View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of New Orleans

University of New Orleans

ScholarWorks@UNO

University of New Orleans Theses and Dissertations

Dissertations and Theses

Spring 5-22-2020

# Accelerating the Information-Theoretic Approach of Community Detection Using Distributed and Hybrid Memory Parallel Schemes

Md Abdul Motaleb Faysal
mfaysal@uno.edu

Follow this and additional works at: https://scholarworks.uno.edu/td

## Recommended Citation

Faysal, Md Abdul Motaleb, "Accelerating the Information-Theoretic Approach of Community Detection Using Distributed and Hybrid Memory Parallel Schemes" (2020). *University of New Orleans Theses and Dissertations*. 2739.
https://scholarworks.uno.edu/td/2739

This Thesis-Restricted is protected by copyright and/or related rights. It has been brought to you by ScholarWorks@UNO with permission from the rights-holder(s). You are free to use this Thesis-Restricted in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis-Restricted has been accepted for inclusion in University of New Orleans Theses and Dissertations by an authorized administrator of ScholarWorks@UNO. For more information, please contact scholarworks@uno.edu.

# Accelerating the Information-Theoretic Approach of Community Detection Using Distributed and Hybrid Memory Parallel Schemes

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by

Md Abdul Motaleb Faysal

B.S. Bangladesh University of Engineering and Technology, 2014

May, 2020

This work is dedicated to the memory of my beloved nephew *Ataullah Imad* who left this world unexpectedly on $6^{\text{th}}$ May 2019, from an unknown illness.

ACKNOWLEDGMENTS

I want to acknowledge the support and guidance I received from various sources and individuals in finishing this work with immense gratitude and thankfulness. I cannot imagine myself coming up to this point without those essential catalysts of an efficacious endeavor.

First, with the utmost respect, I like to acknowledge the guidance I received from my supervisor Dr. *Shaikh Arifuzzaman*, assistant professor of Computer Science at the University of New Orleans (UNO) who is my supervisor on my ongoing effort in pursuing Ph.D. as well. I can acknowledge without any reservation that this thesis work would not reach the finish line without his guidance and moral support.

Second, I am grateful to my parents and siblings who believe in me and have been supporting me in my pursuit of higher education. I am thankful to have wonderful nephews and nieces who inspire me constantly and amaze me as they grow up. I am truly thankful to have a family of such beautiful people.

Third, I would like to acknowledge the spontaneous supports from my friends both at home and abroad who not only play the roles of teammates or opponents in the playground but also boost me morally with their words of inspiration and wisdom. Specifically, I want to mention Md Khairul Habib Pulok and Md Kauser Ahmmed here at UNO who were by my side during my struggling time.

Finally, I want to acknowledge the grants *(BoR RCS grant LEQSF(2017-20)-RD-A-25)* and *(ORSP SCORE grant 2019)* for the continuation of this thesis project.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

SYMBOLS

| | |
|---|---|
| $G$ | A graph data structure |
| $V$ | Set of vertices/entities in a graph |
| $E$ | Set of edges in a graph |
| $H$ | Minimum Entropy |
| $P(X)$ | Probability of some event $X$ |
| $Q_n$ | Number of questions |
| $L(M)$ | Minimum Description Length or Codelength for $M$ modules |
| $N$ | Total number of vertices in graph $G$ |
| $\#$ | Number of |
| $Q$ | Modularity Score |
| $\varepsilon$ | Parallel Efficiency |

## ABBREVIATIONS

| | |
|---|---|
| BFS | Breadth First Search |
| CS | Computer Science |
| MCMC | Markov Chain Monte Carlo |
| MDL | Minimum Description Length |
| SBM | Stochastic Block Model |
| LFR | Lancichinetti–Fortunato–Radicchi |
| DBLP | Database systems and Logic Programming |
| LONI | Louisiana Optical Network Infrastructure |
| FDR | Fourteen Data Rate |

# GLOSSARY

| | |
|---|---|
| MPI | Message Passing Interface is a framework for communication among processes in distributed-memory parallel computing. |
| OpenMP | Is a shared memory based parallel computing framework for managing threads and thread-based computation. |
| Network | A term used to represent graph data structure with entities represented as vertices and the relationship between entities represented as edges. |
| NP-hard | A class of problems in computational theory that are not solvable in polynomial time. |
| Conductance | A metric with the concept similar to electric conductivity to measure the quality of the discovered community. |
| Modularity | A metric to capture the natural clustering behavior of the groups of vertices within a graph. |
| Metis | A graph partitioning framework. |
| mpi4py | A python-based MPI framework. |
| DBLP | Is a computer science bibliography website. |
| Infomap | A well known information-theoretic algorithm for community detection. |
| Map equation | A mathematical optimization function of the Infomap algorithm to compress the regularity of the information in a network. |

ABSTRACT

There are several approaches for discovering communities in a network (graph). Despite being approximating in nature, discovering communities based on the laws of *Information Theory* has a proven standard of accuracy. The information-theoretic algorithm known as Infomap developed a decade ago for detecting communities, did not foresee the tremendous growth of social networking, multimedia, and massive information boom. To discover communities in massive networks, we have designed a distributed-memory-parallel Infomap in the MPI framework. Our design reaches scalability of over 500 processes capable of processing networks with millions of edges while maintaining quality comparable to the sequential Infomap. We have further developed a novel parallel hybrid approach for Infomap consists of both distributed and shared memory parallelism using MPI and OpenMP frameworks. This achieves a speedup of more than 11× in processing a network of over 100 million edges which is significantly greater than the state-of-the-art techniques.

**Keywords**- Information-Theory, Distributed-memory, Shared-memory, Hybrid, Big Data, Graph Mining, Parallel Computing, Community Detection

# 1 INTRODUCTION

Finding community structures within a network (graph) has become a fundamental technique in analyzing entities in the social network based on mutual interests and similar background, classifying cells or biological units that perform similar kind of activities in biological networks (e.g. grouping brain cells based on their interconnection and activity to perform a specific operation of the body), detecting internet anomaly (e.g., detecting fraudulent websites), building efficient product recommendation system by clustering customers based on their purchase habits, connecting research community based on collaboration network and so on.

The data structure in computer science and mathematics used to express interactions/relationships among entities is called a graph. A graph $G$ can be expressed as $G : (V, E)$ where $V$ represents the set of entities known as vertices or nodes and $E$ represents interactions among entities known as link or edge. The word network is often used as a synonym for a graph. Networks are a standard representation for expressing complex interactions among multiple objects. Although identifying community has become a prominent way of network analysis, there is no bold definition of the term community in the context of network analysis.

As described by Porter et al. [1], Fortunato et al. [2] and Newman et al. [3] community detection, sometimes called network clustering, is the division of the vertices of an observed network into groups such that connections are dense within groups but sparser between different groups. Based on the type of community membership a vertex of a network can have, there are generally two categories. One is overlapping community membership and another is disjoint community membership. In disjoint

membership, a vertex belongs to only one community at a time. In overlapping community membership, a vertex may belong to one or more communities at a particular time. Figure 1.1 illustrates the two types of community membership.

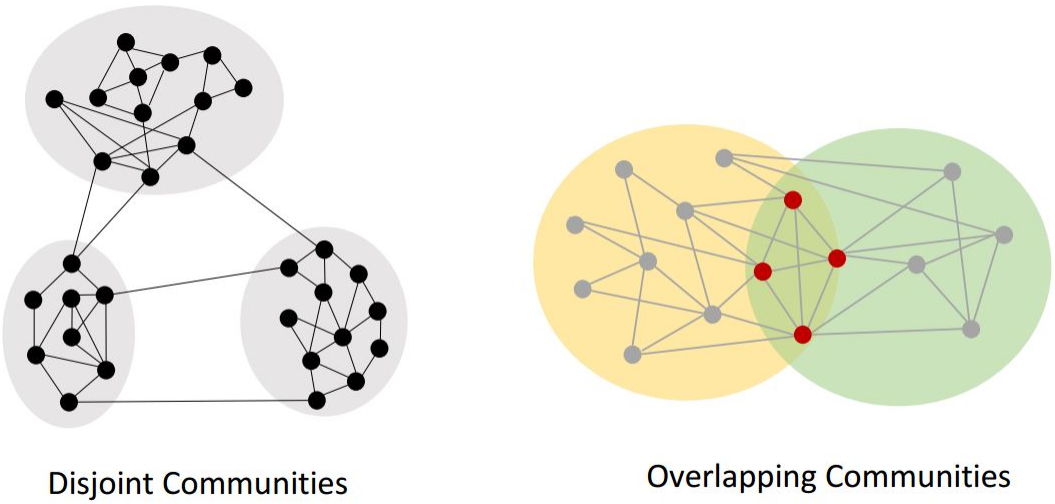



Figure 1.1.: The illustration on the left presents disjoint community (oval shapes represent individual community) where a vertex belongs to one community at a time. The illustration on the right presents overlapping communities where the vertices marked as red belong to multiple communities

Our work focuses on discovering disjoint community membership for the vertices within a network. There exist several algorithms for discovering communities. The community detection algorithms computationally feasible for real-world applications are mainly approximation algorithms as discovering the exact number of communities based on optimization techniques is an NP-hard problem [2, 4]. The approximation algorithms can be categorized based on the methodology being used.

The classification of the community detection methodologies described in the table 1.1 is based on static networks. Arzum et al. [5] provided a brief description of the categories of community detection methodologies. Our approach deals with static

Table 1.1: Classification of the community detection approaches based on methodology

| **Category** | **Methodology** | **Drawbacks** |
|---|---|---|
| Spectral Methods | Based on spectral properties<br>-Eigenvalue & eigenvector | -Computationally inefficient<br>-Unreliable for sparse network |
| Optimization Methods | Optimizing quality metrics<br>-Modularity, Conductance | -Suffers from resolution limit |
| Statistical Inference | Network generative models<br>-Stochastic Block Model | -Accuracy suffers<br>-Computationally expensive |
| Information Theoretic Approach | Uses dynamic process<br>-Random walk, MDL | -Complex logic<br>-Computationally expensive |

networks where the attributes of the networks (e.g., vertices, links) do not change over time. Table 1.1 listed the categories that deal with static networks. This does not mean those methods cannot be used for dynamic networks. There are existing works that incorporate the ability to deal with dynamic networks in optimization methods such as the work by Halappanavar et al. [6], the work of Tiago et al. [7] based on stochastic block modeling (SBM). It is also important to point out, the categories mentioned in table 1.1 have a common challenge of making compromise between speed and accuracy. To maintain high accuracy most of those categories need to compromise speed. For processing big dynamic networks divided into snapshots of different time frames, those categories need a massive amount of time to run

iteratively the same algorithm multiple times on those snapshots.

In section 1.1 we discuss in brief the methodologies of the community detection strategies mentioned in table 1.1.

## 1.1 Descriptions of the Static Community Detection Approaches

In the comparative study of community detection conducted by Fortunato et al. [8], 12 algorithms are described which can be categorized into 4 major groups.

- Spectral methods based on spectral properties of the network. The idea is, if communities are well-identified, the eigenvector components corresponding to vertices in the same community should have similar values. The eigenvalue spectrum of the Laplacian matrix, the adjacency matrix is used to detect communities. A projection of vertices into a metric space by using eigenvectors as coordinates is generated. A limited number of eigenvectors, say $n$ is considered. Each vertex in the network is considered as a geometric point in a Euclidean n-dimensional space where coordinates are the eigenvector components corresponding to that vertex. The points are then grouped using traditional clustering techniques such as K-means clustering. The works by Newman et al. [9, 10], Donetti et al. [11] provide community detection based on spectral methods.

- Optimization methods rely on optimizing a quality function in the process of discovering communities within a network. Modularity is a popular optimization function where the idea is to maximize the difference of structural pattern within an actual network and another network with a random structural pattern. Being an NP-hard problem, approximation algorithms are used with heuristics to optimize this quality function. Based on how the optimization function is posed, the optimization approaches can work as either divisive or agglomerative nature also known as the top-down approach or the bottom-up approach respec-

tively. Among the 12 different approaches for community detection studied in the work by Fortunato et al. [8], 5 of those use the modularity maximization approach in one form or another to detect community. The very first approach in this domain is modularity maximization based on edge betweenness described in the work of Girvan and Newman [12, 13]. Another very popular algorithm that works on modularity maximization is the work from Blondel et al. [14] also known as the Louvain method. Modularity optimization-based techniques suffer from a problem known as *resolution limit* as mentioned by Fortunato et al. [15] where the main idea is if there is a very small community beside a large community then the small community is often overlooked by the algorithm and considered as a part of the large community.

- Community detection based on statistical inference is another category of community detection. Stochastic block modeling is one such approach as described in the works of Tiago et al. [7, 16, 17] and Newman et al. [18]. This strategy comes from the idea that a network can be represented by a generative model where the model parameters determine the properties of the network. In a real-world network, it is not possible to find the exact parameters that generated that particular network. However, statistical inference can be made to determine the parameters. Some statistical processes such as Markov Chain Monte Carlo (MCMC) or Bayesian Inference can be made to determine the partitioning of the network.

- Detecting community based on the information-theoretic approach considers the dynamics of a random walk to reveal the community structure within the network. The core of this kind of strategy lies in Information-Theory and statistics. The concept like minimum entropy theorem is used to compress the data generated by the dynamic process on the basis that high regularity of the information also means a more compressible form of data. Also, the concept of

statistics such as Minimum Description Length (MDL) is utilized to represent the overall quality of the compressed information within the whole network. The higher the structural pattern (community) is in a network, the more chance to compress that information and in the process of compressing the information, communities are revealed. The work by Rosvall et al. [19] provides a method of discovering communities by information-theoretic approach.

Among all the four categories mentioned above, the modularity optimization method or to be more specific the Louvain method is more popular than others because of its' easily comprehensible nature. However, as mentioned earlier, modularity maximization strategy is not only NP-hard, it has also the *resolution limit* problem that may affect the accuracy of the detected communities in a network. The study conducted by Fortunato et al. [8] reveals the information-theoretic algorithm of community detection by Rosvall et al. [19] to be the highest accuracy in the LFR [20, 21] benchmark. There is an MDL based quality function which is named as the *Map equation* by the authors in the study [19]. Fortunato et al. [8] named that algorithm as *Infomap*.

## 1.2 Motivation for Parallel Algorithm in Discovering Community

The different community detection approaches described in section 1.1 have one thing in common. They are all sequential in nature. The algorithms were devised at the timeline when the size of the networks would hardly reach a million vertices. Because of the tremendous growth of social networks and multimedia capturing devices, inexpensive storage, networks are now reaching the size of billions of vertices and billions of edges. The sample networks used by Fortunato et al. [8] for the LFR benchmark had thousands of vertices. Execution time performance was not considered when those algorithms were compared. In today's world when comparing algorithms that deal with a massive dataset, the efficiency of the algorithm is also a

key point to consider beside the accuracy. The sequential nature of those methods (section 1.1) heavily affects the computational efficiency. Modern computers come with multiple processing cores with inherent support for parallel computing. Recent state-of-the-art techniques are now trending to devise algorithms that can exploit the benefits of shared-memory parallelism or distributed-memory parallelism. The sequential algorithms are being redesigned to process network data in parallel by using many threads or processing cores.

In this thesis work, we develop a parallel algorithm for the approach called Infomap devised by Rosvall [19]. Several reasons motivated us to devise a parallel algorithm of Infomap. One reason is being approximate in nature, this algorithm is highly accurate. Another reason is although having higher accuracy than the Louvain method as demonstrated here [8] and being free from the *resolution limit* problem present in modularity optimization techniques such as the Louvain method, there is a very little amount of work in devising parallel algorithm of Infomap. There are also state-of-the-art techniques emerging for designing parallel algorithms based on statistical inference such as Stochastic Block Modeling (SBM). We will discuss more of those parallel algorithms in the literature review section. If an efficient and scalable parallel algorithm can be developed for Infomap, it will deliver fast processing of massive networks with a highly accurate result making it an ideal choice for community discovery.

In this work, we devised first a distributed memory parallel Infomap algorithm and showed that it can achieve very high scalability reaching 512 processes. To reach that high scalability we used a few network partitioning and load balancing strategies. One of the strategies was using a simplistic partitioning method where an equal number of vertices were distributed among processes to compute the communities in each iteration. The uneven degree distribution of the vertices in the real-world network guided us to use novel graph partitioning strategy Metis [22] to ensure proper load balancing

across the working processes. To design a distributed memory parallel algorithm, we have come up with a few heuristics to handle the issues of graph processing across the distributed environment. To maintain the accuracy of the detected communities similar to the sequential Infomap we designed a few other problem-specific heuristics. In the process of devising a distributed parallel algorithm, we observed some parts of the algorithm to be more efficient if the communication cost of the distributed algorithm can be evaded while engaging shared memory based parallelism entity such as thread. Based on that observation, we designed a hybrid algorithm that incorporates both distributed and shared memory based parallelism. We were able to process massive networks that take hours to process in less than 15 minutes while maintaining similar accuracy and quality of the detected communities.

To summarize, we made the following contribution

- We designed a Message Passing Interface (MPI) based distributed-memory parallel algorithm for community detection using an information-theoretic approach.
- We used a vertex-based graph partitioning strategy to manage workload across processes. It helped us to attain scalability of up to 256 processes. We then refined our load balancing strategy by adopting Metis [22] graph partitioner that let us use edge-cut based partitioning across the MPI processes. This increased the scalability to 512 processes while ensuring higher execution speedup.
- We come up with a few heuristics to ensure the fast processing of massive networks across distributed platform while maintaining high accuracy similar to sequential Infomap. Those heuristics can be applied for similar kind of graph computation problems in the distributed platform. In a later section, we will discuss more of those heuristics.

- We designed a hybrid algorithm in a combination of distributed memory parallelism (MPI) and shared memory parallelism (OpenMP) to exploit the benefit of both types of parallelism in the appropriate parts of our algorithm. It gave us speedup factors for massive networks higher than any state-of-the-art parallel Infomap techniques to the best of our knowledge.

# 2 LITERATURE REVIEW

Research in designing parallel algorithms to minimize the computation time for different computational problems are getting much attention in recent years. The emergence of supercomputers with millions of processing cores and the tremendous growth of big data due to the advancement of information technology both are playing complementary roles for the development of research in this direction. Research in parallel algorithms [23–30] for graph data analysis is an essential outcome of that. Parallelizing different community detection approaches mentioned in table 1.1 in chapter 1 is no exception.

There exist several sequential algorithms based on modularity optimization [12, 31–35]. The work [31] is a fast implementation of the work by Newman et al. [12]. The work by Guimerá [32] claimed that finding the modularity of a network is analogous to finding the ground-state energy of a spin system and demonstrated that random graphs and scale-free networks can exhibit modularity. The work of Claire et al. [33] used modularity optimization with the combination of Monte Carlo methods with simulated annealing. The work of Andres et al. [34] is also based on the combination of modularity optimization with simulated annealing. The work by Radicchi et al. [35] is in the spirit of the work by Girvan and Newman [12]. This is a divisive hierarchical method based on the edge clustering coefficient unlike edge betweenness in [12]. The work by Blondel et al. [14] is a well-known community detection approach based on modularity maximization using a greedy agglomerative heuristic.

Several parallel implementations exist for the modularity based approach of the Louvain method. An OpenMP implementation is given by Bhowmick et al. [36].

Hiroaki et al. [37] demonstrated a fast modularity based community detection by avoiding searching all the vertices in each iteration. Zhang et al. [38] demonstrated a distributed framework that speeds up the convergence rate by considering the most suitable candidate vertices to be processed in each iteration. GPU based parallel Louvain is presented in the study of Cheong et al. [39] and Naim et al. [40]. A combination of the Louvain algorithm and the breadth-first search (BFS) is used by Staudt et al. [41, 42] for distributed-memory parallelization. Zeng et al. [43, 44] designed parallel Louvain that can achieve high scalability over thousands of CPU cores. More recent work is emerging on parallel implementation of the Louvain algorithm such as Sattar et al. [45]. Sayan et al. [46] demonstrated a distributed+shared memory (MPI + OpenMP) based work on the Louvain algorithm.

The study by Guimera et al. [32] showed that a random network with irregular community structure can still display high modularity value. As a result, during the process of detecting communities relying on modularity may not deliver high-quality clusters and the detected communities may not reflect the actual communities. Another caveat of modularity based approach is it may suffer from the *resolution limit* problem and therefore may struggle at detecting small communities as described by Fortunato et al. [15].

The use of statistical inference and generative models to infer communities in a network is gaining attention in recent years [47–49]. Among those models, the most popular one is the stochastic block model (SBM) [50–52] where the idea itself is not very recent. The idea is to divide the vertices in the network into $B$ blocks and a $B \times B$ matrix specifies the probabilities of edges existing between vertices of each block. The model generalizes the community structure [8] by accommodating assortative connections. In this context, the task of detecting communities is transformed into a process of statistical inference of the parameters of the generative model given

the observed data. The problem of network partitioning using a statistical inference model is discussed in the studies of Tiago et al. [7, 16, 17]. His work on the stochastic block model for partitioning (the term community detection is more often called as partitioning in the context of SBM) incorporates the degree corrected model by Karrer et al. [53] for large scale dynamic network. The algorithm is of sub-quadratic complexity $O(Nln^2N)$ for a sparse graph where $N$ is the number of vertices with $N \approx E$. The model provided by Peixoto [7] can either function as a greedy heuristic when partitioning in the block-level or Markov Chain Monte Carlo (MCMC) method when sampling individual vertex. Peixoto provided an OpenMP based implementation [54]. Another OpenMP based work with the modified heuristic for fast network processing has emerged [55]. Distributed parallelization techniques [56, 57] on SBM in python and *mpi4py* have emerged. The raw performance speedup of parallelization in native code is difficult to achieve while using a scripting language such as python. The major limitation of python is being computationally slower than C or C++. This has been pointed out by comparing 3 different versions of the baseline algorithm in the study of streaming graph challenge [58].

As demonstrated by Lancichinetti et al. [59] empirically, Infomap is one of the finest algorithms in discovering high-quality communities. Later this fact is corroborated by a more detailed comparative analysis from Aldecoa et al. [60]. The original Infomap algorithm which is sequential in nature is developed by Rosvall et al. [19] in 2008. Compared to parallelizing the modularity based community detection algorithm, there are very few works in parallelizing Infomap. There is a shared memory based parallel execution model of Infomap proposed by Bae et al. [61]. While achieving high-quality communities similar to the sequential Infomap, there are limitations in shared memory based implementation. The scalability of shared memory based implementation is limited by the number of physical cores and memory in a single machine. An asynchronous distributed memory-based implementation using the

GraphLab framework [62] was introduced by Bae et al. [63]. This distributed implementation demonstrated the scalability of up to 128 processing cores. In recent years, the work of Zeng et al. [64] has shown scalability for thousands number of processors. However, the obtained speedup given the huge number of processors they used is relatively very low. In their work, they did not provide the quality analysis of their implementation compared to the sequential Infomap except for some small networks (e.g. DBLP, Amazon). It is equally important to achieve high scalability of distributed community detection as well as maintaining high quality. The high quality of the detected communities is the reason that makes Infomap standing out over other approaches for discovering communities [59, 60].

To discover high-quality community and to process massive networks fast, we have implemented an MPI based distributed information-theoretic community detection algorithm [65] based on the work of Rosvall et al. [19]. Later we extend our previous work [65] by combining together the MPI based distributed-memory parallelism and the OpenMP based shared-memory parallelism and achieved a speedup higher than the state-of-the-art techniques available. To the best of our knowledge, this is the only work available of this kind that utilizes the benefits of both shared-memory and distributed-memory based parallelism.

# 3 PROBLEM SPECIFICATION

## 3.1 How Infomap Works

Infomap uses a standard data compression technique on a dynamic process (random walk). Infomap exploits the duality between compressing a data set and extracting significant patterns or structures in that data set. This duality is discussed in a branch of statistics named MDL or Minimum Description Length statistics [66, 67]. The data we are interested in this context is the trace of the flow on the network. The trace of the flow can be represented as some binary codeword. If an optimal code can be found for describing places traced by a path on a network, it also solves the duality problem of finding the structural features of that network. Therefore, Infomap looks for a way to assign codewords to vertices that is efficient considering the dynamics on the network. This takes us to the heart of information theory, and we can employ Shanon's source coding theorems or Shanon's minimum entropy theorem [68] to find the limits on how far we can compress the information.

Shanon's minimum entropy theorem can be mathematically expressed in the following ways

$$H = \sum_{i=1}^{n} p_i \times \log_2(X) \tag{3.1}$$

or,

$$H = \sum_{i=1}^{n} p_i \times \log_2(1/p_i) \tag{3.2}$$

or,

$$H = -\sum_{i=1}^{n} p_i \times \log_2(p_i) \tag{3.3}$$

To understand how Shanon's minimum entropy works and therefore can be used to get the optimal compression of the information, we are going to use an example. Let's say, we have 2 machines generating information in the form of events. Machine 1 generates 4 events A, B, C, D with the following probabilities

$$P(A) = 0.25$$

$$P(B) = 0.25$$

$$P(C) = 0.25$$

$$P(D) = 0.25$$

Machine 2 on the other hand, generates the above 4 different events with the following probabilities

$$P(A) = 0.50$$

$$P(B) = 0.125$$

$$P(C) = 0.125$$

$$P(D) = 0.25$$

To explain, between the 2 machines which one is producing more information, we can pose the problem in the form of a decision-tree as illustrated in figure 3.1. From this illustration, we can determine which machine is producing more information and which machine is producing less. Let's say both of the machines generated 100 events each. We want to know how many questions we need to ask to guess all the 100 events correctly. In equation 3.1 we have a term $X$. If we express it in terms of probability, the number of possible outcomes of an event is equal to the inverse of the probability of that event, i.e., $X = 1/p$. This is how we get to the equation 3.2 from equation 3.1. From equation 3.2, for machine 1, the average number of questions $Q_n$ we need to ask to determine a particular event is

$$Q_n = p_A \times \log_2(1/p_A) + p_B \times \log_2(1/p_B) + p_C \times \log_2(1/p_C) + p_D \times \log_2(1/p_D)$$

*or*,

$$Q_n = 0.25 \times 2 + 0.25 \times 2 + 0.25 \times 2 + 0.25 \times 2$$

*or*,

$$Q_n = 2$$

For machine 2, the average number of questions we need to ask to determine what is the exact event that occurs

$$Q_n = p_A \times \log_2(1/p_A) + p_B \times \log_2(1/p_B) + p_C \times \log_2(1/p_C) + p_D \times \log_2(1/p_D)$$

*or*,

$$Q_n = 0.5 \times 1 + 0.125 \times 3 + 0.125 \times 3 + 0.25 \times 2$$

*or*,

$$Q_n = 1.75$$

Since both of the machines generate 100 events each, for machine 1 we need to ask 200 questions to determine the outcomes of the 100 events and for machine 2 we need to ask 175 questions to determine the outcomes of the 100 events. From this example, it is clear that machine 1 is producing more information than machine 2. The reason for machine 2 producing less information is based on the regularity of the information machine 2 produces. Based on the probability of event A for machine 2, it is more likely for the machine 2 to be generating event A more than other events. To put it in another way, event A is more regular than other events in machine 2. As a result, the information produced by machine 2 is compressed on average to 1.75 questions than 2 questions in machine 1. That is the beauty in Shanon's minimum entropy theorem. From this explanation of the minimum entropy theorem, it can be understood that the regularity of the information can be exploited to compress that information. We can get a theoretical limit on how much the information can be compressed without physically encoding the information and then compressing that code.

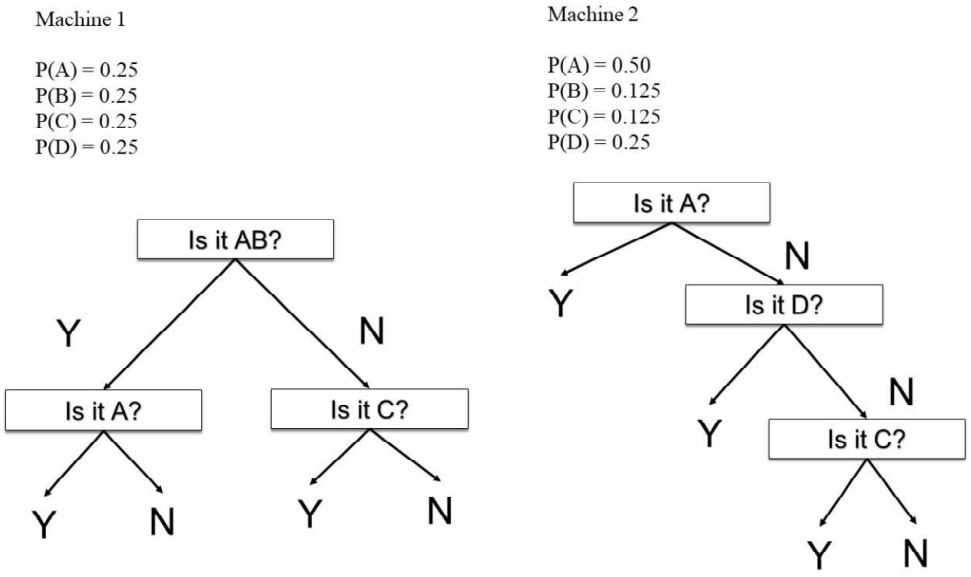



Figure 3.1.: The information generated by machine 1 (left) and machine 2 (right). The information itself is how many questions on average we need to ask to correctly guess the exact event generated by each machine.

As we were discussing Infomap looks for an efficient codewords, a straightforward way to assign codewords to vertices is to use Huffman coding which gives shorter codewords to common events and long codewords to rare ones. The codewords for all the vertices form a codebook. In this codebook, each Huffman codeword specifies a particular vertex, and the codeword lengths are derived from the ergodic node visit frequencies of a random walk. The average node visit frequencies of an infinite-length random walk can be calculated by Google's PageRank algorithm [69].

Let's say there is a significant structural pattern in a network and our goal is to discover the structural patterns (communities). Also, let's name the structural pattern in the network as modules. A random walker moving in the network can be expressed by two different types of moves. One, a random walker moving inside a

structural pattern traversing from one vertex to another. Second, the random walker moving across different modules. An intuitive way of understanding how Infomap works based on the random walk is similar to the traffic within a city and between cities. Traffic within a city stays longer in the city and travels rarely across cities. The target of finding a city (structure/community within a network) is to determine the region within which traffic (random walk) delivers maximal flow. Maximizing the flow within a cluster and minimizing flow among clusters ensures correctness and the quality of detected communities. Based on this concept, the codebook of the vertices can be divided into two parts. The codewords that represent the moves across the modules can be named as the index codebook. The codewords that represent the successive moves inside a module can be named as the module codebook. The codeword lengths in the index codebook are derived from the relative rates at which a random walker enters each module, while the codeword lengths for each module codebook are derived from the relative rates at which a random walker visits each node in the module or exists the module. Using multiple codebooks, the problem of minimizing the description length of places traced by a path is transformed into the problem of how we should best partition the network concerning the flow dynamics. The Huffman coding process is described to make it clear how the coding structure works. But of course, the aim of community detection is not to encode a particular path through the network. In community detection, the goal is to simply find the modular structure of the network concerning flow and to exploit the inference-compression duality to do so. It is not needed to devise an optimal code for a given partition to estimate how efficient that optimal code would be. The detection of the optimal community structure of a network becomes the problem of computing the theoretical limit for different partitions and greedily choosing the one that gives the shortest code length. The optimization function that lets us compute that theoretical limit is called the *Map Equation*.

### 3.1.1 The Map Equation

The optimization function that represents the code length is called the *Map equation*. The target of the optimization is to minimize the code length over all possible assignments of vertices into communities. The *Map equation* is standing on the concept of MDL (Minimum Description Length) which states [67] that any regularity of information can be used for compressing that information. Eq. 3.4 is the given form of the *Map equation* by Rosvall et al. [19].

$$L(M) = q_{\curvearrowright} H(Q) + \sum_{m \in M} p^m_{\circlearrowleft} H(\rho^m) \tag{3.4}$$

In this equation, there are two parts on the right side. The first part is $q_{\curvearrowright} H(Q)$ which can be further divided into two terms where the first term $q_{\curvearrowright}$ represents the sum of exit probability of the random walk for each module in the network. The term $H(Q)$ represents the average codelength of the movements between the modules where $Q$ stands for the probability distribution of the module entering rate. The average codelength of the movements between the module is called index codelength. The second part of the right side of the *Map equation* is $\sum_{m \in M} p^m_{\circlearrowleft} H(\rho^m)$ where the term $p^m_{\circlearrowleft}$ stands for the stay probability of the random walk within module $m$. The parameter $p^m_{\circlearrowleft}$ can be calculated by summing the visit probability of the random walk and the exit probability of the random walk for that module. The term $H(\rho^m)$ is the average code length of the random walk within the module which is named as module code length. The term $\rho^m$ is the probability distribution of the code of module $m$. A more detailed form of the *Map equation* is given in eq. 3.5.

$$\begin{aligned} L(M) = (\sum_{m \in M} q_m) \log (\sum_{m \in M} q_m) - 2 \sum_{m \in M} q_m \log q_m - \sum_{\alpha \in V} p_\alpha \log(p_\alpha) + \\ \sum_{m \in M} (q_m + \sum_{\alpha \in m} p_\alpha) \log(q_m + \sum_{\alpha \in m} p_\alpha) \end{aligned} \tag{3.5}$$

Here the term $q_m$ is the exit probability of module $m$ and is defined by the relative weight of links exiting the module $m$, $\sum_{m \in M} q_m$ is the sum of the relative weight of

links between modules, the term $p_\alpha$ is the visit probability of a vertex $\alpha$ during the random walk, $V$ is the set of all vertices in the network, $p_m$ is the visit probability of a module $m$ calculated by $\sum_{\alpha \in m} p_\alpha$. Interested readers are encouraged to read the appendix section of the original work of Infomap [19] to learn more about the *Map equation*.

## 3.2 Sequential Infomap Algorithm

In the sequential Infomap algorithm 1, line $6-9$ compute the initial visit rate for each vertex using power iteration method in a similar fashion of the PageRank [69] approach, the total number of modules or communities at the very beginning is set equal to the total number of vertices (line 10) and the exit probability for each module is calculated (line 12). Line 13 computes the code length following equation 3.4. Line $15-23$ do the greedy optimization part of the *Map equation* which include finding the best community for a randomly chosen candidate vertex and updating the new code length $L$ (line $17-21$) followed by converting the newfound communities into some super nodes having possibly more than one vertices in an iteration (line 22) and also updating the total number of communities ($M$) in this process. As long as we will get a change in code length ($L$) where it is smaller than the previous iteration code length ($L_{old}$) by more than some threshold value $\tau$, the algorithm will continue execution. The algorithm stops when convergence achieved for the value of the code length in consecutive iterations. The output (line 24) of Algorithm 1 is the total number of communities after convergence ($M$) which is usually less than the total number of vertices in the network.

**Algorithm 1** Sequential Infomap

**Require:** A graph $G(V, E)$, $V$ total vertices, $E$ total edges, $N \leftarrow |V|$

**Ensure:** $M : M \leqslant N$, M is the total number of communities, $M \ll N$

$m_i$, i$^{\text{th}}$ module

$q_{m_i}$, exit probability of module $m_i$

$\tau$, minimum threshold for codelength improvement

$L_{old}$, codelength of previous iteration

$L$, codelength of current iteration

**for** $i = 1$ to $N$ **do**

    calculate initial vertex visit rate $p_{v_i} \leftarrow 1/N$

    compute vertex visit rate $p_{v_i}$ by power iteration

**end for**

declare initial total module $M \leftarrow \{m_i = \{v_i\} | v_i \in V\}$

**for** $m_i = 1$ to $M$ **do**

    calculate exit probability $q_{m_i}$

**end for**

calculate initial codelength $L \leftarrow L(M)$

**do**

    $L_{old} \leftarrow L$

    **for** $i = 1$ to $N$ **do**

        pick randomly a vertex $v_i$

        $m_{new} \leftarrow findNewBestModule(v_i)$

        calculate $L$

    **end for**

    update $M \leftarrow convertModulestoSuperNode()$

**while** $(L_{old} - L) > \tau$

**return** $M$

# 4 SOLUTION STRATEGY: DISTRIBUTED INFOMAP, RESEARCH CHALLENGES

We present the overview of our distributed-memory parallel algorithm in Algorithm 2. Our algorithm consists of two major parts. One part is distributing a partial graph to each MPI process and working on that partial graph inside that process in parallel across all the processes. Another part of our parallel approach is synchronizing the results of community membership for each partial graph distributed across all the processes. Synchronization phase includes the operation of merging the partial graphs distributed across processes in a manner so that uniform community membership is maintained for all of the vertices in $G(V, E)$ across all of the MPI processes.

The synchronization phase is essential for our approach since the assignment of vertices to processors may change in different iterations and each processor may work with a different set of vertices in each iteration. Therefore, each processor must update community information about the vertices it is going to work with before starting execution of the next iteration. All the processors participate in both computing communities and synchronizing updates.

Line $7 - 15$ of the algorithm 2 are similar to the sequential algorithm 1. How we decide to divide the workload across the processes highly influences the scalability of our distributed algorithm. From the parallel computing perspective, we know if each of the processes deals with an equal amount of workload then it is very likely that all of the processes will reach the finish line of the computation in similar time contributing to the overall scale-up of our distributed computing. Initially, we chose to divide the workload in the form of partial graphs among the processes using a naive approach

by adopting the metric of an equal number of vertices in each process. While this resulted in better execution time compared to the sequential algorithm but there was a serious issue of performance bottleneck in some processes. From that observation, we have gone with a more sophisticated graph partitioner Metis [70] from Karypis lab where the partial graphs are computed based on the edge-cut metric. The graph partitions returned from the Metis partitioner have better workload distribution in terms of the number of vertices and edges. However, we are aware of the fact that graph partitioning itself is a time-consuming process, and relying heavily on Metis partitioner in each iteration will drastically increase the overall execution time of our algorithm. Therefore, we have come up with a combination of sophisticated partitioning and naive partitioning where the very first graph distribution is done based on the outcome from Metis partitioner (performed offline) and subsequent workload distributions follow the naive vertex-based distribution approach. We report the experimental outcome before and after using Metis partitioner in Chapter 5.

Each process starts the execution of finding communities (line $16 - 25$) with essentially the same value of $L$ and $M$. Line 18 highlights the use of Metis partitioner for the very first iteration of our algorithm which subsequently is replaced by the naive equal number of vertex distribution approach. Each process does the same operation of randomly choosing vertices, assigning them community membership if applicable and subsequent computation of codelength ($L$) on their own set of vertices in an iteration (line $19 - 21$). However, the bigger challenges of maintaining uniform parameters ($L$, $M$, etc.) come during the supernode creation and the preparation of next iteration phases (line $23 - 24$). We present those challenges in section 4.1 and how we tackle them in section 4.2.

## 4.1 Research Challenges

In this section, we describe the challenges we faced while solving the problem of processing a network partially across distributed processes.

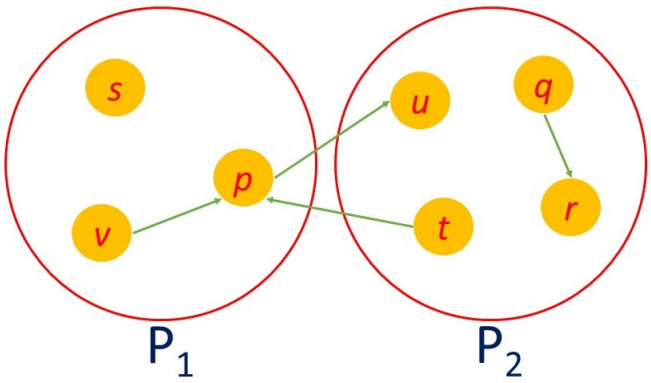



Figure 4.1.: Assignment of modules to vertices in two distributed processes

One major challenge when performing the module assignment for the individual vertices is maintaining uniformity of community assignment during the parameters synchronization stage mentioned in Algorithm 2. This happens because each process updates the communities of each vertex in different orders based on the arrived updated information from other processes. To illustrate this, consider the simple scenario of 7 vertices p, q, r, s, t, u, v divided into two processes P1 and P2. Now based on some processing order of the above vertices, following are the moves of the vertices from own module to another module (N.B. by the term *moves*, here we mean the community assignment decision that results from the maximum code length reduction in each case for the above individual vertex). Let's assume that based on the processing order, following are the moves

$p \longrightarrow u$, $t \longrightarrow p$, $v \longrightarrow p$, $q \longrightarrow r$. Here the symbol $p \longrightarrow u$ means vertex p is moving to the community of vertex u. When we refer to the current community $C$ assignment of a vertex $u$, we represent it by the symbol $C_u$. If the above moves are

performed in the same order as exactly mentioned, following will be the community assignment resulted from the execution of those moves in the sequential algorithm.

$$C_p \leftarrow C_u,\ C_t \leftarrow C_p(= C_u),\ C_v \leftarrow C_p(= C_u),\ C_q \leftarrow C_r \tag{4.1}$$

In a distributed system, a possible scenario can be when those 7 vertices are distributed for processing into two processes $P_1$ and $P_2$, where vertices $p$, $v$, $s$ go to process $P_1$ and vertices $q$, $r$, $t$, $u$ go to process $P_2$ as illustrated in figure 4.1.

Here, the 2 big circles represent 2 different processes and the circles inside represent individual vertices with their name. The arrows represent the move between communities with the direction of arrowhead indicates *from* and *to* of the moves. Vertex without an arrow (e.g. $s$) indicates no move has been found for that vertex that results in compression of code length at the current iteration. In the distributed platform, the community assignments in process $P_1$ are $C_p \leftarrow C_u$, $C_v \leftarrow C_p$ *i.e.* $C_u$. The community assignments in process $P_2$ are $C_t \leftarrow C_p$, $C_q \leftarrow C_r$. After this community assignment information is exchanged between these two processes $P_1$ and $P_2$ for synchronization across the processes following things happen.
In process $P_1$

$$C_p \leftarrow C_u,\ C_v \leftarrow C_p(= C_u),\ C_t \leftarrow C_p(= C_u),\ C_q \leftarrow C_r \tag{4.2}$$

In process $P_2$

$$C_t \leftarrow C_p,\ C_q \leftarrow C_r,\ C_p \leftarrow C_u,\ C_v \leftarrow C_p(= C_u) \tag{4.3}$$

If we visualize the resultant communities in two different processes, it looks like figure 4.2. This is not what we want. We want after synchronization, every process will have the same community information.

Another problem we faced after distributing the vertices among processes is the *vertex bouncing* problem. The notion behind this problem is when two vertices having

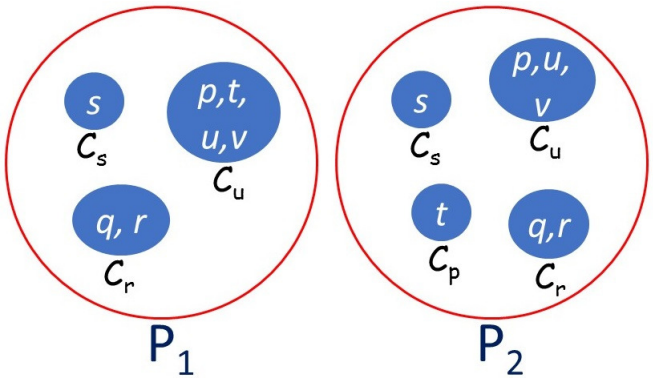



Figure 4.2.: Resultant communities in two different processes

strong affinity are distributed across processes, those vertices make multiple moves that essentially represent one single move. It reduces code length erroneously multiple times. This problem would not have effect if those strongly affined vertices were feed into the same process in an iteration (line 20-25) described in Algorithm 2.

When two vertices $u$ and $v$ are distributed in two different processes. In one process $P_1$ where vertex $u$ is assigned for computation, vertex $u$ will move to the community of vertex $v$. In process $P_2$ where vertex $v$ is assigned, $v$ will move to the community of vertex $u$. These two vertices should have moved only once in a particular iteration. But because of the distribution across processes, it resulted in one extra move and extra reduction of codelength. The problem can be illustrated in figure 4.3.

An important observation is, in the initial few iterations, most of the vertices change their communities. As the algorithm progresses, the number of vertices changing their communities decreases. It is intuitive that after a vertex moves to some community and stays in that communities for a few subsequent iterations, it is likely to stay in that community until the program finishes. That is because most of the vertices find their communities in early iterations. Those vertices become stable in

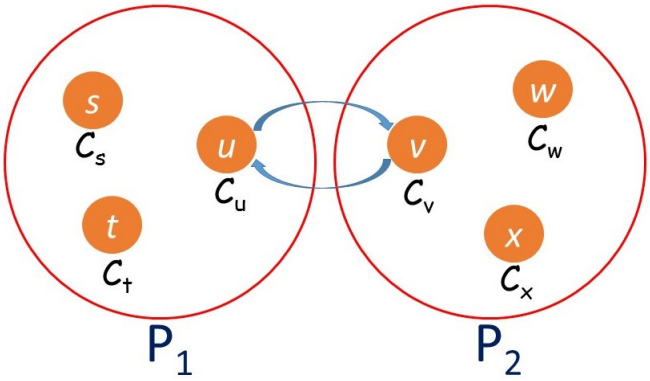



Figure 4.3.: Vertices bouncing between communities

their assigned communities. In later iterations fewer to fewer community updates take place. This observation leads us to the conclusion that in every iteration, considering all of the vertices in the network for new communities as in line 22 of Algorithm 2 incurs redundant activities that waste CPU time and resources. It means we have to reduce the number of vertices that are being considered for community assignments after each iteration. We need to have a measure to distinguish and pick those vertices which are more likely to change their communities in subsequent iterations.

## 4.2 Applied Heuristics

To maintain uniform assignment of the community for each vertex across all of the processes, we have taken the heuristic of priority-based community assignment. In this scheme, the decision of community assignment for a particular vertex is taken by the process which is computing the new community for that vertex. This is a simple yet effective approach for solving the challenge depicted in figure 4.2. So every process will update the community assignment information for those vertices belonging to other processes based on the decision those processes made. A process will not further try to change the community assignment based on the combination of its available information and newly received information from other processes. Figure 4.4 depicts the communities resulting from the same moves of vertices as depicted in

figure 4.1. Now if we have a retrospect of why the non-uniform communities resulted in figure 4.2, that is because process $P_1$ further updated the community information of vertex $t$ to $C_u$ from $C_p$ based on the available information it had along with received information from the process $P_2$. Interestingly, the process $P_1$ didn't have to perform an additional step to get to this outcome. To summarize, the combination of the received information from other processes and the own computational outcome may result in inconsistency in community assignment of a few vertices.

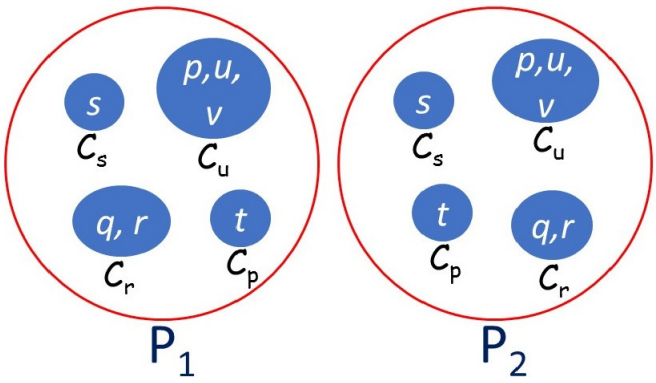



Figure 4.4.: Uniform communities across processes for priority ordering

To prevent recomputing the community assignment for those vertices which are unlikely to move from their current communities, we need to separate those vertices from other vertices that are likely to move in a later iteration. Let's name those vertices as *Inactive Vertices*. On the other hand, those vertices which may move to different communities in some later iteration, let's call those as *Active Vertices*. There is no deterministic way to decide which vertices will be active or inactive at a particular iteration. Rather it is intuitive and empirically observed that those vertices changing their community in an iteration will likely change their community in the next iteration too. Moreover, the neighbors of such vertices may become active due to the active vertices. So we need to have some prediction list of the vertices that may be active before an iteration starts. And that prediction can be made from

the outcome of the community assignment of the previous iteration. The prediction list may contain those vertices which change their communities in the previous iteration, the immediate neighbors of those vertices. It is empirically observed that when a vertex is moved from one community to another, its' immediate neighbors contribute to more than 95% of the quality improvement for subsequent iteration [61].

To counter the vertex bouncing problem, we adopted ordering in assigning vertices to communities to prevent multiple moves of vertices among communities which otherwise should be a single move. Consider the scenario in figure 4.3, to prevent the two moves of $u \rightarrow v$ and $v \rightarrow u$ in process $P_1$ and process $P_2$ respectively, we first check the value of current communities of $u$ and $v$. For instance, those are $C_u$ and $C_v$ with two different community Ids. If the value of the Id for community $C_u$ is less than the value of the Id for community $C_v$ we permit the move of $u$ moving to $C_v$ instantly and do not permit the move otherwise.

**Algorithm 2** Distributed Infomap

**Require:** A graph $G(V, E)$, $V$ total vertices, $E$ total edges, $N \leftarrow |V|$

**Ensure:** $M : M \leqslant N$, M is the total number of communities, $M \ll N$

$m_i$, i$^{\text{th}}$ module

$q_{m_i}$, exit probability of module $m_i$

$\tau$, minimum threshold for codelength improvement

$L_{old}$, codelength of previous iteration

$L$, codelength of current iteration

$P$, number of MPI processes spawned

**for** $i = 1$ to $N$ **do**

    calculate initial vertex visit rate $p_{v_i} \leftarrow 1/N$

    compute vertex visit rate $p_{v_i}$ by power iteration

**end for**

declare initial total module $M \leftarrow \{m_i = \{v_i\} | v_i \in V\}$

**for** $m_i = 1$ to $M$ **do**

    calculate exit probability $q_{m_i}$

**end for**

calculate initial codelength $L \leftarrow L(M)$

**do**

    $L_{old} \leftarrow L$

    **for** $i = 1$ to $(N/P \leftarrow metis)$ in parallel **do**

        pick randomly a vertex $v_i$

        $m_{new} \leftarrow findNewBestModule(v_i)$

        calculate $L$

    **end for**

    update $M \leftarrow convertModulestoSuperNode()$

    synchronize parameters across all processes

**while** $(L_{old} - L) > \tau$

**return** $M$

# 5 EXPERIMENTAL ANALYSIS

## 5.1 Experimental Setup

Most of the experiments and corresponding results we included in this study are executed on the Louisiana Optical Network Infrastructure (LONI) [71] system. The computing cluster we used is QB2 [72]. It is a 1.5 Petaflop peak performance cluster with 504 compute nodes, 20 processing core per node, more than 10000 Intel Xeon processing cores, 2.8 PB Lustre file system. The computing cluster has RedHat Enterprise Linux 6 Operating System, 56 Gb/sec (FDR) InfiniBand, 1 Gb/sec Ethernet management network.

## 5.2 Implementation

We developed our implementation in C++ using the MPI framework with g++ compiler. The source code of our implementation is available online [73]. The program supports the network in pajek (.net) format [74]. The major phase of the algorithm i.e. the greedy optimization phase runs in multiple iterations. In the very first iteration, each vertex represents its' module. In each iteration, every process takes an almost equal chunk of the vertices from the active vertices list. Each process computes the change of MDL for a possible move of that vertex following any of the links that vertex is connected to. It then greedily chooses the move to some module that reduces the MDL most. Each processor prepares an information list of the vertices which have been moved from one module to another. In the synchronization phase, each process then sends that list and updates the community information based on the received information. Each module is also converted to a notion what we call as supernode

which is a group of vertices with the same module id. All the inter edges between a pair of supernodes are converted to a single edge with weight equal to the sum of all edges between that pair of supernodes. After creating a network with supernodes, the greedy optimization of reducing MDL is executed again on the supernode level similar to what was performed on the vertex level previously. After each iteration, the list of active vertices for the next iteration is computed. This process continues until no more reduction in MDL happens in some successive iterations, i.e., it reaches convergence. The final output of the program is the number of detected communities along with the final compressed value of the MDL.

## 5.3 Performance Comparison

We performed a qualitative comparison of our distributed implementation of the Infomap against the one designed by Bae et al. [61]. We performed the parallel performance comparison against the distributed implementations [63] and showed that our work outperforms that implementation in terms of scalability. The implementation of Zeng et al. [64] is not publicly available online. Consequently, we had to rely on the data provided in their paper [64] to reflect on the superiority of our work in terms of speedup gain in later discussion.

## 5.4 Dataset

We used a network dataset of different sizes ranging from the network of 0.31M vertices and 1.04M Edges to the network of $3M$ vertices and $117M$ edges. Table 5.1 gives a brief description of the dataset where columns 2 and 3 show the number of vertices and edges in the network respectively. We have used *Amazon, DBLP, Youtube, Wiki-topcats*, and *soc-Pokec* networks for our experiments with distributed Infomap. The other networks in table 5.1 are significantly bigger and have shown good scalability in our hybrid platform. Therefore, we discuss the experiment results

for those networks in chapter 7. All of the networks in our dataset are collected from SNAP [75]. The reason for us choosing these networks is because of those networks showing good community structures and thus suitable for evaluation and comparison of our implementation.

Table 5.1: Network dataset for our experiments. We used several social and information networks

| **Network** | **# Vertices** | **# Edges** | **Description** |
|---|---|---|---|
| Amazon | 334863 | 925872 | Amazon co-purchased network |
| DBLP | 317080 | 1049866 | CS bibliographical network |
| Youtube | 1134890 | 2987624 | Youtube social network |
| Wiki-topcats | 1791489 | 28511807 | Hyperlinks network from Wikipedia |
| soc-Pokec | 1632803 | 30622564 | Pokec online social network |
| LiveJournal | 3997962 | 34681189 | LiveJournal online social network |
| Orkut | 3072441 | 117185083 | Orkut online social network |

## 5.5 Evaluation

### 5.5.1 Quality analysis of the Detected Modules

Infomap delivers better quality of communities among state-of-the-art techniques as observed by several benchmark-studies [59, 60]. For quality comparison of the detected communities, we used Modularity, Conductance, and convergence MDL value. We compare our result with RelaxMap [61] which discovers communities with quality as good as the original Infomap. In table 5.2 the values of Modularity and Conductance are given for the shared memory based Infomap [61]. Distributed-memory based implementations can achieve quality up to their sequential counterpart at best.

Thus our comparative study with RelaxMap makes it a sufficient comparison in terms of qualitative analysis.

Table 5.2: Modularity and Conductance of the networks for the sequential Infomap

| **Network** | **Modularity** | **Conductance** |
|---|---|---|
| Amazon | 0.77 | 0.23 |
| DBLP | 0.59 | 0.41 |
| Youtube | 0.39 | 0.56 |
| wiki-topcats | 0.43 | 0.57 |
| soc-pokec | 0.52 | 0.47 |
| libJournal | 0.47 | 0.53 |
| com-orkut | 0.42 | 0.54 |

Convergence of the Objective Function

Our objective function of Infomap minimizes the MDL. It is challenging to improve the MDL in distributed implementation in comparison to the sequential or shared-memory based optimization. The outcome of the compression of a previous move may not be available to other processors and the decision of change in MDL may be affected by that which is not the case in sequential/shared implementation. In case of distributed implementation, there is a possibility of premature convergence resulting in an outcome of less improvement of the MDL as also observed by [63]. The outcome we achieved by optimizing the objective function 3.5 is very close to the MDL improvement found in [61]. In table 5.1 we showed the initial MDL of the used networks. In figure 5.1, we have shown the final converged value of the MDL. The difference in MDL is very insignificant in all the cases with the highest difference is for the network *Wiki-topcats* having a final MDL value greater than the MDL value

of [61] by only 0.39. It indicates the detected communities after convergence are similar to that of the [61]. Our algorithm does not suffer from under clustering or over clustering problems.

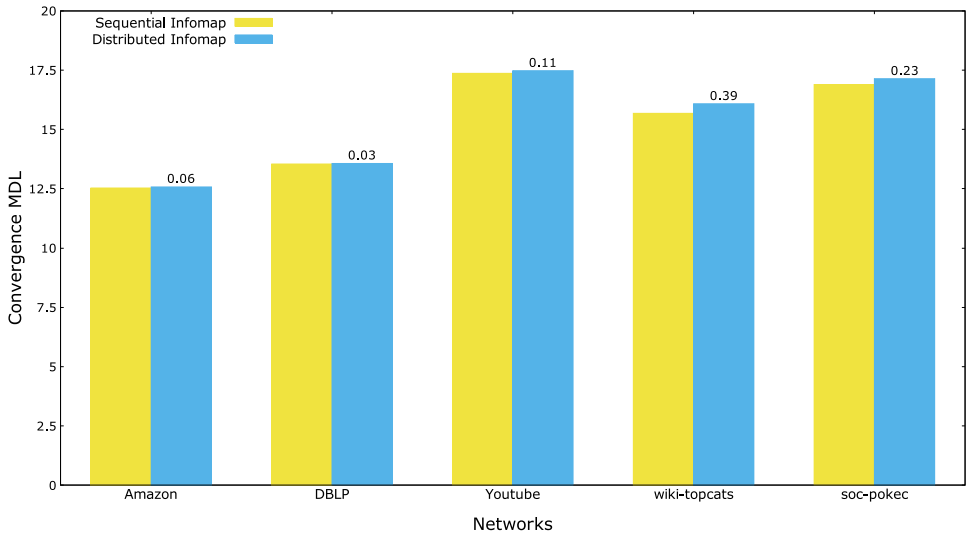


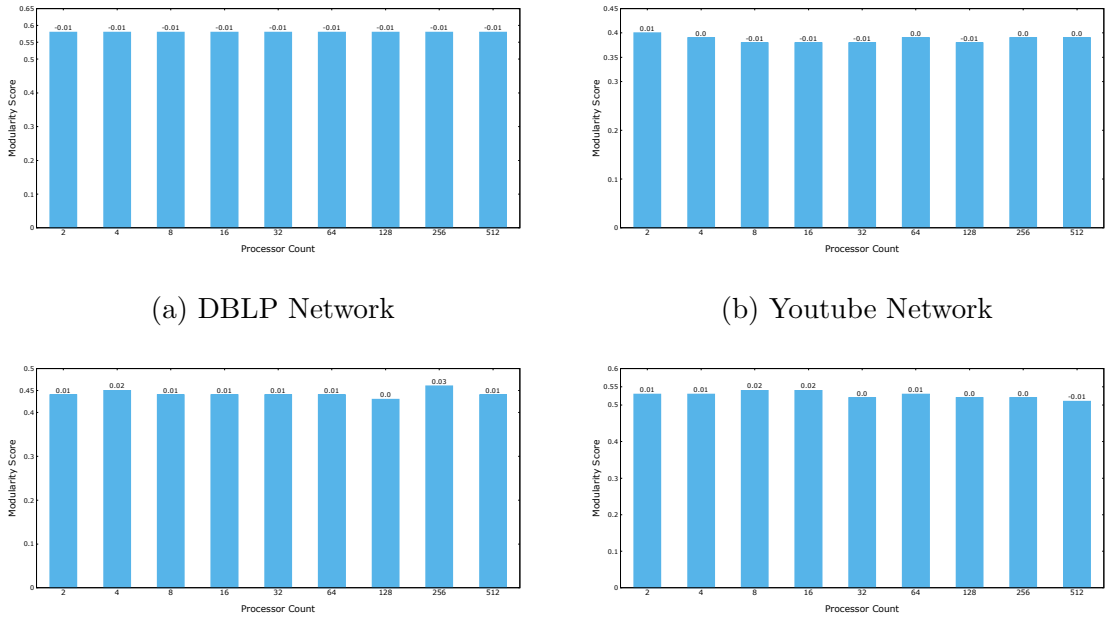

Figure 5.1.: Comparison of MDL after convergence between sequential and distributed Infomap

Modularity

To measure the quality of the detected communities, we used Modularity (Q) measure [13]. This is a measure of how well a network is partitioned into communities. Given a network, Modularity score (Q) of that network means the fraction of edges that fall within the communities minus the expected value of the same quantity if the edges fall at random in a network with the same degree sequence. The mathematical definition of this measure is given in equation 5.1.

$$Q = \sum_{i} (e_{ii} - a_i^2) \tag{5.1}$$

Here, $e_{ii}$ is the fraction of edges that fall within communities, $a_i^2$ is the expected value of the above quantity for the graph with the same degree sequence with random edges. It is a positive decimal value if the number of edges within communities exceeds the expected number. Typically a value of $Q$ in the range $0.3 - 0.7$ means significant community structure [76]. One notable point is, we are not optimizing the Modularity value to detect communities which is the general approach as mentioned in other community detection mechanism [13, 14, 31]. Rather we are using the quality metric (Q) for analyzing the quality of our detected communities we got by optimizing *Map equation* as mentioned in equation 3.4. For the dataset in table 5.1 we have used for our experiments, we listed the quality measurement metrics (i.e. Modularity, Conductance) and their corresponding values from [61]. For measuring the quality of Modularity, we also wanted to see whether the quality fluctuates with the increasing number of processors. We can see from figures 5.2a, 5.2b, 5.2c, 5.2d the values of Modularity vary insignificantly. In the histogram of the above-mentioned figures, we put the difference of values corresponding to the sequential score. For instance, in figure 5.2c and at the bar 256, the value of 0.03 means the Modularity score we obtained for the network *Wiki-topcats* by running it on 256 processing core is greater than by 0.03 from the Modularity value in [61]. One important note is, higher Modularity values signify better communities. We marked all the Modularity histogram plots bar-labels with the difference of Modularity from [61]. The $+ve$ labels indicate by how much the obtained Modularity is greater and the $-ve$ labels indicate by how much the obtained Modularity is lower. To summarize, the Modularity values we obtained for different networks we used are as good as the Modularity value we can find from some sequential or shared-memory based implementation of Infomap. Also, the quality of the detected communities does not vary with the number of partitioning across an increasing number of processors.




(a) DBLP Network

(b) Youtube Network

(c) Wiki-topcats Network

(d) Soc-pokec Network

Figure 5.2.: Change of Modularity values across the different number of MPI processes for (5.2a) *DBLP* network, (5.2b) *Youtube* network, (5.2c) *Wiki-topcats* network, and (5.2d) *soc-Pokec* network. The numeric value on top of each histogram bar of each figure demonstrates the change of Modularity compared to the value of Modularity for sequential algorithm where a positive value indicates higher Modularity and a negative value indicates lower Modularity than the sequential version. The higher the Modularity score, the better the quality of the discovered communities.

### Conductance

According to the study by Yang et al. [77], when the network contains well-separated disjoint communities, Conductance delivers the best quality analysis of the detected communities. For unweighted networks, Conductance measures the fraction of the total number of edges that point outside the community, and for weighted networks, it is the fraction of the total weight of such edges. For directed network,

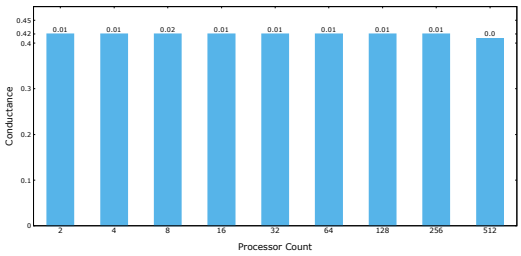
Conductance $\frac{|{E_c}^{out}|}{|{E_c}^{in}|+|{E_c}^{out}|}$ and for the undirected network, Conductance $\frac{|{E_c}^{out}|}{2|{E_c}^{in}|+|{E_c}^{out}|}$. Motivated by the idea of electric conductivity where the higher value of Conductivity means connected paths and 0 or less conductivity means no connection or loosely coupled connection, high Conductance means communities are not well-separated and disjoint, the portions of intra-edges and inter-edges are not well-separated. On the other hand, a low value of Conductance means the communities are well-separated and if not completely but highly disjoint. The smaller the value of Conductance is, the better the quality of the discovered community is.

We use the similar concepts and figures that we used for modularity in 5.5.1 with the only difference is having $-ve$ difference of Conductance from shared/sequential Infomap means a higher quality of the detected communities. From figures 5.3a, 5.3b, we can see the conductance value is insignificantly greater than [61] and for figures 5.3c, 5.3d the Conductance values are insignificantly lower. Therefore we can conclude that the quality of the detected communities of our distributed Infomap is as good as the sequential Infomap.

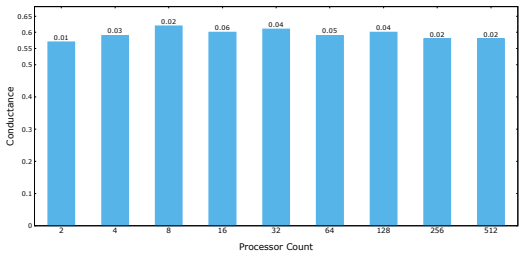
### 5.5.2 Distributed Performance Analysis

#### Workload Balancing

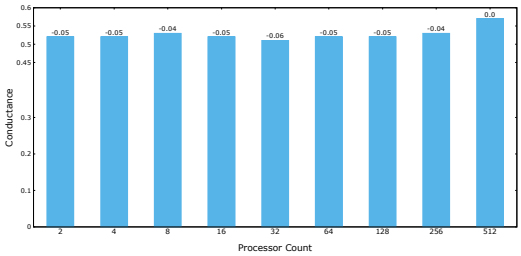
We used *Metis* [70] graph partitioner to distribute workload among processors. The purpose is not only to ensure equal workload balance among individual rank but also to attain minimizing edge-cut across the different partition sub-graphs to reduce the effect of vertex bouncing problem as mentioned in section 4.1. To ensure that each process shares an equal amount of computational workload, each processor is given the subset of vertices and corresponding edges as returned by the *Metis* partitioner. Across different iterations, we have observed that the number of vertices each processor has to deal with is fairly equal. As a result, each of the processors takes



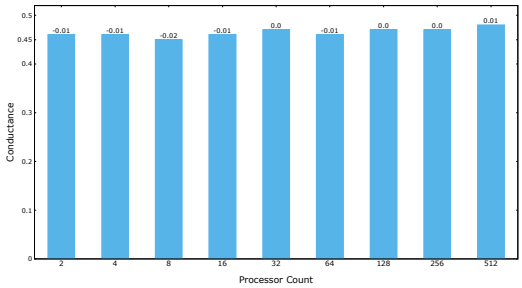

(a) DBLP Network




(b) Youtube Network




(c) Wiki-topcats Network




(d) soc-Pokec Network

Figure 5.3.: Change of conductance values across the different number of MPI processes for (5.3a) *DBLP* network, (5.3b) *Youtube* network, (5.3c) *Wiki-topcats* network, and (5.3d) *soc-Pokec* network. The numeric value on top of each histogram bar of each figure demonstrates the change of conductance compared to the value of conductance for sequential algorithm where a positive value indicates higher conductance and a negative value indicates lower conductance than the sequential version. The lower the value of conductance is, the better the quality of the discovered communities is.

almost an equal amount of time to complete execution of the algorithm as shown in figures 5.4 and 5.5.

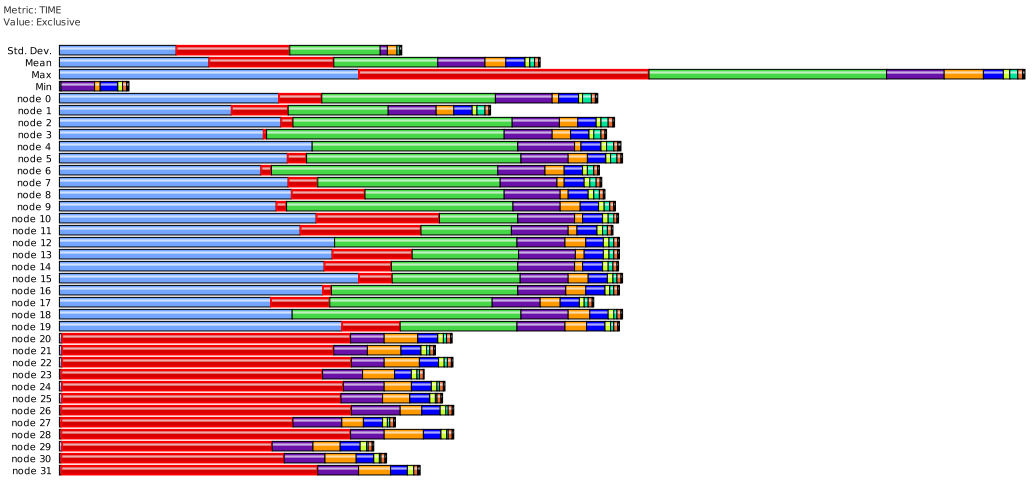



Figure 5.4.: Workload imbalance resulting from naive vertex distribution across processes

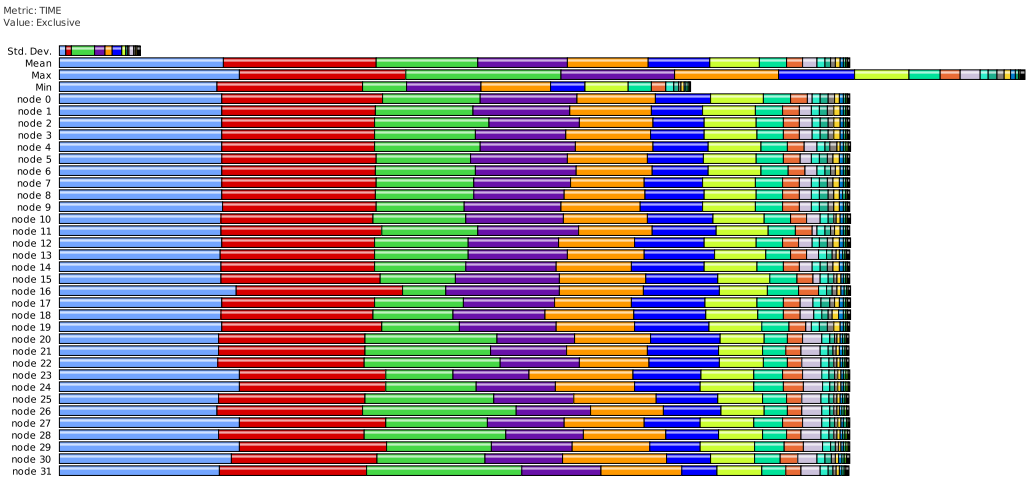



Figure 5.5.: Balanced workload across processes resulting from workload distribution by *Metis* partitioner

Speedup and Parallel Efficiency

We measure the speedup and time-performance using the networks in our experimental dataset. Figure 5.6 depicts the runtime of our algorithm for 3 different

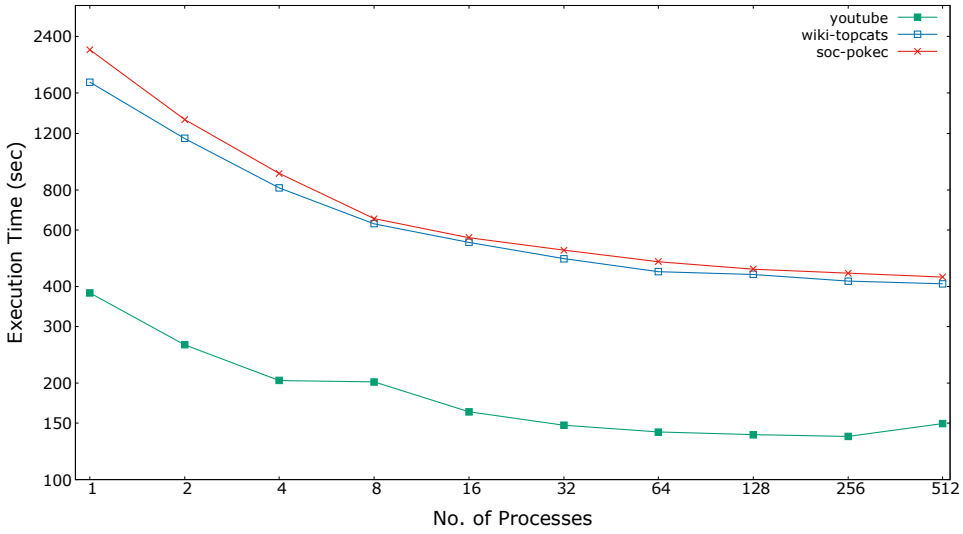



Figure 5.6.: Reduction of processing time for networks of different sizes from a single process to 512 processes

networks using a different number of processors. From those figures, it can be observed that we have achieved high scalability of reaching 512 processors for the bigger networks (e.g. *wiki-topcats, soc-Pokec*). For a fairly large network like *Youtube*, we have achieved consistent scalability improvement using up to 256 processors. For smaller networks we used for our experiments (e.g. *Amazon, DBLP*) the benefit of scalability is overruled by *MPI* communication cost across increased number of processors. Communication cost is dependent on the underlying network infrastructure of the computation nodes whereas the computation cost is controlled by the amount of computation to be performed due to the size of the network dataset. For bigger networks there is much computation to be performed and thus we can reach higher scalability. This is natural and expected.

In table 5.3 we have shown the maximum speedup of our algorithm achieved using the different number of processors in comparison to the sequential runtime of our algorithm. The speedup gains get higher for bigger graphs with the increasing

number of processors. For smaller networks, the speedup gains decrease after rising to a limit. One important thing is, the amount of speedup we can achieve depends highly on the problem type we are dealing with. For the problem of Infomap, the speedup gain is not radically higher as also evident from the work of [64] although they have used a significant number of processors.

Table 5.3: Speedup factors on various social and information networks.

| **Network** | **Speedup** |
|---|---|
| Amazon | 1.64 |
| DBLP | 1.92 |
| Youtube | 2.80 |
| wiki-topcats | 4.25 |
| soc-pokec | 5.10 |

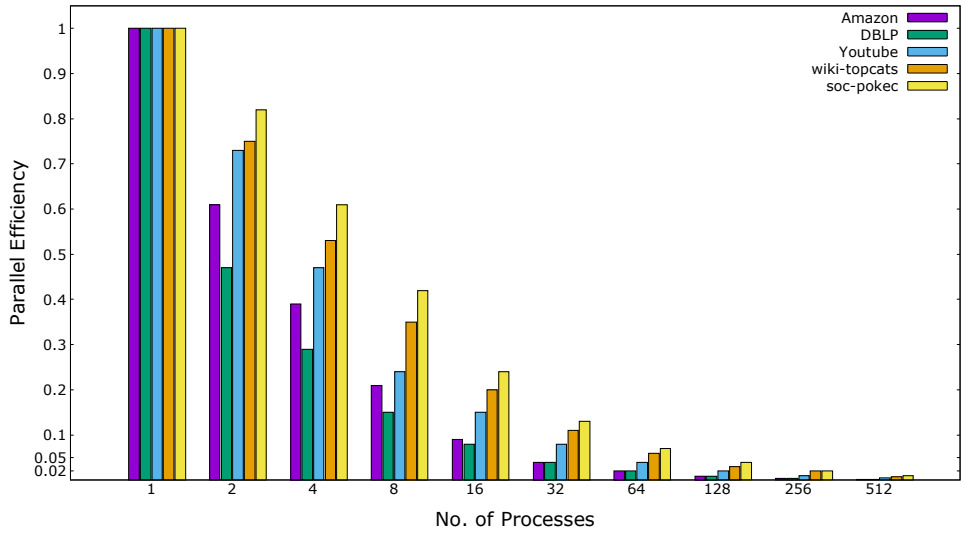



Figure 5.7.: Degree of parallelism obtained against different processor count

We have used parallel efficiency measure to perform quality analysis of our distributed implementation in terms of workload balancing and effect of increasing processing nodes. The Parallel efficiency $\varepsilon$ of an algorithm compares the parallel runtime to the best possible runtime assuming perfect scalability [61]. The parallel efficiency $\varepsilon = \frac{T_{seq}}{pT(p)}$, where $p$ is the number of parallel units, $T(p)$ is the time with $p$ parallel units, and $T_{seq}$ is the time of the sequential version.

Figure 5.7 depicts parallel efficiency in the form of histogram plot for the different networks across the different number of processes in the distributed platform. The higher the change in the histogram bar is, the less amount of efficiency gain is obtained by increasing the number of processing units. On the other hand, the less change in height of the histogram bar is with an increasing number of processors signifies a greater amount of parallelism. In figure 5.7, we can see the histogram bar for *soc-Pokec* has less change in height than others followed by *Wiki-topcats* and the rest for an increasing number of processes until all of them converge close to 0. It means for larger networks our algorithm delivers better parallel efficiency than smaller ones which is understandable as big networks need more computation work which can benefit from adding more processors. Converging close to 0 means adding extra processing units for computation may not benefit the parallel efficiency.

In table 5.3 we showed the maximum speedup we achieved for different networks we used for our experiments. The notable thing is for every network the speedup is not the same as the number of edges and vertices of the network play an important role in computation and communication costs. Speedup for every network in the dataset is measured against the runtime of detecting communities in a single processing unit.

Table 6.1: Comparison of our work with state-of-the-art techniques

| **Work Name** | **Type** | **Strength** | **Weakness** |
|---|---|---|---|
| Infomap | Sequential | Highly accurate | Computationally expensive |
| RelaxMap | Shared-memory parallelism | Highly accurate as sequential Infomap | Limited scalability |
| Gossipmap | Asynchronous distributed-memory parallelism | Less inter-process communication | Scalability up to 128 processes |
| Distributed Infomap | Synchronous distributed-memory parallelism | Highly accurate as sequential Infomap | Moderate speedup |
| Hybrid Infomap | Synchronous distributed+shared memory parallelism | High accuracy & high speedup gain | |

## 6.1 Experimental Setup

To compare runtime performance with existing distributed implementation [63] Gossipmap we have used our local computing servers in the department of CS at UNO. The server is a single computing node with 32 processing cores and 512 GB of memory. The operating system used is Ubuntu 16.04 of codename Xenial Xerus. The reason behind using our local computing server instead of the more powerful LONI [71] system is the user level restriction in installing required libraries in a publicly shared computing domain. Gossipmap uses Graphlab Powergraph [78] as the building framework which we could not install in the LONI server.

## 6.2 Comparison with GossipMap

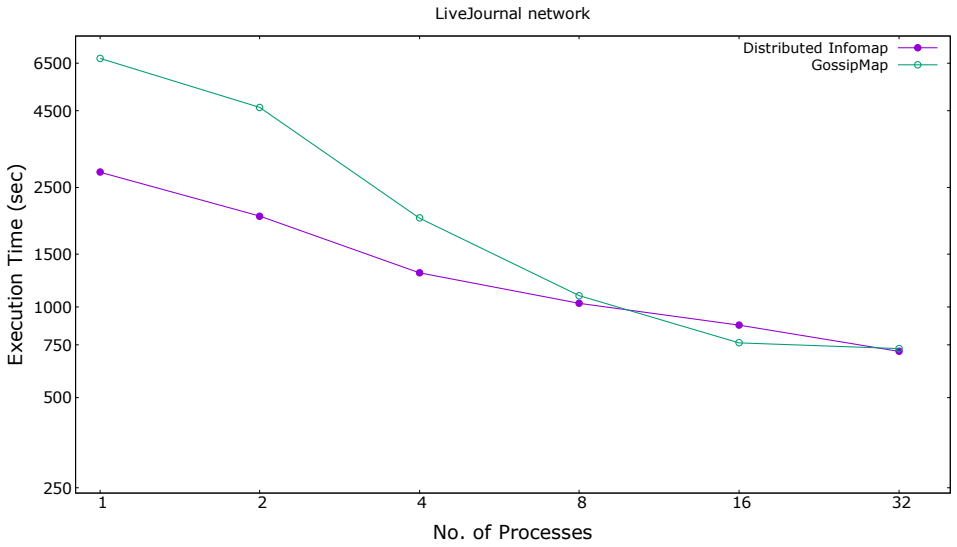



Figure 6.1.: Runtime comparison between Gossipmap and our distributed Infomap for the network *LiveJournal* for up to 32 MPI processes

We report the runtime comparison for three different networks (*LiveJournal*, *soc-Pokec* and *Wiki-topcats*) executed by Gossipmap and our distributed Infomap. Fig-

ures 6.1, 6.2, and 6.3 illustrate the outcome. From these figures, it can be realized the sequential runtime performance of our implementation is way better than the Gossipmap. For instance, in figure 6.1 the sequential completion time for finding communities is 6734.53 seconds for Gossipmap whereas the sequential runtime for our distributed implementation is 2813.93 seconds with runtime reduction of 2.40×. Gossipmap seems to get better parallel runtime reduction because of this poor sequential execution time. However, we observed in all the figures our approach is getting a smooth decrease in runtime with an indication of better utilization of CPU resources or good parallel efficiency in the context of MPI processes. Also, the change of runtime for 16 to 32 MPI processes almost become flat as evident in figures 6.1 and 6.2 indicating that parallel efficiency gain is getting poor for a higher number of MPI processes in Gossipmap which is not the case for our distributed Infomap. We did not test on a higher number of MPI processes (e.g., 64 or 128 processes) because of the CPU core limitation of our local computing server. That kind of test may not deliver the genuine scalability performance of Gossipmap and our distributed Infomap.

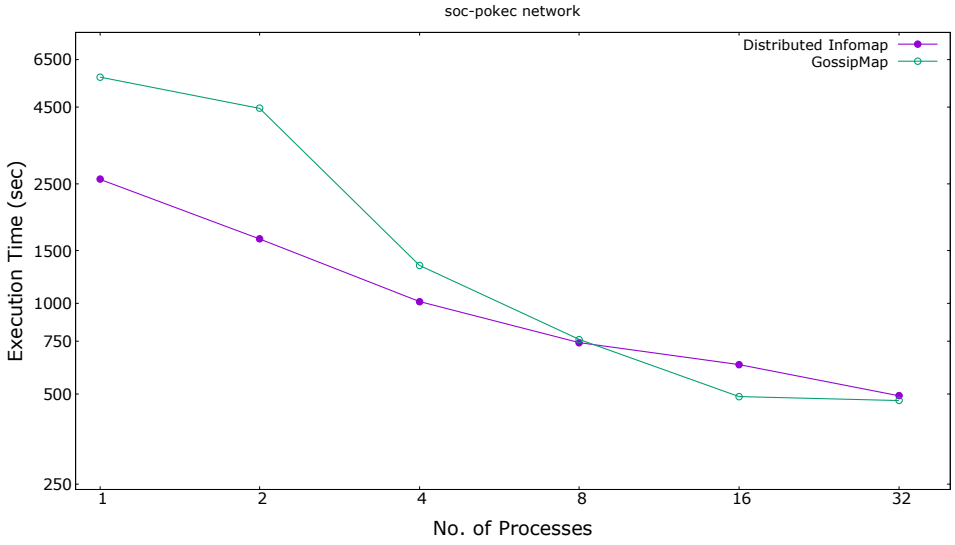



Figure 6.2.: Runtime comparison between Gossipmap and our distributed Infomap for the network *soc-Pokec* for up to 32 MPI processes

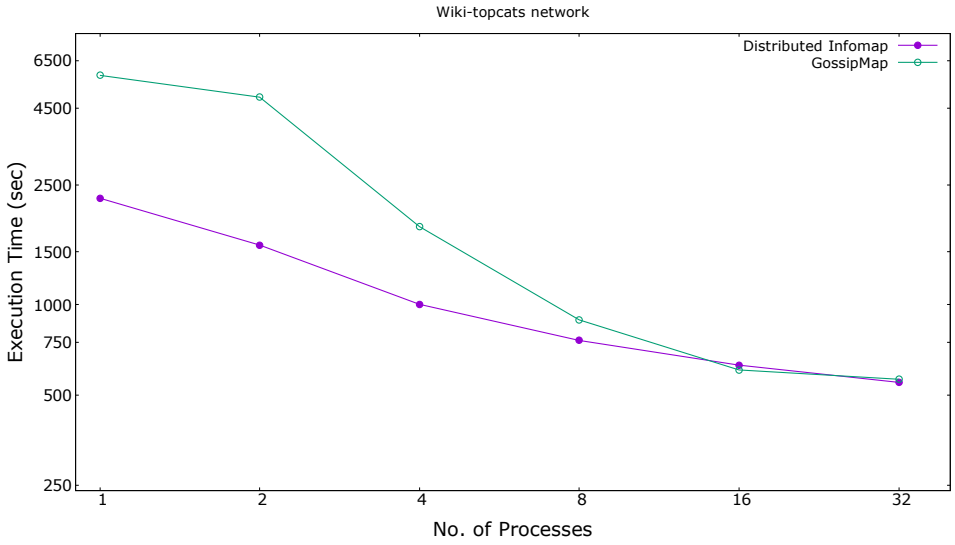



Figure 6.3.: Runtime comparison between Gossipmap and our distributed Infomap for the network *Wiki-topcats* for up to 32 MPI processes

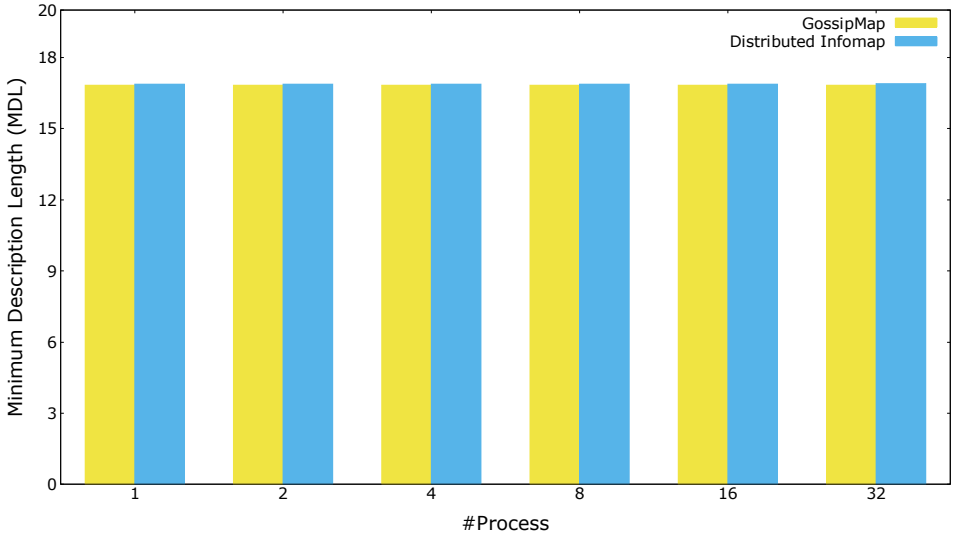



Figure 6.4.: Minimum description length (MDL) comparison after convergence for the *LiveJournal* network between Gossipmap and distributed Infomap.

When making a qualitative comparison in the context of MDL for the *LiveJournal* network we did not see any significant difference between Gossipmap and our distributed Infomap. It indicates that both of the implementations can converge to some point delivering similar quality of the discovered communities.

# 7 HYBRID (DISTRIBUTED + SHARED) MEMORY PARALLELISM

The hybrid implementation of Infomap is a continuation of our previous work [65] on distributed Infomap. In our hybrid design, we overcome the limitation of not being able to process very large networks in our distributed Infomap. The hybrid work has shown scalability with the highest amount of computing resources we could get as an individual researcher from the LONI system and has a promising prospect of scaling up to billion size networks given a bigger high-performance computing platform than LONI.

## 7.1 Experimental Setup

For the experiments in the hybrid platform, we used the LONI [71] clusters. We have used 10 OpenMP threads in each of the MPI processes to ensure maximum speedup gain in each of the computing nodes in the clusters. In each computing node in LONI clusters, there are 20 processing cores. Therefore, we ran 2 MPI processes per computing node each having 10 OpenMP threads to ensure maximum performance. Because of the resource limitation of how many computing nodes a researcher can request for computation in LONI, we could not go beyond 128 computing nodes and test hybrid performance beyond 256 MPI processes.

## 7.2 Algorithmic Analysis and Performance Measure

Our distributed algorithm on Infomap delivers high scalability up to 512 processors. Community detection using the Information-theoretic approach is highly sequential in nature. In certain parts of the algorithm, applying distributed paral-

lelism incurs significant communication cost across processors which outweighs the benefit of distributed computation. However, we observed that those parts of the algorithm can still exploit the benefit of shared memory parallelism using multiple threads. Therefore, we have used OpenMP to use shared memory parallelism inside our distributed algorithm. We have extended our distributed algorithm as we described in chapter 4 to hybrid implementation (MPI+OpenMP). We found significant performance benefit using this approach over the state-of-the-art distributed information-theoretic approaches without compromising the quality of the discovered communities. Analyzing the differences between our distributed-memory parallel Infomap and hybrid approach can be more realizable if we look at the algorithm 3.

The significant difference between the distributed algorithm and hybrid algorithm is seen in line $8-11$ where we used $t$ number of OpenMP threads in each MPI process to compute the vertex visit rate using power iteration. For bigger networks, the outcome of this approach is highly blissful. We reused the spawned threads (t threads) in computing exit probability inside each process (line $13-15$). In the community detection phase (line $17-26$), OpenMP threads are utilized again to speed up the supernode creation and parameter synchronization phases (line $24-25$) which is another difference between distributed Infomap and hybrid Infomap. The outcome of using OpenMP threads inside each MPI process to speedup computation turns out highly efficacious as we report in subsequent comparisons.

Figure 7.1 shows the breakdown of execution time for different networks. The smaller networks such as *Amazon, DBLP* do not have that much computation that can exploits the benefits of parallelism. Consequently, the line becomes almost flat after $16-32$ processes. For larger networks the performance gain across an increasing number of processors is realizable. For *Youtube* network with approximately $1.1M$ vertices and around $3M$ edges, the execution time for a single process is around

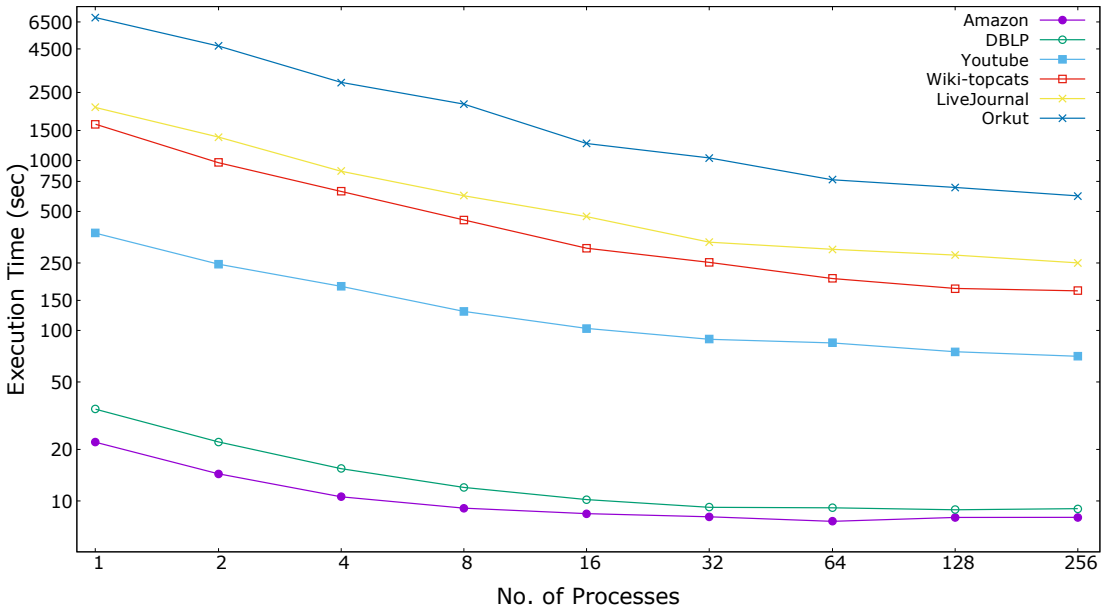



Figure 7.1.: Execution time comparison (drawn in log scale)

370 seconds which reduces to around 70 seconds for 256 processes. For bigger networks such as *Wiki-topcats* with $1.7M$ vertices and $28M$ edges, *LiveJournal* with $4M$ vertices and $34.6M$ edges, the scalability curves are steeper. For instance, for *Wiki-topcats* network we achieved a processing time of 171 seconds for 256 processes where it takes 1630 seconds to compute in sequential algorithm. For the *LiveJournal* network, the sequential processing time is 2040 seconds and the parallel processing time is 249 seconds for 256 processes. The massive network of our experiment is the *Orkut* social network with $3M$ vertices and $117M$ edges. The sequential algorithm takes 6888 seconds to discover communities whereas it takes 615 seconds to discover communities in 256 processes. This is a massive performance boost over sequential execution time.

In figure 7.2, we illustrated the performance gain in terms of speedup. Our hybrid approach obtained better speedup than state-of-the-art techniques to the best

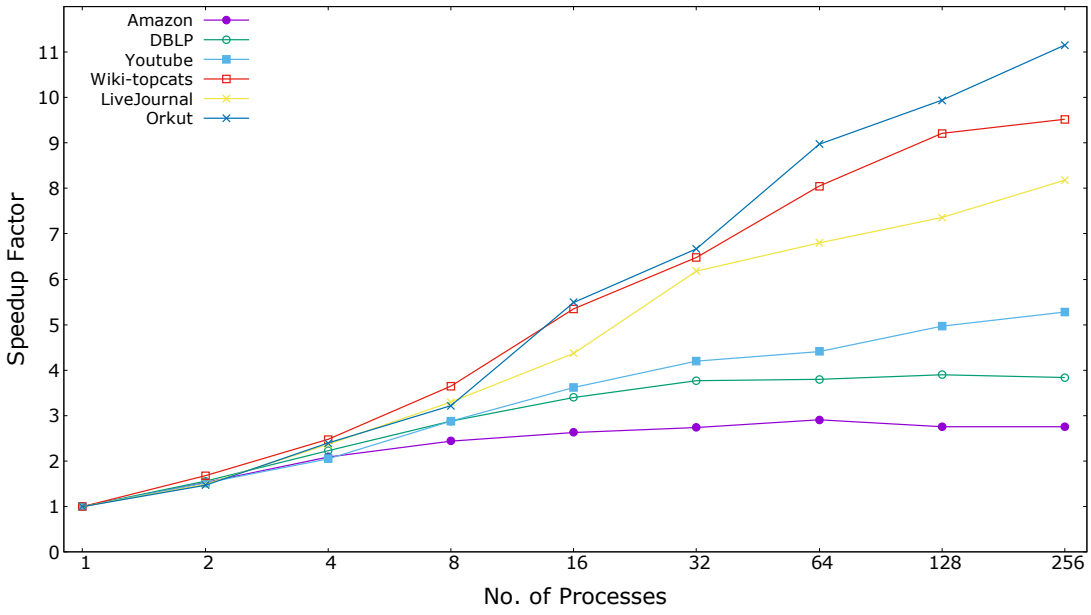



Figure 7.2.: Speedup factor achieved for different networks

of our knowledge. For smaller networks in our dataset, the speedup gain is comparable to state-of-the-art techniques. However, for a large network such as *LiveJournal*, we have achieved a much better speedup (8.18×) than the work of Zeng et al. [64] which achieves a speedup of 3.05× despite using thousands of processes. The highest speedup they achieved in their work is 6.02× for *UK-2007* network whereas the highest speedup we achieved is 11.15× with our largest network of *Orkut*. It also demonstrates that the size of the networks controls the speedup gain in our algorithm. The bigger the network is, the higher the speedup gain is as evident from the curves in figure 7.2.

Figure 7.3 illustrates a coarse estimation of how much time on average it can take to process a network. We have seen in the speedup figure 7.2 that we get higher speedup for bigger networks, that observation is also corroborated by the findings in figure 7.3. Although, the number of vertices in a network also plays a role in the pro-

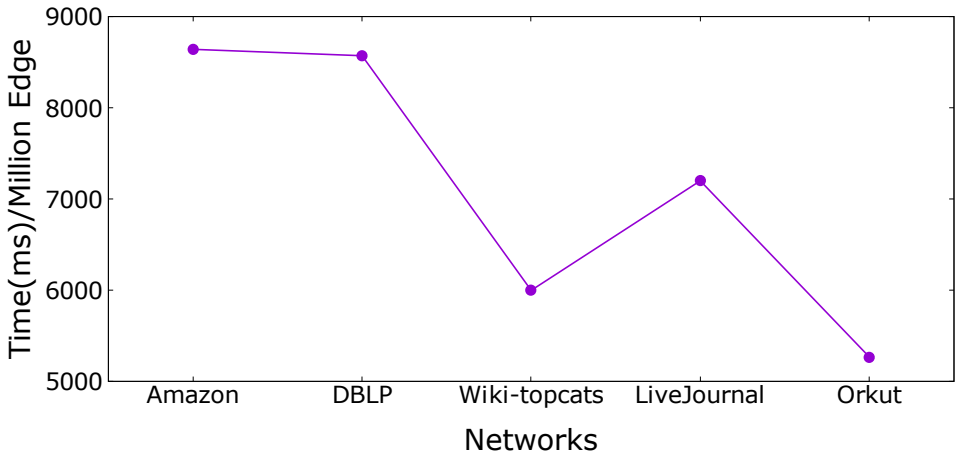



Figure 7.3.: Time taken on average in millisecond for processing per million of edges for sample networks. The larger the networks, the time taken to process a million edge gets smaller with a few exceptional cases.

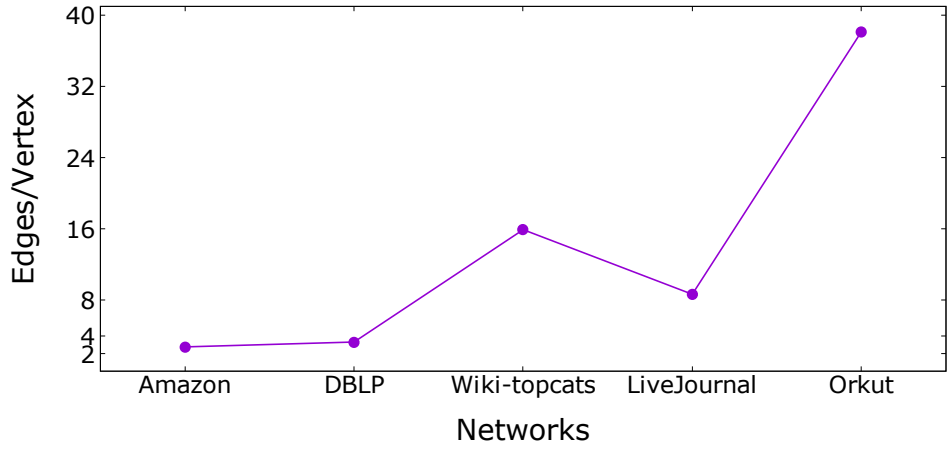



Figure 7.4.: Average edge distribution per vertex determines the speedup gain and processing time.

cessing time but the number of edges majorly determines the amount of computation that needs to be performed. It can be observed that our algorithm achieves better

parallelism when there is much computation to perform which is the case for bigger networks. It can be also seen that we achieved better speedup for the network *Wiki-topcats* than the network *LiveJournal* in figure 7.2, a similar fact is also observed in figure 7.3 where we get a spike for the network *LiveJournal* although the processing time is generally decreasing per-million of edges for the networks. Also, there is another previous observation verified by the current observation in figure 7.3. We have used *Metis* [70] edge-cut partitioner to divide the workload among processes. Before that, we used the naive vertex-based partitioner which turn out to be not-so-good partitioning strategy. Therefore, we can conclude that to ensure workload balancing, an equal edge-based partitioning is more effective than an equal vertex-based partitioning given the nature of this algorithm.

Figure 7.4 is a complementary illustration to the findings of figure 7.3 and figure 7.2. The average number of edges per vertex determines the potential speedup gain. If network $G_a$ has $V_a$ vertices and $E_a$ edges whereas the network $G_b$ has $V_b$ vertices and $E_b$ edges, the average number of edges per vertex for network $G_a$ is $\bar{e}_a = E_a/V_a$ and the average number of edges per vertex for network $G_b$ is $\bar{e}_b = E_b/V_b$. Based on the observation from our findings illustrated in figures 7.2, 7.3 and 7.4, we can say that if the execution speedup achieved for network $G_a$ is some positive real number $s_a$ and for network $G_b$ is some positive real number $s_b$ and $\bar{e}_a > \bar{e}_b$, then $s_a > s_b$.

In figure 7.5, we show the parallel efficiency of our algorithm for the experiment dataset across the different number of processes. We used the same formula of the parallel efficiency $\varepsilon = \frac{T_{seq}}{pT(p)}$ for our distributed performance analysis here in this hybrid approach, where $p$ is the number of parallel units, $T(p)$ is the time with $p$ parallel units, and $T_{seq}$ is the execution time of the sequential version. The parallel efficiency falls with the increasing number of processes which is real and expected. A significant reason is the communication cost increasing with the number of processes

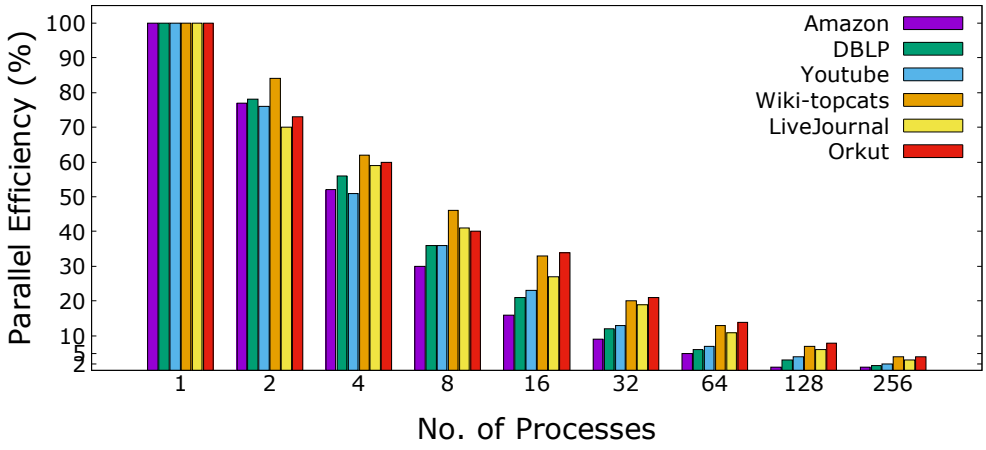



Figure 7.5.: Parallel efficiency (%) corresponding to the number of processes

outweighs the benefit of parallel performance gain. Being highly sequential in nature, the communication cost is inevitable for synchronization across processes in Infomap. However, we continue to gain better parallel efficiency for larger networks such as *Wiki-topcats*, *LiveJournal*, and *Orkut* with more than 20% using 32 processes.

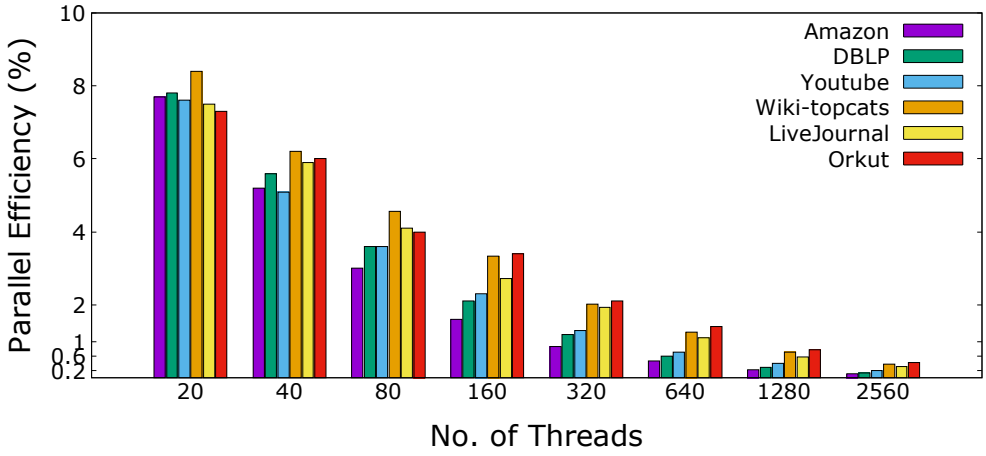



Figure 7.6.: Parallel efficiency (%) corresponding to the number of threads

As our hybrid approach also consists of many threads, we illustrate the parallel efficiency gain for the different number of threads used in figure 7.6. Each of the MPI processes spawns 10 OpenMP threads. The number of used threads therefore also increases with the number of used MPI processes. For instances, the total number of threads used for running the experiments with 128 MPI processes are 1280. We have used OpenMP threads in calculating rank vector for each vertex and creating super nodes during the module update process. The amount of parallel efficiency we achieved as shown in figure 7.6 although seems small but that is because it reflects the parallel efficiency gain for only that small portion of our algorithm computing the rank vector and the modules-update.

## 7.3 Quality Measure

We have achieved a significant performance gain by using the hybrid approach. To ensure this improvement is not obtained by compromising the quality of the detected communities, we compare against the same quality metrics we used in our distributed algorithm comparative study, i.e., Modularity, Conductance, and Minimum Description Length (MDL) of the convergence. We compared the values of those metrics across different networks for the sequential algorithm and the hybrid one with maximum number of processes we used. We obtained almost uniform results for different networks for the minimum and the maximum number of processes that corroborates the fact that the quality of our hybrid approach does not vary over networks or the number of MPI processes.

In figure 7.7, we show the Conductance measure for sequential vs 256 processes. In all of the networks, the hybrid approach returns either the same or similar conductance values. It is important to note a lower value of conductance means a higher quality of the detected communities. For all the networks we have observed less than 5% change in Conductance values. In case of *Wiki-topcats* network, we received 3%

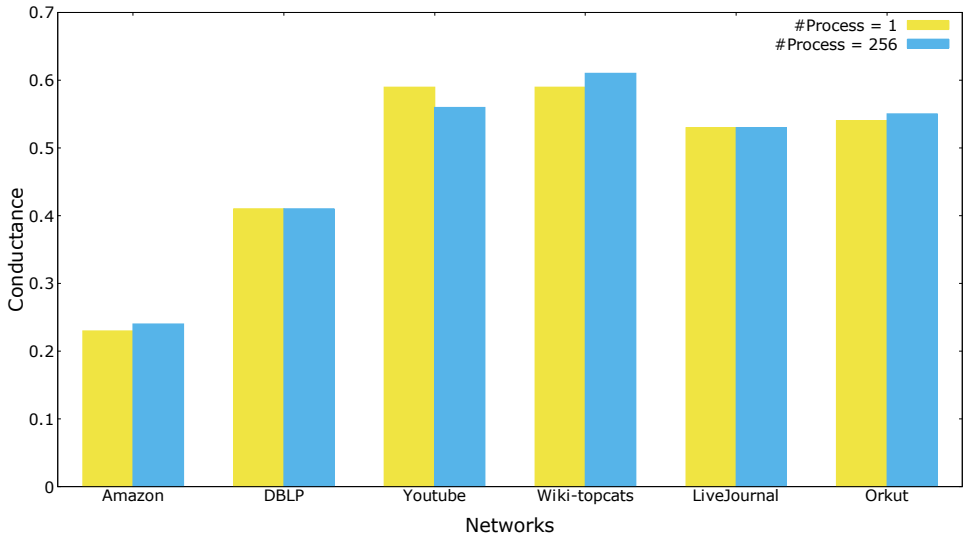



Figure 7.7.: Conductance measured for minimum (1) and maximum (256) number of processes for different networks

higher conductance value than the sequential one. One possible reason for such fluctuation can be the conversion of network in CSR (Compressed Sparse Row) format. All the networks we have used in our experiments are undirected except Wiki-topcats. Conversion to CSR format results in the undirected network for the corresponding directed one adding extra edge information. This might have happened in the case of *Wiki-topcats* network and doing distributed processing of that network resulted in a bit lower-quality of the detected community.

A similar trend is also observed for the quality of the discovered community by Modularity in figure 7.8. For all the networks we observed the same value of the Modularity metrics except for the *Wiki-topcats* network where the Modularity value is less by 7.5%. The reason is explained above for the conductance metric. It is important to note that lower Modularity value means less quality of the discovered community.

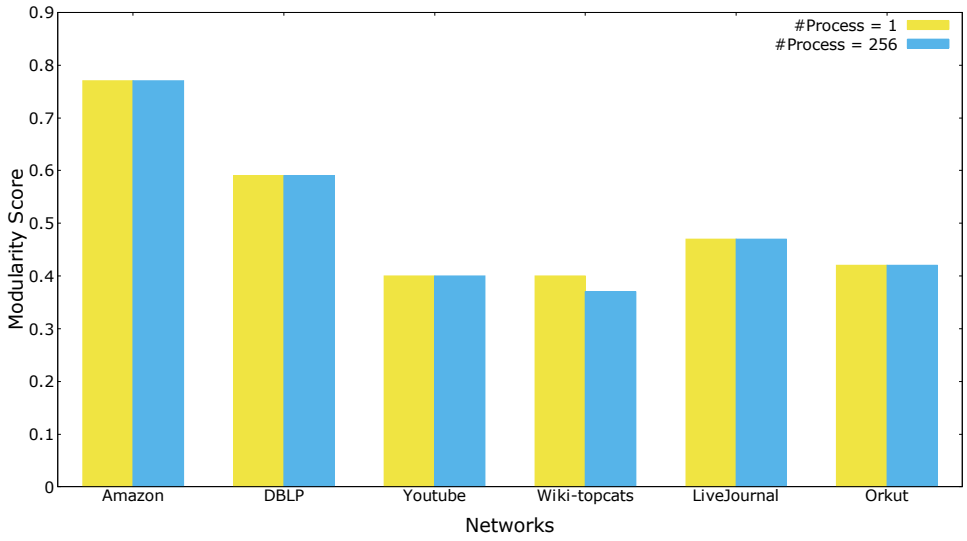



Figure 7.8.: Modularity measured for minimum (1) and maximum (256) number of processes for different networks

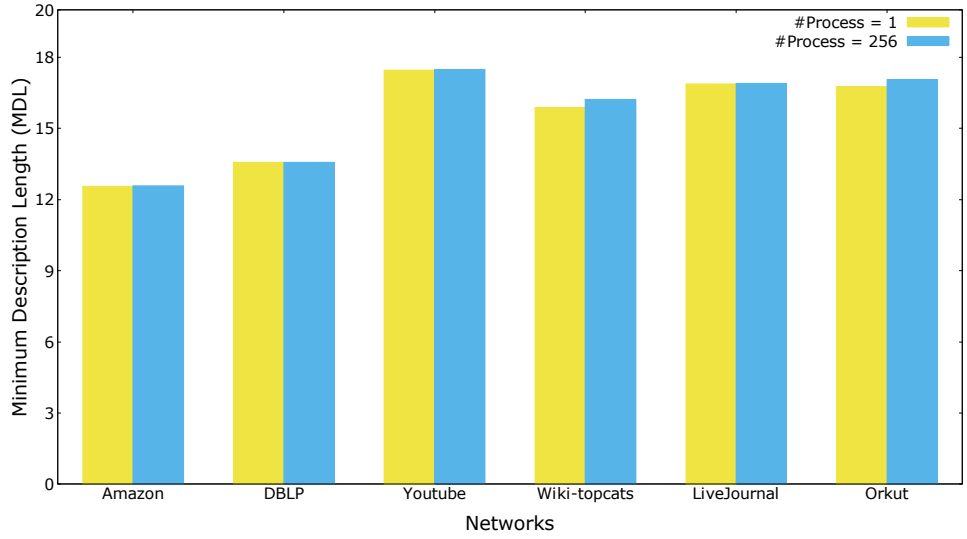



Figure 7.9.: Convergence Minimum Description Length (MDL) for minimum (1) and maximum (256) number of processes

In figure 7.9, the resultant Minimum Description Length (MDL) value are illustrated. The MDL value indicates the average number of bits per step required to

describe an infinite random walk on a network partitioned based on the resultant number of discovered communities. For all the networks we have observed the same MDL values for both 256 processes and sequential algorithm with 2% under-convergence for *Wiki-topcats* network and 1% under-convergence for *Orkut* network. As the event of under-convergence is expected in distributed platform as also observed by Bae et al. [61], this minimum change of the convergence MDL is an expected outcome for a distributed algorithm.

**Algorithm 3** Hybrid Infomap

**Require:** A graph $G(V, E)$, $V$ total vertices, $E$ total edges, $N \leftarrow |V|$

**Ensure:** $M : M \leqslant N$, M is the total number of communities, $M \ll N$

$m_i$, i$^{\text{th}}$ module

$q_{m_i}$, exit probability of module $m_i$

$\tau$, minimum threshold for codelength improvement

$L_{old}$, codelength of previous iteration

$L$, codelength of current iteration

$P$, number of MPI processes spawned

$t$, number of OpenMP threads spawned

**for** $i = 1$ to $N$ in $t - way$ parallel **do**

    calculate initial vertex visit rate $p_{v_i} \leftarrow 1/N$

    compute vertex visit rate $p_{v_i}$ by power iteration

**end for**

declare initial total module $M \leftarrow \{m_i = \{v_i\} | v_i \in V\}$

**for** $m_i = 1$ to $M$ in $t - way$ parallel **do**

    calculate exit probability $q_{m_i}$

**end for**

calculate initial codelength $L \leftarrow L(M)$

**do**

    $L_{old} \leftarrow L$

    **for** $i = 1$ to $(N/P \leftarrow metis)$ in parallel **do**

        pick randomly a vertex $v_i$

        $m_{new} \leftarrow findNewBestModule(v_i)$

        calculate $L$

    **end for**

    update $M \leftarrow convertModulestoSuperNode()$ in $t - way$ parallel

    synchronize parameters across all processes

**while** $(L_{old} - L) > \tau$

**return** $M$

# 8 CONCLUSION

In this thesis, we have presented our design of distributed-memory parallel Infomap capable of discovering communities with similar quality to sequential Infomap. Our experimental analysis shows that we can scale down the execution time of processing massive networks. While doing experimental analysis with our distributed Infomap design, we observed that certain parts of our algorithm can benefit from parallelism but communication cost is dominating the parallel computation benefit. We then redesign a hybrid algorithm which led us to even more scaling down of the execution time and greater speedup compared to the state-of-the-art techniques. This happens without compromising the quality of the detected communities while processing even bigger networks with 100 millions of edges. Our hybrid work overcomes the limitation of our distributed design. In the future, we want to extend our endeavor to deal with the dynamic network and target-oriented community search.

LIST OF REFERENCES


[1] Mason A. Porter, Jukka-Pekka Onnela, and Peter J. Mucha. Communities in networks. *ArXiv*, abs/0902.3788, 2009.

[2] Santo Fortunato. Community detection in graphs. *ArXiv*, abs/0906.0612, 2010.

[3] M. Newman. Communities, modules and large-scale structure in networks. *Nature Physics*, 8:25–31, 12 2011.

[4] Bhaskar DasGupta and Devendra Desai. On the complexity of newman's community finding approach for biological and social networks. *Journal of Computer and System Sciences*, 79(1):50 – 67, 2013.

[5] A. Karataş and S. Şahin. Application areas of community detection: A review. In *2018 International Congress on Big Data, Deep Learning and Fighting Cyber Terrorism (IBIGDELFT)*, pages 65–70, Dec 2018.

[6] M. Halappanavar, H. Lu, A. Kalyanaraman, and A. Tumeo. Scalable static and dynamic community detection using grappolo. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sep. 2017.

[7] Tiago P. Peixoto. Efficient monte carlo and greedy heuristic for the inference of stochastic block models. *Phys. Rev. E*, 89:012804, Jan 2014.

[8] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75 – 174, 2010.

[9] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74(3), Sep 2006.

[10] M. E. J. Newman. Spectral methods for community detection and graph partitioning. *Physical Review E*, 88(4), Oct 2013.

[11] Luca Donetti and Miguel A Muñoz. Detecting network communities: a new systematic and efficient algorithm. *Journal of Statistical Mechanics: Theory and Experiment*, 2004(10):P10012, Oct 2004.

[12] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[13] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69:026113, Feb 2004.

[14] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, Oct 2008.

[15] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[16] Tiago P. Peixoto. Parsimonious module inference in large networks. *Phys. Rev. Lett.*, 110:148701, Apr 2013.

[17] Tiago P. Peixoto. Entropy of stochastic blockmodel ensembles. *Phys. Rev. E*, 85:056122, May 2012.

[18] Brian Karrer and M. E. J. Newman. Stochastic blockmodels and community structure in networks. *Phys. Rev. E*, 83:016107, Jan 2011.

[19] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[20] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E*, 78:046110, Oct 2008.

[21] Andrea Lancichinetti and Santo Fortunato. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Phys. Rev. E*, 80:016118, Jul 2009.

[22] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[23] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[24] S Szabó. Parallel algorithms for finding cliques in a graph. *Journal of Physics: Conference Series*, 268:012030, jan 2011.

[25] Shaikh Arifuzzaman and Bikesh Pandey. Scalable mining, analysis, and visualization of protein-protein interaction networks. *International Journal of Big Data Intelligence (IJBDI)*, 6(3/4), 01 2019.

[26] M. A. Motaleb Faysal and S. Arifuzzaman. A comparative analysis of large-scale network visualization tools. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 4837–4843, Dec 2018.

[27] Shaikh Arifuzzaman, Maleq Khan, and Madhav V. Marathe. PATRIC: a parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM 2013), San Francisco, CA, USA*, pages 529–538, October 2013.

[28] S. Arifuzzaman and M. Khan. Fast parallel conversion of edge list to adjacency list for large-scale graphs. In *Proceedings of the 23rd High Performance Computing Symposium (HPC 2015), Alexandria, VA, USA*, pages 17–24, April 2015.

[29] S. Arifuzzaman, Maleq Khan, and Madhav Marathe. A space-efficient parallel algorithm for counting exact triangles in massive networks. In *Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications (HPCC 2015), New York City, USA*, pages 527–534, August 2015.

[30] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Fast parallel algorithms for counting and listing triangles in big graphs. *ACM Trans. Knowl. Discov. Data*, 14(1), December 2019.

[31] Aaron Clauset, Mark E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Phys. Rev. E*, 70:066111, Dec 2004.

[32] Roger Guimerà, Marta Sales-Pardo, and Luís A. Nunes Amaral. Modularity from fluctuations in random graphs and complex networks. *Phys. Rev. E*, 70:025101, Aug 2004.

[33] Claire P. Massen and Jonathan P. K. Doye. Identifying communities within energy landscapes. *Phys. Rev. E*, 71:046101, Apr 2005.

[34] Andres Medus, Guillermo Acuña, and CO Dorso. Detection of community structures in networks via global optimization. *Physica A: Statistical Mechanics and its Applications*, 358:593–604, Dec 2005.

[35] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences*, 101(9):2658–2663, 2004.

[36] Sanjukta Bhowmick and Sriram Srinivasan. *A Template for Parallelizing the Louvain Method for Modularity Maximization*, pages 111–124. Springer New York, New York, NY, 2013.

[37] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. Fast algorithm for modularity-based graph clustering. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, page 1170–1176. AAAI Press, 2013.

[38] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritizing iterative computations. *IEEE Transactions on Parallel and Distributed Systems*, 24(9):1884–1893, Sep. 2013.

[39] Chun Yew Cheong, Huynh Phung Huynh, David Lo, and Rick Siow Mong Goh. Hierarchical parallel algorithm for modularity-based community detection using gpus. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 775–787, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[40] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo. Community detection on the gpu. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 625–634, May 2017.

[41] Christian L. Staudt and Henning Meyerhenke. Engineering high-performance community detection heuristics for massive graphs. In *2013 42$^{nd}$ International Conference on Parallel Processing*, pages 180–189, Oct 2013.

[42] Christian L. Staudt and Henning Meyerhenke. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):171–184, Jan 2016.

[43] Jianping Zeng and Hongfeng Yu. Parallel modularity-based community detection on large-scale graphs. In *2015 IEEE International Conference on Cluster Computing*, pages 1–10, Sep. 2015.

[44] Jianping Zeng and Hongfeng Yu. A study of graph partitioning schemes for parallel graph community detection. *Parallel Computing*, 58(C):131–139, October 2016.

[45] Naw Safrin Sattar and Shaikh Arifuzzaman. Parallelizing louvain algorithm: Distributed memory challenges. In *Proceedings of 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing (DASC), Athens, Greece*, pages 695–701, 2018.

[46] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarrià-Miranda, A. Khan, and A. Gebremedhin. Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895, May 2018.

[47] J. Reichardt and D. R. White. Role models for complex networks. *The European Physical Journal B*, 60(2):217–224, Nov 2007.

[48] Jake M. Hofman and Chris H. Wiggins. Bayesian approach to network modularity. *Phys. Rev. Lett.*, 100:258701, Jun 2008.

[49] M. E. J. Newman and E. A. Leicht. Mixture models and exploratory analysis in networks. *Proceedings of the National Academy of Sciences*, 104(23):9564–9569, 2007.

[50] Carolyn J Anderson, Stanley Wasserman, and Katherine Faust. Building stochastic blockmodels. *Social Networks*, 14(1):137 – 161, 1992. Special Issue on Blockmodels.

[51] Katherine Faust and Stanley Wasserman. Blockmodels: Interpretation and evaluation. *Social Networks*, 14(1):5 – 61, 1992. Special Issue on Blockmodels.

[52] Paul W. Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social Networks*, 5(2):109 – 137, 1983.

[53] Brian Karrer and M. E. J. Newman. Stochastic blockmodels and community structure in networks. *Phys. Rev. E*, 83:016107, Jan 2011.

[54] Tiago Peixoto. graph-tool.

[55] M. A. M. Faysal and S. Arifuzzaman. Fast stochastic block partitioning using a single commodity machine. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3632–3639, 2019.

[56] Ahsen J. Uppal and H. Howie Huang. Fast stochastic block partition for streaming graphs. *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6, 2018.

[57] A. J. Uppal, G. Swope, and H. H. Huang. Scalable stochastic block partition. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5, Sep. 2017.

[58] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song, D. Staheli, and S. Smith. Streaming graph challenge: Stochastic block partition. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–12, Sep. 2017.

[59] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: A comparative analysis. *Phys. Rev. E*, 80:056117, Nov 2009.

[60] Rodrigo Aldecoa and Ignacio Marìn. Exploring the limits of community detection strategies in complex networks. *Scientific Reports*, 3:2216, Jul 2013.

[61] Seung-Hee Bae, Daniel Halperin, Jevin West, Martin Rosvall, and Bill Howe. Scalable flow-based community detection for large-scale network analysis. In *2013 IEEE 13th International Conference on Data Mining Workshops*, pages 303–310, Dec 2013.

[62] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[63] Seung-Hee Bae and Bill Howe. Gossipmap: a distributed community detection algorithm for billion-edge directed graphs. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2015.

[64] Jianping Zeng and Hongfeng Yu. A distributed infomap algorithm for scalable and high-quality community detection. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 4:1–4:11, New York, NY, USA, 2018. ACM.

[65] M. A. M. Faysal and S. Arifuzzaman. Distributed community detection in large networks using an information-theoretic approach. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4773–4782, Dec 2019.

[66] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, September 1978.

[67] Peter D. Grünwald, In Jae Myung, and Mark A. Pitt. *Advances in Minimum Description Length: Theory and Applications (Neural Information Processing)*. The MIT Press, 2005.

[68] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[69] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, APR 1998.

[70] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[71] Louisiana optical network infrastructure. `http://hpc.loni.org/resources/hpc/system.php?system=QB2`.

[72] LSU HPC. Qb2 cluster. `http://www.hpc.lsu.edu/docs/guides.php?system=QB2`.

[73] Md A M Faysal. Distributed infomap. `https://github.com/Drakule-Mihawk101/DistInfomap.git`.

[74] Vladimir Batagelj and Andrej Mrvar. Pajek. `http://mrvar.fdv.uni-lj.si/pajek/`.

[75] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[76] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining Social-Network Graphs*, page 325–383. Cambridge University Press, 2 edition, 2014.

[77] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.*, 42(1):181–213, January 2015.

[78] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.

# VITA

Md Abdul Motaleb Faysal was born in a small village in the suburb area of Dhaka in Bangladesh. He is the 2$^{nd}$ youngest member in his family of the 6 siblings.

As a student, he finished his undergraduate degree of bachelor of science in Computer Science and Engineering (CSE) from Bangladesh University of Engineering and Technology (BUET) in the year 2014. He started his career as a software engineer in a software company in Bangladesh in the same year. To pursue higher education, Faysal resigned from his position as a software engineer in July, 2017.

Later, he joined the University of New Orleans (UNO) in fall 2017 as a Graduate Research Assistant in the Computer Science department to pursue his Ph.D. He is a part of the Big Data and Scalable Computing group supervised by Dr. Shaikh Arifuzzaman (Assistant Professor, CS at UNO). His research interest focuses on high-performance computing, graph algorithms, big data mining, and scalable computing.

Apart from the study and research, Faysal is often seen in the UNO campus playing cricket, soccer, and badminton with his friends. He loves to go on a long drive and travel places with scenic beauty.