Vertex, v belonging to Processor P_i
- v's community, C belonging to Processor P_j
- $d_{v,C}$
- Number of self-loops of v
- Weighted degree of v

$$P_i \underrightarrow{msg} P_j; v \in P_i, C \in P_j$$

$$msg \ni v, C, d_{v,C}, v_{self-loop}, v_{weighted-degree}$$

Upon receiving the data, the v is inserted to its community C and this insertion updates *in* and *tot* arrays internally.

d) Resolving Community Duality: There remains an inconsistency to calculate the total number of communities in the full network. If vertices a and b belong to the same community, the community-label can be either a or b. The community-label does not have any effect on the result. But in current scenario, same community will be counted twice as a changes its community to b and vice-versa. To eliminate this problem, we keep the higher-number-labelled communities and change the lower-number-labelled communities to higher ones. If $a > b$, vertex a's community is changed again to a and vertex b retains its community a.

If vertices a and b belong to different processors, communication among processors is required to circulate the update. We update the vertices' communities those belong to current processor and track the communities those belong to other processors and communication is required. Processor, P_i send to other processor, P_j the community-id, $community_{old}$ to get the updated community of that id. P_j after receiving the message, sends back to P_i the current community of $community_{old}$. After receiving info from P_j, P_i updates the communities of the vertices those need update. In this communication step, subsequent send-receive is done.

$$P_i \underrightarrow{msg1} P_j; P_j \underrightarrow{msg2} P_i$$

$$msg1 \ni Community; Community \in P_j$$

$$msg2 \ni Community, n2c[Community]$$

e) Computation of Modularity: Each processor, P_i , where $0 <= i < N$ finds out all the unique communities by iterating over all of its vertices belonging to it. To calculate total unique communities in the world, each Processor, P_k, where $(1 <= k < N)$ sends its unique community list to root processor, P_0. P_0 then merges all the unique communities received from each processor, P_k and eliminates duplicate ones. P_k also sends to P_0 values calculated from *in*, *tot* arrays and $total\_weight$ of graph-partition for calculation of modularity of the full network.

f) Generation of Next Level Graph: This step is performed by only the root processor, P_0. P_0 renumbers the communities from 0 to $Z - 1$ to formulate the new input graph to be used for next level.

$Z =$ Total number of unique communities after merging and eliminating duplicate.

So, Z is the number of vertices for input graph of next level. The connectivity of edges and corresponding weights are calculated from available data and the new graph is generated.

V. Experimental Setup

A. Environment

We use Louisiana Optical Network Infrastructure (LONI) QB2 compute cluster to perform all the experiments. QB2 [9] 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 up to 100 processors for most of our experiments.

B. Dataset

We have used real-world networks from SNAP dataset [10] as shown in Table III.

Table III: Datasets used in Experiment

Network	Vertices	Edges	Description
email-Eu-core	1005	25,571	Email network from a large European research institution
ego-Facebook	4039	88,234	Social circles ('friends lists') from Facebook
wiki-Vote	7115	103,689	Wikipedia who-votes-on-whom network
p2p-Gnutella08 p2p-Gnutella09 p2p-Gnutella04 p2p-Gnutella25 p2p-Gnutella30 p2p-Gnutella31	6301 8114 10876 22687 36682 62586	20,777 26,013 39,994 54,705 88,328 147,892	A sequence of snapshots of the Gnutella peer-to-peer file sharing network for different dates of August 2002
soc-Slashdot0922	82,168	948,464	Slashdot social network from February 2009
com-DBLP	317080	1,049,866	DBLP collaboration (co-authorship) network

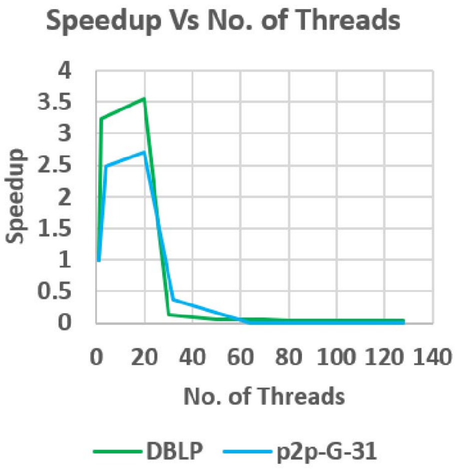
VI. Results

In our experimentation, network DBLP is the largest, so we present our result for this network in most of the cases. The experiments on other networks are omitted for brevity. Many of them have similar characteristic and findings.

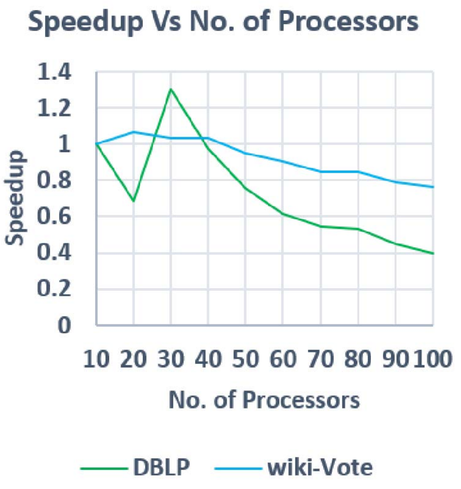
Figure 1 represents the speedup of the parallel algorithms. Figure 1b shows poor speedup, whereas figure 1a shows a relatively better speedup. For figure 1a, we get a speedup of around 4. But the number of physical processing core being 20, we cannot leverage the productivity of shared-memory based algorithm. Speedup increases as long as the maximum number of physical cores is not reached. After this certain point, speedup downgrades drastically. For figure 1b, we get a speedup of around 1.5 for 30 processors. The speedup is not high but can be increased if the communication overhead of MPI can be overcome on which we are working further. So, we analyze the results of MPI-based algorithm to find out the bottlenecks to make the algorithm more efficient in our future work.

Figure 2 delineates the time taken at every communication among the processors described in section IV-B2a, IV-B2c, and IV-B2d. Communication time required in Section IV-B2a, is independent of the number of processors. Major portion of total time is spent in communication in Section IV-B2d, as the time increases linearly. With increasing number of processors, runtime increases. The reason is the communication overhead among processors. But for smaller number of processors, runtime is high too because of dealing with higher number of vertices at each processor. After a certain number of processors, runtime starts increasing consistently. From Figure 2, we can see that this optimum number of processor is in between 30-35.

Next, we vary network size keeping number of processor constant (100 processors) to find out the relation between network size and runtime shown in Figure 3. Communication time required in Section IV-B2a, does not depend on the size of network too. It does not follow any trend with



(a) Shared Memory



(b) Distributed Memory

Figure 1: Speedup factor of parallel Louvain for different networks

increasing network size. So, we can conclude that the initial communication time depends on the nature of the graph rather than number of processor or network size. It depends on the distribution of vertices in a graph. Again, time required in section IV-B2c and IV-B2d, both are increasing with network size. As a result, total runtime increases with rising network size.

We have also figured out how the number of community changes at each level of iteration. We label our first level as $L0$ and subsequent ones as $L1, L2, \ldots$ For DBLP network, from Figure 4a we find that in first level, $L0$, number of community decreases more quickly with lower ranked

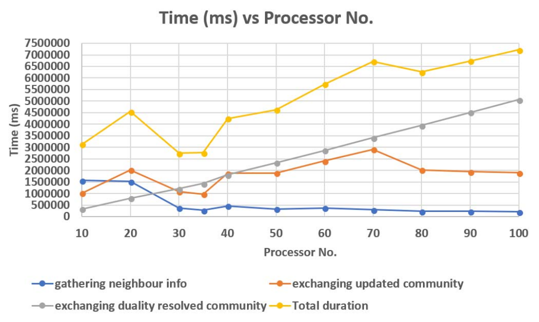


Figure 2: Runtime analysis of DBLP network for varied number of processors

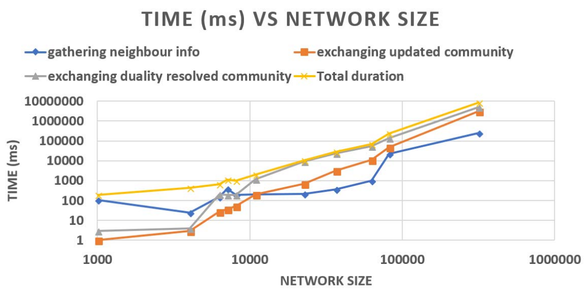
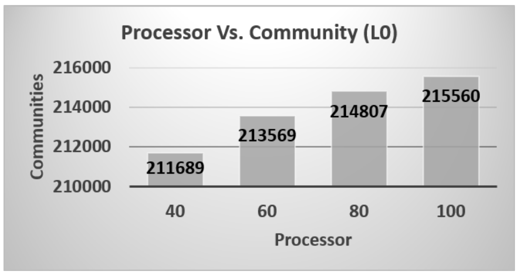
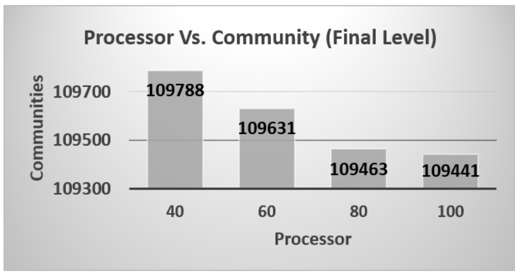


Figure 3: Runtime analysis for varied network size



(a) Initial Level



(b) Final Level

Figure 4: Number of communities at different level of iteration

processors. The reason might be that with higher number of processors, the vertices are scattered more than that of lower number of processors. This tends to accumulating vertices under communities much easier for lower ranked processors. But Figure 4a shows that fewer communities are achieved at final level with higher ranked processors. So, higher ranked processors converge slow but lead to more accuracy.

Table IV: Percentage of Community Size for DBLP Network

Community Size	No. of Community		Percentage (%)	
	Seq.	Par.	Seq.	Par.
1	108877	108877	99.80	99.17
2−10	46	266	0.042	0.242
11−100	40	530	0.037	.483
101−1000	51	88	0.0467	0.080
1001−10000	82	25	0.075	0.023
10001−22000	3	2	0.0027	0.002

Another finding is that very large number of smaller communities contribute to the growing number of communities at the final level of computation. From Table IV we can see that 99.80% communities are single-noded community for sequential [2] implementation and for our implementation the percentage is 99.17%.

VII. Performance Analysis

A. Comparison with Sequential Algorithm

Table V: Deviation of number of Communities

Network	Number of Communities		Deviation (%)
	Sequential	Parallel	
wiki-Vote	1213	1216	0.042164441
com-DBLP	109104	109441	0.106282326

We have compared our algorithm with the sequential version [2] to analyze the accuracy of our implementation. Deviation of community between sequential and our implementation is represented in Table V. The deviation is negligible compared to network size. Figure 5 shows the comparison between sequential and parallel version of DBLP network at different levels of iteration. For both version, the number of community is within the range 109000-109500.

B. Comparison with Other Parallel Algorithms

Most parallel algorithms for Louvain method are implemented using shared-memory approach. The work in [4] is a distributed-memory oriented parallel Louvain algorithm. They have parallelized their algorithm only on the first level of Louvain algorithm and calculated later levels in sequential manner. For larger networks, after reduction at first level, the network size might not considerably decrease. In this case, later levels also require parallelization. Our approach is unique in the way that we have parallelized all levels of the sequential Louvain algorithm compared to theirs. They also get speedup for upto 16 processors that we can increase upto 30-35 processors.

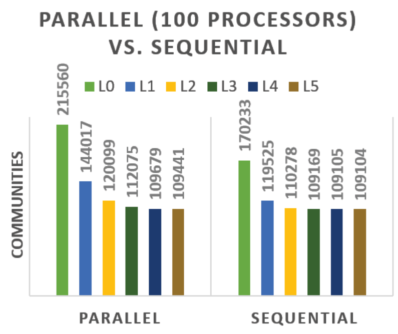


Figure 5: Community Comparison (Parallel Vs Sequential)

VIII. Conclusion and Future Work

Although distributed-memory parallel algorithms have many advantages over shared-memory based algorithms, the challenges to implement an efficient MPI based parallel Louvain algorithm are cumbersome. Our shared-memory based implementaiton achieves up to 4-fold speedup, which is limitated by the number of physical cores available to our system. For distributed-memory algorithms, communication overhead is introduced while exchanging local information after each major computational task. Due to such overhead, the power of parallelism cannot be fully utilized. In this paper, we have identified the impediments in implementing distributed-memory parallel algorithms for Louvain method. Our distributed-memory parallel Louvain algorithm scales up to 30 to 35 processors, which is larger than that of the existing only distributed memory algorithm. However, the number is still not large and we will work in future to improve the scalability of our algorithms by further reducing the communication overhead. An efficient load balancing scheme is also desired to make the algorithm more scalable. Further, in our future work, we paln to eliminate the effect of small communities that hinder the detection of meaningful medium sized communites. We will also investigate the effect of node ordering (e.g., degree based ordering, k-cores and clustering coefficients) on the performance of the parallel Louvain alogorithm.

Acknowledgments

This work has been partially supported by Louisiana Board of Regents RCS Grant LEQSF(2017-20)-RD-A-25 and University of New Orleans ORSP Award CON000000002410.

References

[1] J. McCalpin, ”Memory bandwidth and machine balance in current high performance computers”, IEEE Technical Committee on Computer Architecture Newsletter, 1995, pp. 19-25.
[2] V. Blondel, J. 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.
[3] S. Bhowmick and S. Srinivasan, A Template for Parallelizing the Louvain Method for Modularity Maximization, in Dynamics On and Of Complex Networks, Volume 2, Springer New York, 2013, pp. 111124.
[4] C. Wickramaarachchi, M. Frincuy, P. Small and V. Prasannay, ”Fast Parallel Algorithm For Unfolding Of Communities In Large Graphs”, in High Performance Extreme Computing Conference (HPEC), 2014.
[5] M. Fazlali, E. Moradi and H. Tabatabaee Malazi, ”Adaptive parallel Louvain community detection on a multicore platform”, Microprocessors and Microsystems, vol. 54, pp. 26-34, 2017.
[6] C. L. Staudt and H. Meyerhenke, Engineering Parallel Algorithms for Community Detection in Massive Networks, IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 1, pp. 171184, Jan. 2016.
[7] Cray Documentation Portal”, Pubs.cray.com. [Online]. Available: https://pubs.cray.com/content/S-3014/3.0.UP00/cray-graph-engine-user-guide/community-detection-parallel-louvain-method-plm.
[8] E. Moradi, M. Fazlali, and H. T. Malazi, Fast parallel community detection algorithm based on modularity, in 2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS), 2015.
[9] ”Documentation — User Guides — QB2”, Hpc.lsu.edu. [Online]. Available: http://www.hpc.lsu.edu/docs/guides.php?system=QB2.
[10] ”Stanford Large Network Dataset Collection”, Snap.stanford.edu. [Online]. Available: https://snap.stanford.edu/data/index.html.
[11] S. Arifuzzaman and M. Khan, “Fast parallel conversion of edge list to adjacency list for large-scale graphs,” in *23rd High Performance Computing Symposium*, 2015.
[12] S. Arifuzzaman, M. Khan, and M. Marathe, “A Space-efficient Parallel Algorithm for Counting Exact Triangles in Massive Networks,” in *17th IEEE International Conference on High Performance Computing and Communications*, 2015.
[13] S. Arifuzzaman, M. Khan, and M. Marathe, “PATRIC: A parallel algorithm for counting triangles in massive networks,” in *22nd ACM International Conference on Information and Knowledge Management*, 2013.
[14] S. Arifuzzaman, M. Khan, and M. Marathe, “A fast parallel algorithm for counting triangles in graphs using dynamic load balancing,” in *2015 IEEE BigData Conference*, 2015.
[15] S. Fortunato and A. Lancichinetti, “Community detection algorithms: a comparative analysis,” in *4th International ICST Conference on Performance Evaluation Methodologies and Tools*, 2009.
[16] M. Girvan and M. E. J. Newman, “Community structure in social and biological networks,” *PNAS*, vol. 99, no. 12, pp. 7821–7826, June 2002.
[17] H. Kwak, C. Lee, H. Park, and S. Moon, “What is twitter, a social network or a news media?” in *WWW*, 2010.
[18] J. Ugander *et al.*, “The anatomy of the facebook social graph,” *CoRR*, vol. abs/1111.4503, 2011.
[19] A. Clauset, M. E. J. Newman, and C. Moore, “Finding community structure in very large networks,” *Phys. Rev. E*, vol. 70, p. 066111, Dec 2004.
[20] S. K. Usha Nandini Raghavan, Reka Albert, “Near linear time algorithm to detect community structures in large-scale networks,” *CoRR*, vol. abs/0709.2938, 2007.