FAST COMMUNITY STRUCTURE ANALYSIS OF CALL GRAPHS FOR MALWARE DETECTION

by

Pooja Patil

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the Degree of

Master of Science

Department of Computer and Information Technology

West Lafayette, Indiana

May 2019

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. John Springer, Chair

Department of Computer and Information Technology

Dr. Eric Matson

Department of Computer and Information Technology

Dr. Julia Taylor Rayz

Department of Computer and Information Technology

Approved by:

Dr. Eric T. Matson

Head of the Graduate Program

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Springer, for guiding and supporting me throughout my journey as a graduate student at Purdue University. I would also like to thank him for motivating and constantly encouraging me.

I would also like to thank my committee members Dr. Julia Taylor Rayz and Dr. Eric Matson for providing feedback and suggestions. Next, I would like to thank my parents and sister for their unconditional love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

APK	Android Application Package
GML	Graph Modeling Language
OS	Operating System
OOP	Object Oriented Programming

GLOSSARY

Community: “The division of network nodes into groups within which the network connections are dense, but between which are sparser.“ (Newman & Girvan, 2004, p. 1)

Call Graph: “A call graph consists of nodes, representing procedures, linked by directed edges, representing calls from one procedure to another.“ (Grove & Chambers, 2001, p. 689)

Modularity: Measures the quality of each partition. It helps in deciding if a partition is better than other partition. The value of modualrity ranges from -1 to 1. (Barabási, 2016)

ABSTRACT

Author: Patil, Pooja. M.S.
Institution: Purdue University
Degree Received: May 2019
Title: Fast Community Structure Analysis of Call Graphs for Malware Detection
Major Professor: John Springer

The use of graph-structured data in applications is increasing day by day. In order to infer useful information from such data, fast analytics and software tools are required. One of the graph analytics techniques used is community detection. Community detection is the technique of finding structural communities within a graph. Such communities are defined as groups which have highly connected nodes and have similarities with each other.

This research proposes a parallel heuristic for faster community detection using the parallel version of the Louvain algorithm: Grappolo. The Louvain algorithm is a hierarchical algorithm that focuses on modularity optimization. It gained popularity because of its ability to detect high-quality communities faster than the other existing community detection algorithms. However, the Louvain algorithm is a sequential algorithm. To reduce the execution time of the Louvain algorithm, a parallel version named Grappolo exists in the literature. This algorithm proposes parallel heuristics that address the challenges that occur due to parallelizing the sequential Louvain algorithm.

In this study, the researcher is investigating if Grappolo can be further parallelized to further reduce the execution time maintaining the quality of communities detected. To evaluate the proposed heuristic, it was tested on an OpenMP multithreaded environment. It was implemented on source codes of Android malware applications. However, as compared to Grapplolo, the proposed modified version resulted in higher execution times for the inputs tested. The modularity of the communities detected was similar to the Grappolo implementation.

CHAPTER 1. INTRODUCTION

This chapter provides an overview of the research conducted. This chapter includes an initial introduction to community detection followed by a brief description of the problem statement, the research question, the scope of the research, and significance of this study. It also briefs about the assumptions, limitations, and delimitations of this study.

1.1 Community Detection

Many systems can be represented as a form of a network with the set of nodes joined together by edges. A myriad of examples include World Wide Web, biological networks, technological networks, food webs, social networks, transportation systems, etc. Since all these networks are growing in size and complexity, the development of network analysis tools and algorithms is a topic of rising interest (Afsariardchi, 2012). One feature of networks that has been emphasized in recent study as a solution to analyze the rapid increasing networks is community structures. A community structure is the organization of vertices into groups such that there is a high density of edges within the groups as compared to between the groups (Barabási, 2016). The process of finding such community structures within a network is called as community detection.

1.2 Problem Statement

The sale of Android smartphones has increased in the past years. The reason behind its popularity is that it is an open source operating system and provides a world-class platform for creating apps and games. These third-party applications can be easily installed and downloaded from Google Play. This has motivated hackers to penetrate Android smartphones using malicious applications. Once a malware enters into the system, it performs various activities behind the scene such as stealing information

and signing up the user for various subscriptions. Hence such malicious software should be detected as early as possible. There are various existing signature-based malware detection approaches. But these approaches face challenges of code obfuscation and manual analysis of patterns.

This quantitative study focuses on fast detection of malware in an Android operating system using the community structures of malware source codes. To achieve this, the researcher implemented the modified version of the Grappolo algorithm for finding communities within a software network. The modifications on the algorithm are such that it will reduce the computational time of the Grappolo algorithm. According to Newman and Girvan (2004), modularity for a good community structure of a network ranges from 0.3 to 0.7, greater than this threshold is very rare. The modifications to the algorithms are such that it preserves the modularity of communities between the range 0.3 to 0.7. Detecting communities within a software network helps in understanding the underlying community structure of that software. Implementing community detection algorithms on Android malware helps in understanding the characteristics of malware which in turn helps in detecting malware.

1.3 Research Question

Can the proposed parallel version of the Grappolo algorithm perform faster malware source code analysis in an Android system as compared Grappolo, keeping the modularity index between 0.3 and 0.7?

1.4 Significance

Malware or malicious software is a computer program whose intent is to damage users and compromise the sensitive information of the user. Malwares are used to send spam emails, to commit web frauds, and to carry out many other illegal activities. Hence surfing the web, sharing information, and using social media is not as safe as it used to be. According to Kelly (2014), in 2013 Android was the target of 97% of the global mobile

malware. Also, in Q1 of 2018, coin malware ransomware has doubled (up to 86%) with 2.5 million new samples of malware (*McAfee Labs Threats Reports*, n.d.). Today, the increasing malware infestation is one of the biggest problems faced by the internet community. These attacks can pose a great threat to national security in the near future. Hence, these attacks must be discovered before they affect the victim and goes beyond control.

Anti-virus vendors try to keep up with the trend of malwares to protect the increasing number of Android users. Traditional solutions for detection and analysis of malwares include signature-based approach and behavior-based approach (Paleari, 2011). The above existing approaches face many challenges. The drawbacks of these approaches are that they lag behind, are easily influenced by code confusion mechanisms, and are inefficient for zero-day malware detection (Du, Wang, & Li, 2017).

To overcome these drawbacks, another approach for malware detection is the analysis of the community structures of call graphs of malware programs (Du et al., 2017). Community detection is used to divide a call graph of the malware into subgraphs. These subgraphs exhibit structural information (features) of the malware. Once the defender gets information on the features of the malware, using a machine learning approach the Android system can be trained to classify a particular software as benign or malicious software. The goal of this research is that the methodology used will help the community to detect malwares in real time with better accuracy and less computational time.

1.5 Scope

Community detection in call graphs has wide variety of applications like software evolution process modelling, software evolution prediction (Li, Zhao, Cai, Xu, & Ai, 2013), software structure interpretation and evolution (Qu, Guan, Zheng, Liu, Zhou, & Li, 2015), fault prediction (Qu, Guan, Zheng, Liu, Wang, et al., 2015), code refactoring, malware detection, etc. This study focuses on community detection of call graphs for malware detection.

The popularity of an Operating System (OS) and widespread use of an OS attracts the attention of hackers and motivates them to develop malwares and viruses to corrupt the devices running on that OS. Some of the popular OSs that are infected by malwares are Android, macOS, Windows, Linux, DOS, etc. (*Adoption Rate and Popularity*, n.d.). As number of Android users are increasing day by day, the threats against Android OS are also increasing. In Q1 2018, Kaspersky Lab reported 1,322,578 malicious software installations on Android (*Adoption Rate and Popularity*, n.d.). Hence, this study focuses on using community detection to analyze the structure of malware applications in an Android system.

Among the various community detection algorithms present in the literature, this study focuses on understanding the Louvain algorithm and its parallelized version, Grappolo. Also, this study aims to investigate if Grappolo can be further parallelized to reduce the execution time.

1.6 Assumptions

The assumptions for this study include:

- The execution environment would remain constant and worked with equal reliability and efficiency for both the algorithms considered for comparison.
- The malware samples (dataset) is a representative of the real-world malware families.
- The number and size of the communities are not known a prior.

1.7 Limitations

The limitations for this study include:

- Modularity is the only metric considered for evaluating the quality of the detected community structure.

- The malware dataset used is only limited to malware families from the Debrin malware dataset.

1.8 Delimitations

The delimitations for this study include:

- This research only focuses on static community detection.
- This study focuses only on the Grappolo algorithm. It does not consider other community detection algorithms for comparison.
- The study only uses function calls to malicious functions or APIs for community detection.

1.9 Summary

This chapter provided a brief introduction of the research conducted. It also underlined the scope, significance, research question, assumptions, limitations, delimitations, definitions, and other background information for the research project.

CHAPTER 2. REVIEW OF LITERATURE

This chapter starts with shedding some light on how software source codes can be represented as graphs. Later, it provides an insight into community detection and graph metrics. This is followed by providing a review on community detection algorithms in networks.

2.1 Software Systems as Graphs

As mentioned in the previous chapter, malware is an executable program. A program is made of tens of thousands of lines of codes. Such a huge program is difficult to maintain and monitor. Hence to simplify the source code of a software, use of Object Oriented Programming (OOP) has increased over time. OOP represents specific software modules and connections among those software modules. A program can be visualized as a network where the nodes will be functions (software modules) and edges will be the calls to these functions.

A graph in mathematical terms is a set of nodes and a set of connections between the nodes (edges). Graph theory has various applications in many research areas (Dunn, Dudbridge, & Sanderson, 2005). Graphs are extensively used in Biology and Bioinformatics area. For example, Dunn et al. (2005) used community detection for finding clusters of interconnected proteins in protein interaction networks. Balaban (1985) reviews the applications of graph theory in theoretical chemistry, chemical nomenclature, coding, and information retrieval/processing. Graph coloring is used for scheduling tasks like aircraft scheduling, task scheduling, etc. Another application of graph theory is for software engineering like software evolution process modeling, software evolution prediction, fault prediction, code refactoring, and malware detection.

This study focuses on using graph theory to analyze malware programs that will help in detecting malware attacks in a large software system. Relationships between nodes represent knowledge about the network. Analysis of these relationships tells us about the community structure of that software network. For example, on analyzing the source code

of a malware program, patterns of malicious behaviors and community structures of malicious codes can be detected. Malware families often have code similarities. Hence, information on community structures of malware source codes can be further used to detect malicious programs in the system.

2.1.1 Graph representation of a software

In order to analyze software networks, researchers have represented software programs in different types of graphs and performed various clustering algorithms. Dietrich, Yakovlev, McCartin, Jenson, and Duchrow (2008) analyzed the dependency graphs of Java programs using the Girvan-Newman clustering algorithm to compute the modular structure of the program.Pan, Li, Ma, Liu, and Qin (2009) implemented clustering algorithm on attribute-method network and method-method network of an OO Software JHotDraw 5.1 for code refactoring. Šubelj and Bajec (2011) analyze class dependency networks that provide significant community structure that matches with the original network structure. Extensive research has been done on analyzing software programs as a network of classes. However, little research has been reported on the analysis of a software program as a network of functions. This can be achieved by using call graphs. “A call graph consists of nodes, representing procedures, linked by directed edges, representing calls from one procedure to another.“ (Grove & Chambers, 2001, p. 689) .

This study focuses on analyzing the source codes of malware by representing them as a call graph and then implement further algorithms to discover patterns of malicious behavior.

2.2 Community Detection

One of the most common solutions for analyzing large graphs is community detection. It can be used to analyze growing networks to detect communities within the network and perform analysis at the community level instead of at a node level. A community is defined as a "division of network nodes into groups within which the network connections are dense, but between which they are sparser" (Newman & Girvan, 2004, p. 1). This feature of graphs is very widely used in scenarios where networks (systems) can be represented as graphs.

Discovery of communities within a graph has gained popularity for various reasons. Networks like social interaction networks, cyber networks, software networks, etc. are too big. Hence computations at each node are time and resource consuming. Communities divide the graph into multiple independent subgraphs. Hence the further analysis of the network can be done at the community level that is easier and faster. Community detection also helps in visualizing a dense network. Good visualization of a network makes analysis efficient and significantly easier (Shanbhaq, 2016). Community detection is also known as clustering. Community detection algorithms divide the network into independent sub-graphs, and these subgraphs are then replaced with a meta node. In this way, a large graph is reduced to a coarse graph by replacing all subgraphs with their corresponding meta nodes (Huang & Huang, 2015). A graph with a smaller number of nodes that represents the original graph is visually more understandable. Moreover, "the ability to find and analyze such groups can provide invaluable help in understanding and visualizing the structure of the network" (Newman & Girvan, 2004, p. 1). Researchers have revealed that just like other complex networks, networks constructed from a software exhibit small world properties (Myers, 2003; Qu, Guan, Zheng, Liu, Wang, et al., 2015; Valverde & Solé, 2003). The results of Myers (2003); Qu, Guan, Zheng, Liu, Wang, et al. (2015), and Valverde and Solé (2003) show that software systems also exhibit typical properties of complex network systems and thus motivate the use of community detection that was originally developed for complex networks for analyzing software systems.

Consider a subnetwork B of a large network and let k_i^{int} denote the internal degree of node i. The internal degree of a node is the total number of links that connect to node i in subnetwork B. Similarly, let k_i^{ext} denote the external degree of node i. The external degree of a node is the total number of links that (do not belong to the subnetwork B) connect to node i. A community C can be said as a strong community if it satisfies Equation 1. A community is said to be weak if it satisfies Equation 2.2, that is, the sum of the internal degree of all the nodes in community C exceeds the sum of the external degree of all the nodes present in community C (Barabási, 2016).

$$k_i^{int}(C) > k_i^{ext}(C) \tag{2.1}$$

$$\sum_{i \in C} k_i^{int}(C) > \sum_{i \in C} k_i^{ext}(C) \tag{2.2}$$

Thus, detecting and characterizing such community structures is called as community detection (Chen, Kuzmin & Szymanski, 2014).

2.3 Community Detection Algorithms in Call Graphs

Detection of community structures is considered to be a technique of data analysis to explore characteristics of structure and behavior of a network. Analyzing a coarse graph is much faster than analyzing a graph which is made of tens of thousands of nodes and edges. This section sheds some light on various graph metrics developed for quantifying the quality of the communities detected and also talks about various approaches taken for community detection by the researchers.

2.3.1 Graph metrics

Once the network is divided into communities we need to check the quality of that community in order to decide whether a particular community partition is better than some other one. Various metrics have been used by researchers to assess the quality of the partition. This section provides a brief description of various metrics that can be used to quantify the quality of the communities detected.

2.3.1.1 Modularity Newman and Girvan (2004) first introduced the concept of evaluating the quality of the communities detected. They coined the term modularity that measures the goodness of the partitioned network. Fortunato (2010) refers to this quality function as Q. Q can be calculated using Equation 2.3 Newman and Girvan (2004)

$$Q = (\frac{1}{2m}) \sum_{i,j} (A_{i,j} - P_{i,j}) \delta(C_i, C_j) \tag{2.3}$$

where,

m: total number of edges in the network

$A_{i,j}$: Adjacency matrix list of the network

$P_{i,j}$: Expected number of links between i and j if the network is randomly wired.

δ is an indicator function that yields 1 if vertices belong to the same community and otherwise it yields 0.

The value of Q ranges from -1 to 1. Larger the values of Q indicate stronger community structure. According to Newman and Girvan (2004), for good community structures of network the value of Q lies in the range 0.3 to 0.7.

Apart from modularity, Leskovec, Lang, and Mahoney (2010), gives a list of criteria which can be used to quantify the quality of community. The researchers have categorized criteria into multi criterion and single-level criterion scores.

- Multi-Criterion: Internal density, conductance, cut ratio, normalized cut, maximum out degree fraction, and average out degree fraction.

- Single-Criterion: volume, modularity ratio, and edges cut.

2.3.1.2 Class cohesion In software engineering the most widely and traditionally used quality metric is class cohesion (Qu, Guan, Zheng, Liu, Wang, et al., 2015). Classes are a basic component of an OO program. Class cohesion is a key attribute that is used to assess the quality of the classes and represents to what extent the class and its attributes are related. A class with high cohesion indicates that it is understandable, maintainable and reusable. There are various categories of class cohesion metrics. The usage of the metric depends on the context, such as what type of interactions are considered, the development phase during which they are applicable, and also the types of methods considered (Al Dallal, 2012). For example the metrics: Cohesion Among Methods in a Class (CAMC), Normalized Hamming Distance (NHD), and Method-Method through Attributes Cohesion(MMAC) are considered for the high-level design phase. Whereas, the metrics: The Lack of cohesion in Methods1 (LCOM1), LCOM2, Loose Class Cohesion (LCC), and Tight ClassCohesion (TCC) are based on counting the number of method pairs that share common attributes or do not share common attributes.

2.3.2 Algorithms

Research over community structures in networks has a long and rich history. This section talks about various approaches taken by the researchers for community detection in networks.

One of the techniques for community detection is graph partitioning. Graph partitioning is a process in which a graph is divided into groups of a predetermined size such that the edges in that network are minimized (Fortunato, 2010). A variant of the graph partitioning algorithm is Graph Bisection (Boppana, 1987). Graph Bisection partitions the network into two subgraphs such that the number of edges between the two subgraphs is minimized. As the size and number of clusters are predetermined, graph partitioning is not a suitable method in scenarios where both the parameters (size and number) are unknown. Moreover, according to Fortunato (2010) , the algorithm must be

able to reveal information about the structure of the network instead of asking the information as prior knowledge. To overcome the above limitation, researchers developed another method of community detection called hierarchical clustering (Fortunato, 2010). This type of algorithm finds clusters with high similarity in a network. Hierarchical clustering generates a hierarchy of several clusters at each level. A metric for measuring the similarity or dissimilarity between the clusters needs to be determined to carry out hierarchical clustering. There are two approaches for hierarchical clustering (Afsariardchi, 2012; Barabási, 2016):

- Agglomerative (bottom up) approach: initially all the nodes are considered as a single cluster. The clusters are then merged recursively into same community if there exist high similarity.
- Divisive (top down) approach: initially the whole network is considered as one whole community. The communities are then isolated by removing dissimilar nodes within a particular community.

Some of the networks have natural similarity indexes, but in most of the networks, the similarity indexes are chosen according to their suitability for example: correlation coefficients, matrix methods, path lengths, etc. But one concern of agglomerative methods is that sometimes they fail to find correct communities when the community structure is known and also these methods detect only the cores of communities and ignore the peripheries (Newman & Girvan, 2004). Due to the limitations in agglomerative method, Newman and Girvan (2004) used divisive methods to detect community resulting in reliable and sensitive community detection from artificially generated networks.

2.4 Louvain Algorithm

The Louvain algorithm was developed by Blondel, Guillaume, Lambiotte, and Lefebvre (2008). This algorithm is a greedy algorithm that focuses on modularity optimization. Louvain algorithm is an agglomerative clustering algorithm with each node as a single separate community. It works in two phases that are repeated iteratively until the only node is left or the modularity cannot be optimized further. The two phases are as follows:

1. Each node is assigned to its own community. Modularity gain is calculated for movement of the node to each of its adjacent neighbor. Then the decision of the movement of the node to its adjacent neighbor is based on the highest modularity gain. The modularity gain is computed from Equation 2.4 (Blondel et al., 2008).

$$\Delta Q = [\frac{\sum_{in}+2k_{i,in}}{2m} - (\frac{\sum_{in}+2k_{i,in}}{2m})^2] - [\frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2 - (\frac{k_i}{2m})^2] \qquad (2.4)$$

   where,

   $\sum_{in}$: indicates the summation of weight of edges within a community,

   k_i: is total weight of edges incident to i,

   $\sum_{tot}$ indicates total weight of edges incident to all the nodes in the community,

   m: is the sum of weights of all the edges in the network, and

   $k_{i,in}$: addition of the weights of edges from node i to all nodes in community.

   The node i is removed from its own community and placed in the adjacent community with the highest modularity gain. If there is no change in the modularity, i remains in its own community.

2. Each of the communities detected in phase 1 will be replaced by a meta node thus reducing the size of the graph and forming a new network. The weights between the newly formed nodes are calculated by adding up the weights of the edges between the nodes of the corresponding community.

The run time complexity of the Louvain algorithm is *O(nlogn)*. This algorithm is fast as the number of communities to consider reduces after the first few passes. However, the researchers of this algorithm state that the speed of the algorithm can be optimized by adding or modifying simple heuristics of the algorithm.

2.5 Modifications on Louvain Algorithm

In the past few years, researchers have made several efforts in parallelizing and improvising the Louvain algorithm. Bhowmick and Srinivasan (2013) attempted to parallelize the Louvain algorithm using shared memory. This approach focuses on evaluating the vertices in parallel and hence updates the community structures on the fly. Another effort to parallelize the Louvain algorithm is by using distributed memory (Wickramaarachchi, Frincu, Small, & Prasanna, 2014). This approach apriori partitions the input graph using a graph partitioner, runs the sequential algorithm on each partition separately, and then merges the results by using an aggregation process. Another parallel effort called PLM was conducted by Staudt and Meyerhenke (2013) that uses label propagation to parallelize the Louvain algorithm. Lu, Halappanavar, and Kalyanaraman (2015) parallelized the Louvain algorithm using parallelization heuristics like graph coloring for fast community detection. The parallel algorithm is called Grappolo. Grappolo provides higher modularity results than PLM (Lu et al., 2015).

2.5.1 Grappolo

In spite of the increasing popularity for high modularity communities and fast and memory efficient community detection, the Louvain algorithm is sequential and thus limits its scalability. Lu et al. (2015) investigated the following challenges of parallelizing the sequential nature of the Louvain algorithm and proposed parallel heuristics for parallelizing the Louvain algorithm.

1. Vertex following: In many real-world graphs, there exist a large number of nodes with a single degree, i.e., node with only one neighbor. For such nodes, it is unnecessary to explicitly spend resources to make the decision about the community transfer as it is by default going to join its only neighbor. Hence, such nodes can be preprocessed such that they are merged with their neighbors. This preprocessing helps in reducing the number of nodes to be considered during each iteration.

2. Minimum label heuristics: The communities are assigned a numeric label in an arbitrary order. At any given iteration, node *i* will be having multiple neighboring communities that yield same maximum modularity gain, and hence in such cases, the community with the minimum label will be selected as the destination community ofnode *i*

3. Graph Coloring: In this parallel algorithm, distance-1 coloring is used to address parallelization challenges. The distance-1 coloring of a graph assigns colors to nodes such that no two adjacent nodes have the same color. Using this heuristic, all the nodes are partitioned into the same color sets and are processed in parallel with the guarantee that no two adjacent nodes will be processed concurrently.

Just like the Louvain algorithm, the phases are executed one at a time. Within each phase, there are multiple iterations. Every iteration executes a parallel process on vertices with the same color and using the information from the previous iteration. This phase is executed until the modularity gain is negligible between the vertices. After the execution of a phase, the community assignment output graph is then modified by representing each community by a meta-node. This modified graph is input to the next phase.

This parallel algorithm was implemented in C++ using OpenMP. This algorithm was implemented on 11 real-world networks in diverse areas like social networks, biological networks, and scientific computing resulting into equivalent modularity communities as compared to the serial Louvain algorithm. Along with higher modularity communities, it proved to be able to produce stable and consistent communities with measurable speedups (*Advanced Computing, Mathematics and Data Research Highlights*, n.d.).

2.6 Summary

To summarize, the rate of malware attacks is increasing day by day. Hence it is the need of the hour to secure the systems against such attacks. One of the solutions is to use community detection to analyze malware. By analyzing malware programs, a community structure or pattern can be discovered that can be used to further detect malware in large software systems. Once a malware enters into the system, it propagates very quickly hence fast unfolding of communities is required. Research shows that there have been successful efforts to parallelize and improve the speed of the Louvain method: a community detection algorithm. These efforts have not been tested on software networks. Hence this study focuses on understanding one of the parallel versions of the Louvain: The Grappolo algorithm thoroughly to add/modify heuristics to improvise the speed of the algorithm.

CHAPTER 3. FRAMEWORK AND METHODOLOGY

This chapter gives details about the overall research framework and the proposed methodology used in this study. This includes few details on experimental setup, variables, population, and modifications that were done to the Grappolo algorithm to make it faster.

3.1 Research Framework

This research is a study based on the fast unfolding of communities in large software networks for malware detection applications. Detecting communities within large software networks help to obtain multiple levels of granularity that in turn makes the identification of malicious software easier. This study focuses on answering the following research question: Does the proposed modified Louvain algorithm perform better than the existing parallel version of the Louvain algorithm preserving the quality of the communities detected by the algorithm for malware detection in an Android system?

3.2 Modifications to the Louvain algorithm

Section 2.4 gives a detailed explanation of the Louvain algorithm. To summarize the Louvain algorithm: it is modularity optimization algorithm based on local information and is best suited for analyzing large networks. It consists of two phases:

1. Each node is assigned to its own community. For every node, the modularity gain (ΔQ) of transferring it to its neighboring community is calculated and then the node is moved to the community which results in higher modularity gain. This step is executed repeatedly and sequentially for each node until there is no further improvement in modularity.

2. A new representation of the network is generated by replacing the communities with a meta-node.

The above steps are then executed repeatedly until stable communities are detected.

According to Shanbhaq (2016), 40% of the time taken by the algorithm was spent in removing the node, placing it in neighbor's community, and calculating modularity gain for each and every neighboring community. Hence, in order to reduce the time spent in deciding which neighboring community to chose, a heuristics can be added. The process of calculating ΔQ is sequential, to reduce the time spent, this process can be parallelized in the following way:

1. At the initial stage, all the communities are assigned a numeric label in an arbitary order.

2. For every node *i*, there are two groups. One group has neighbors with an even label and the other group has neighbors with an odd label. The process of calculating ΔQ for both the groups with respect to node *i* is parallely.

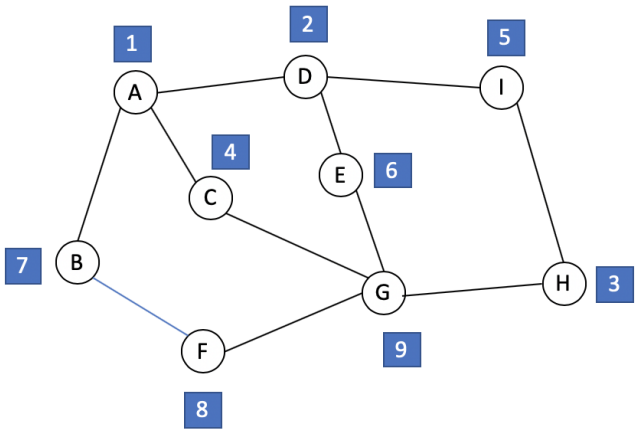

Figure 3.1. Graph representation

For example, in the Figure 3.1, the numbers in the squares represent the label of each community. For node A, the two neighbor groups are:

Even label group: node D and node C

Odd label group: node B

The process of calculating modularity gain of node A with even label group and odd label group is done parallelly. However, the modularity calculation within the group is sequential. For example, ΔQ calculation of node A with respect to node D and node C is sequential.

3. The node that yields highest ΔQ within each group is compared with each other and the larger ΔQ node is selected. For example if,

$$\Delta Q_{A \to D} > \Delta Q_{A \to C}$$

then ΔQ of node D is compared with ΔQ of node B. If,

$$\Delta Q_{A \to D} > \Delta Q_{A \to B}$$

Then, node A is transferred to node D.

3.3 Hypothesis

Based on the above research question, the study focuses on finding a conclusion for the following hypothesis:

H_0: The run time of proposed parallelized version of the Grappolo algorithm has improved than the original Grappolo algorithm.

H_0: $\mu_1 \leq \mu_2$

H_α: The run time of proposed parallelized version of the Grappolo algorithm has not improved than the Grappolo algorithm.

H_α: $\mu_1 > \mu_2$

μ_1 is run time of modified Grappolo algorithm

μ_2 is run time of original Grappolo algorithm

3.4 Research Type

This is a quantitative study with the aim to study the Louvain and parallelized version of the Louvain (Grappolo) algorithm thoroughly and investigate if there are any ways by which the speed of the algorithm can be improved while preserving the quality of the algorithm at the same time. The results are based on the statistical analysis of the running time of the original Grappolo algorithm and running time of proposed version of the algorithm.

3.5 General Methodology

This section talks about the methodology used to address the research question. Figure 3.1 represents the overall procedure of community detection on an APK of an application and the manner in which the community structures can be used for malware detection.

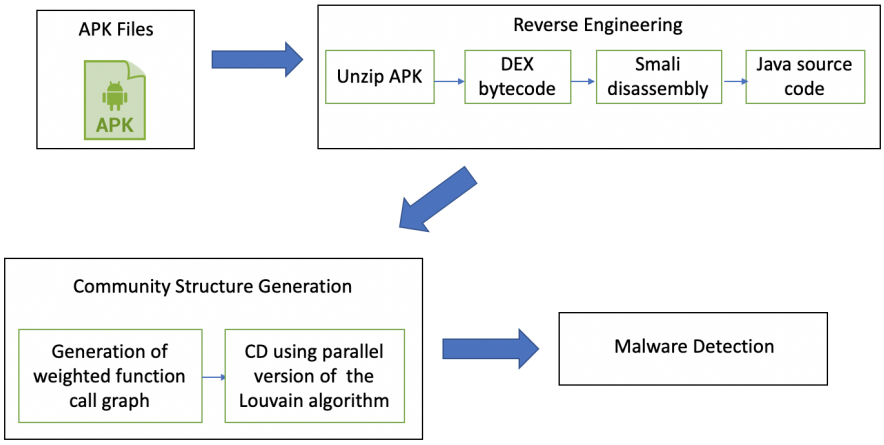

Figure 3.2. Workflow

3.6 Variables

This section talks about independent and dependent variables used in this study. The results of this study are based on the performance of the modified Grappolo algorithm in terms of the running time. The running time of an algorithm varies with the size of the input. In this study, the input is a software network represented as a graph. Hence the running time varies with the size of the network, assuming that the hardware setup is constant. The size of the network can be varied by changing the number of edges and nodes. Independent variables used in this study are as follows:

1. Number of nodes: Total number of unique nodes/vertices present in the graph.
2. Number of edges: Total number of unique links/edges present in the graph.

Dependent variables whose values are calculated and monitored in this study are as follows:

1. Running time: The values of this variable are observed for different input sizes. Also, the running time of Grappolo and modified Grappolo algorithm is compared.
2. Modularity: The modularity values of the communities detected by the modified Grappolo algorithm for different inputs are observed and compared with the Grappolo algorithm.

3.7 Experimental Setup

The original Grappolo and the modified Grappolo algorithm is implemented on common hardware setup. The study uses the Rice community cluster at Purdue University provided by Information Technology at Purdue (ITaP) Research Computing (RCAC). The hardware specifications of Rice are available online on the following link https://www.rcac.purdue.edu/compute/rice

CHAPTER 4. EXPERIMENTAL EVALUATION AND RESULTS

This chapter provides description of the general workflow of the experiment and analysis of the observed results.

4.1 Data Source

The data that was used in this study is a subset of the database of malware samples collected from The Drebin Dataset (Arp, n.d.). The Drebin database consists of 5,560 Android applications representing 179 different malware families. The samples were collected from August 2010 to October 2012. For this research 1002 applications are considered. After analyzing the characteristics (number of nodes, number of edges) of these applications, 20 malware applications were used to test the execution times of the modified and parallelized version of the Grappolo algorithm.

4.2 Data Prepossessing

A call graph is generated by extracting the method calls within an application. The dataset used in this study was in the form of APKs. In order to extract function calls from an APK, first, the APK should be decompiled to get a smali code. A smali code is a human-readable assembler/disassembler code. This code can be analyzed to identify the method calls and thus generate a call graph of the code. To achieve this, the researcher has used a tool called Androguard. Androguard is a reverse engineering tool written in Python. This tool is to analyse Dex/Odex, APKs, Android's binary XML files (*Getting Started androguard 3.3.5*, n.d.).

Following command is used to generate a call graph of an Android application using Androguard.

```
Poojas-MacBook-Pro:CommunityDetection poojapatil$ androguard cg hello-world.apk -o helloworld.gml
[INFO    ] androguard.analysis: End of creating cross references (XREF)
[INFO    ] androguard.analysis: run time: 0min 01s
```

Figure 4.1. Command to generate a call graph from an APK

The generated call graph is unweighted directed graph in Graph Modeling Language (GML) format. To make the graph format compatible with Grappolo, the GML formatted graph was converted into a Pajek file format using Gephi. Pajek files are simple text files where each line is a single element. The first line indicates the number of vertices represented as **Vertices N* where *N* is the number of vertices. This is followed by list of all the vertices present in the graph in turn followed by a list of edges. Figure 4.2 shows a sample Pajek file with 7 vertices.

```
*Vertices 7
1 "A"
2 "B"
3 "C"
4 "D"
5 "E"
6 "F"
7 "G"
*Edges
1 2
2 3
1 4
1 5
3 5
4 7
5 6
6 7
```

Figure 4.2. Sample pajek graph file

Gephi is a graph and network visualisation and exploration software (*The Open Graph Viz Platform*, n.d.). Figure 4.3 shows the interface of Gephi

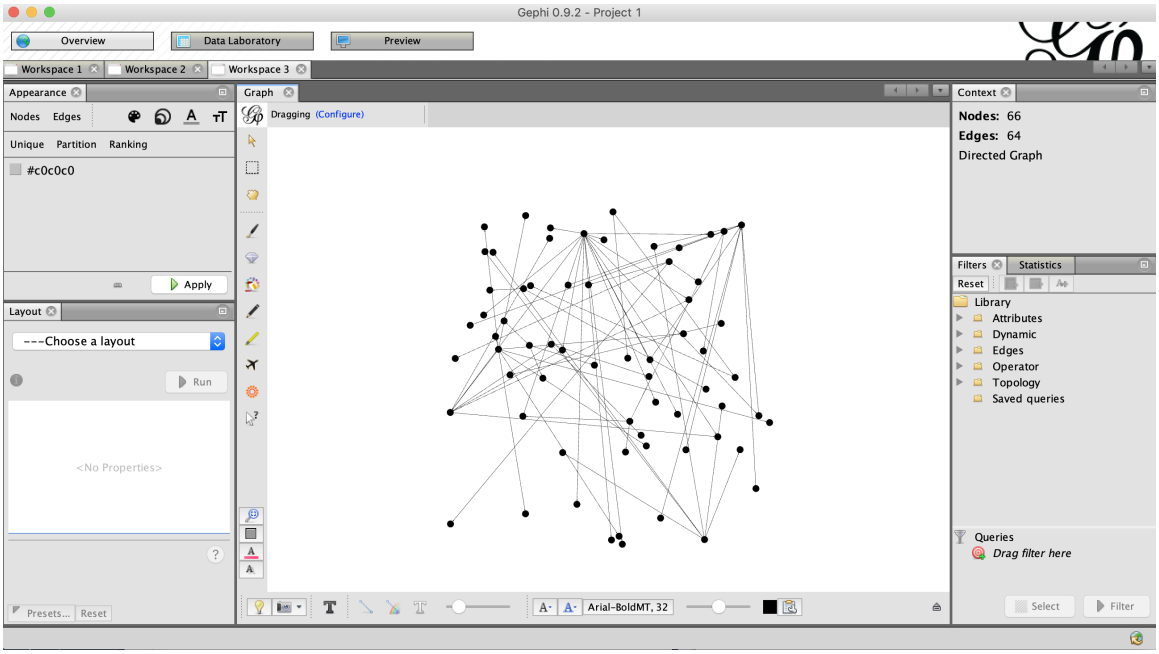

Figure 4.3. User interface of Gephi

4.3 Implementation Details

The Grappolo algorithm was implemented in C++ and OpenMP. The implementation uses C++ STL map data structure to store information about the clusters and the neighbors of the corresponding cluster. The implementation makes use of pointers to memory to store information about community assignments for each and every vertex.

Parallelism can introduce problems like data race, deadlock, etc. To avoid such problems, constructs like locks and atomic operations were used. In a parallel region if multiple threads are trying to write to the same variable, then it may result in garbage values. Hence one should synchronize the access to such variables. One way to do this is by using atomic operations. Atomic operations are executed by one thread at once. For example,

The above example is how one will increment a variable in a parallel context. If there are greater than 2 threads, only one thread will be able to satisfy the if condition. Whenever thread executes this atomic operation, y gets the previously stored value. The disadvantage of using atomic operations is that it performs limited operations like addition

```
int x=0;
#omp parallel
y=__sync__fetch_and_add(*x,1);
if(y==0)
{
    printf("I am first thread to increment x\n");
}
```

Figure 4.4. Example of atomic operation

and subtraction that are not enough to synthesize complex operations. To update the source and target communities, Grappolo uses atomic operations __sync_fetch_and_add() and __sync_fetch_and_sub(). The above functions return value of the variable already stored in the memory and then updates the variable. Figure 4.5 shows the algorithm of the existing parallel version of the Louvain algorithm (Grappolo).

4.3.1 Modifications on Grappolo

This research attempts to parallelize Grappolo. In the Grappolo algorithm, the process of calculating modularity gain of transferring each community to its neighboring community is done sequentially. In this research, the researcher has attempted to parallelize this task by using nested parallelism. The researcher has modified the execution of lines 12-14 from Algorithm 1. The algorithm of modified section of Grappolo is as follows:

The idea behind the modification is such that instead of sequentially calculating modularity gain for each and every neighbor, divide the neighbors into two groups and then simultaneously calculate the modularity gain for both the groups. The division of the neighboring communities into two groups is done on the lines 13-15. Then the calculation of modularity gain for each neighboring community in both the groups is done simultaneously (lines 16-24). This is achieved by using nested parallelism.

4.3.1.1 Nested parallelism For nested parallelism, OpenMP makes use of the fork-join

Algorithm 1 The parallel Louvain algorithm (a single phase).

1: **procedure** PARALLEL LOUVAIN($G(V, E, \omega), C$)
2: **for each** $i \in V$ in `parallel` **do**
3: $C(i) \leftarrow \{i\}$; $\ell(C(i)) \leftarrow i$
4: $C^i_{int} \leftarrow 0$ ▷ counter for the #intra-community edges due to i
5: **for each** $j \in \Gamma(i)$ **do** $C^i_{tot} \leftarrow C^i_{tot} + \omega(i, j)$
6: $Q_C \leftarrow 0$; $Q_P \leftarrow -\infty$ ▷ Current & previous modularity
7: **while** `true` **do** ▷ Iterate until modularity gain becomes negligible.
8: **for each** $i \in V$ in `parallel` **do**
9: $C_{old} \leftarrow C(i)$; $N_i \leftarrow C(i)$
10: **for each** $j \in \Gamma(i)$ **do** $N_i \leftarrow N_i \cup C_j$
11: $maxGain \leftarrow 0$; $C_{new} \leftarrow C_{old}$
12: **for each** $c \in N_i$ in `parallel` **do**
13: $curGain \leftarrow$ Calculate $\Delta Q_{i \to c}$
14: **if** $((curGain > maxGain)$ or $(curGain = maxGain$ and $\ell(c) < \ell(C_{new}))$ **then** ▷ Minimum label heuristic
15: $maxGain \leftarrow curGain$; $C_{new} \leftarrow c$
16: **if** $maxGain > 0$ **then**
17: $C_{old} \leftarrow C_{old} \setminus \{i\}$; $C_{new} \leftarrow C_{new} \cup \{i\}$
18: **for each** $c \in C$ `AND` $c \neq \emptyset$ in `parallel` **do**
19: $C^c_{int} \leftarrow 0$; $C^c_{tot} \leftarrow 0$
20: **for each** $(i, j) \in E$ in `parallel` **do**
21: **if** $C(i) = C(j)$ **then**
22: $C^i_{int} = C^i_{int} + \omega(i, j)$
23: $C^i_{tot} = C^i_{tot} + \omega(i, j)$
24: **else**
25: $C^i_{tot} = C^i_{tot} + \omega(i, j)$
26: $C^j_{tot} = C^j_{tot} + \omega(i, j)$
27: $e_{xx} \leftarrow 0$
28: $a^2_x \leftarrow 0$
29: **for each** $c \in C$ `AND` $c \neq \emptyset$ in `parallel` **do**
30: $e_{xx} += C^c_{int}$; $a^2_x += (C^c_{tot})^2$
31: $Q_C = \frac{e_{xx}}{m} - \frac{a^2_x}{(2m)^2}$
32: **if** $|\frac{Q_C - Q_P}{Q_P}| < \theta$ **then** ▷ θ is a user specified threshold.
33: `break` ▷ Phase termination
34: **else**
35: $Q_P \leftarrow Q_C$

Figure 4.5. Algorithm of Grappolo

model. Whenever a thread encounters a parallel region, it creates a team of threads including itself. The thread encountering the parallel construct is a master thread whereas other threads are slave threads. All the slave threads execute the code in the parallel region. After a thread finishes executing the code within the parallel region it waits for the implicit barrier, i.e., wait for the rest of the slave threads to execute the parallel code. Once all the threads complete execution of the code, the slave threads leave the barrier.

```
12. For each community Ci in parallel do
13.     For each neighboring community Ni
14.         Ni← L (where L is a random numeric value)
15.     Based on the label divide the Ni into two groups: Odd No and Even Ne
16.     Execute the following two block of codes parallely
17.         Block1: For each Ci in Ne
18.             curGaine ← Calculate ΔQi←c
19.             if (curGaine >maxGaine) then
20.                 maxGaine > curGaine; Ce← c
21.         Block2: For each Ci in No
22.             curGaino ← Calculate ΔQi←c
23.             if (curGaino > maxGaino)
24.                 maxGaino > curGaino; Co← c
25.         At implicit barrier
26.         if (maxGaine > maxGaino) then
27.             Cnew ← Ce
28.         Else
29.             Cnew ← Co
```

Figure 4.6. Algorithm of the modified Grappolo

The master thread continues its execution of the code while slave threads wait to join another team.Nested parallelism can be enabled by setting omp_set_nested() to True. If it is disabled, the parallel region will be executed just by the thread that encountered the parallel region.

The modified version of Grappolo implements two levels of parallelism. The outer loop is a parallel for loop. A subset of code inside the parallel for loop is parallelized using OpenMP tasks.

The code written in the task construct is wrapped as a block of work and is made available to threads to be executed parallelly. The execution flow is as follows:

1. On entering a parallel region, a team of threads will be created.
2. At a time a single thread creates tasks and adds them to the queue.
3. Depending on the task scheduler, the tasks are executed by the team of threads.

A point to note is that a thread that executes a task can be different from the thread that encountered it. Figure 4.7 gives an example of a task construct

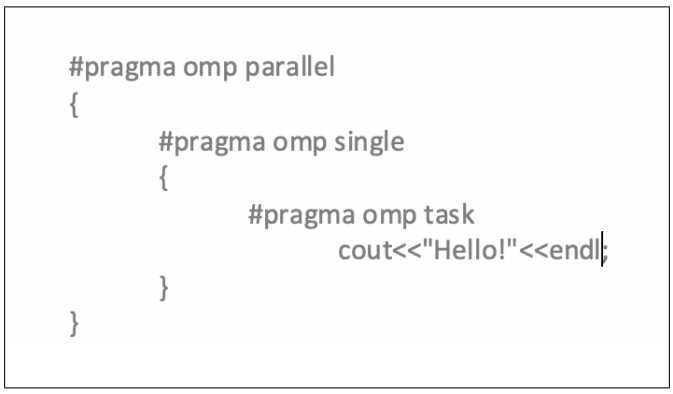

Figure 4.7. An example of task construct

In the above example, at an instance, a single thread will create tasks and add them to the queue, and these tasks will be executed by a team of threads. In the modified algorithm of Grappolo, the lines 17-20 and lines 21-24 are defined as two different tasks. Hence on encountering a parallel construct, the master thread creates two tasks and adds them to the queue to be executed by slave threads. To control in what order the tasks are executed by threads, synchronization constructs can be used. For task synchronization, *taskwait* construct is used to make sure that all the tasks have completed their execution. Lines 25-29 of the modified algorithm compare the maximum modularity gain from both the groups, and the community with the highest modularity gain from either of the group is chosen for the community assignment. This block of operation is executed in the taskwait construct. The reason for doing so is that until both the tasks are not completed, the maximum modularity gain of both the groups won't be available. Hence the taskwait construct is executed only when the two tasks are completed, and we have the values of maximum modularity gain from both the groups to compare.

4.4 Results and Analysis

In order to conclude the hypothesis mentioned in the section 3.3, the two sample t-test is used. The t test is used to check if there is a significant difference between two groups. The two samples collected for this test are independent of each other, i.e., they are unpaired. Two sample t test tests the difference between the two population means. This test is used when the standard deviation of the population is unknown. It calculates confidence interval and performs hypothesis test of the difference between two population means. The researcher has considered the level of significance (α) as 0.5 for this research. α is the measure of the strength of evidence that must be present in your sample in order to reject the null hypothesis. If the p-value is less than the level of significance (0.5) then the result is statistically significant and therefore can reject the null hypothesis. In other words, the sample has strong enough evidence to reject null hypothesis at the population level. This test was conducted on multiple variations of the samples by varying the number of edges and nodes of the graph.

4.4.1 Input variaions

Table 4.1 shows variations in the input formed by varying the number of nodes and number of edges.

Table 4.1. Variations of input

Variable	Size					
Number of nodes	12K	10K	8K	6K	4K	2K
Number of edges	100K	70K	50K	30K	24K	18K
	12K	6K	3K	1.5K	725	

Another variation of input is based on the relation between the number of nodes and the number of edges. The variations are as follows:

1. N(Edges) $>$ 2*N(Nodes)
2. N(Edges) = *N(Nodes)

3. N(Edges) < N(Nodes)

From the dataset, for each variation of input one APK was chosen that has the number of nodes/edges in the range specified in the “Size“ column of the Table 4.1. Hence in total 15 APKs were chosen from the dataset for this test.

4.4.2 Results of varying the number of nodes

The original and the modified Grapppolo was executed for 100 iterations on 6 different graphs that had varying number of nodes. Two sample t-test was implemented on each graph, where the two samples consisted 100 execution times of the original and modified Grappolo algorithms for that particular graph. The following table shows average execution times and the p-values for the t test statistic,

Nodes	Average Modularity		Average Execution Time		p-value
	Original	Modified	Original	Modified	
12K	0.759351	0.7550306	0.02697339	0.043505386	0.0001
10K	0.7724074	0.773494	0.01294805	0.035787028	0.0001
8K	0.7528576	0.7536984	0.01294805	0.029676081	0.0001
6K	0.7692644	0.760741	0.00842692	0.020751936	0.0001
4K	0.765108	0.770747	0.00739345	0.016507934	0.0001
2K	0.691062	0.69152	0.00727503	0.008684314	0.0001

Figure 4.8. Results for varying the number of nodes

From the above table, it can be observed that the execution time for modified Grappolo is larger than the original Grappolo, it is almost double the original Grappolo execution time. This can be statistically proven by observing the p-value. The p-value for all the variations is 0.0001, which is less than significance level 0.05. With p-value < 0.05, we reject the null hypothesis at a 5% level of significance. We conclude that the modified Grappolo algorithm has a larger execution time as compared to the original Grappolo.

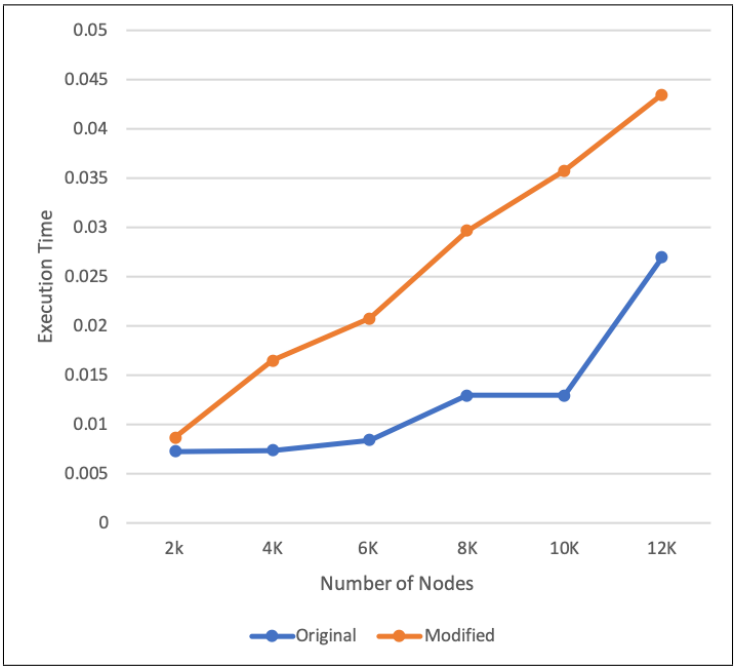

Figure 4.9. Lineplot comparing average execution times

It can be observed from the Figure 4.9 that, for a smaller number of nodes, the increase in the execution time is less, but as the size of the nodes increases, the execution time of modified Grappolo increases. A possible explanation for this is that as the size of the graph increases, the overhead caused due to initial setup (declaration and assignmnent of values to variables) in the modified algorithm also increases.

4.4.3 Results of varying the number of edges

The original and the modified Grapppolo was executed for 100 iterations on 6 different graphs that had a varying number of edges. The number of edges ranged from 24K to 750. A two sample t-test was implemented on each graph, where the two samples consisted of 100 execution times of the original and modified Grappolo algorithms for that particular graph. The following table shows average execution times and the p-values for t test statistic,

Nodes	Average Modularity		Average Execution Time		p-value
	Original	Modified	Original	Modified	
12K	0.759351	0.7550306	0.02697339	0.043505386	0.0001
10K	0.7724074	0.773494	0.01294805	0.035787028	0.0001
8K	0.7528576	0.7536984	0.01294805	0.029676081	0.0001
6K	0.7692644	0.760741	0.00842692	0.020751936	0.0001
4K	0.765108	0.770747	0.00739345	0.016507934	0.0001
2K	0.691062	0.69152	0.00727503	0.008684314	0.0001

Figure 4.10. Results for varying the number of nodes

From the above table, it can be observed that the execution time for the modified Grappolo is larger than the original Grappolo, it is almost double the original Grappolo execution time. This can be statistically proven by observing the p-value. The p-value for all the variations is 0.0001, which is less than significance level 0.05. With p-value < 0.05, we reject the null hypothesis at a 5% level of significance. We conclude that the modified Grappolo algorithm has larger execution time as compared to the original Grappolo.

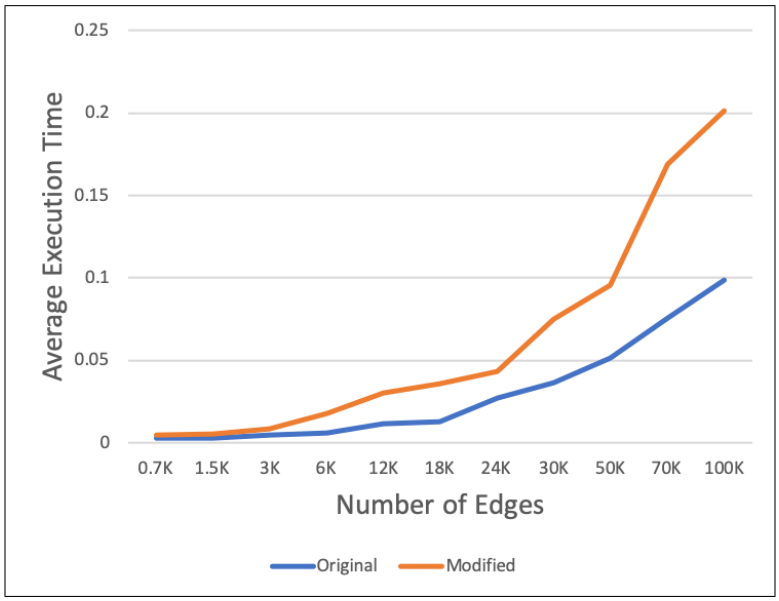

Figure 4.11. Lineplot comparing average execution times

It can be observed from the Figure 4.11 that for a smaller number of edges, the increase in the execution time is less, but as the size of the edges increases, the execution time of modified Grappolo increases.

4.4.4 Results for graph type 1: E<2V

The graph used in this test had 10,000 vertices and 24,000 edges. The implementation of the original Grappolo and the modified Grappolo on such graph did not complete. The programs aborted giving the error "Temporary buffer is not enough."

4.4.5 Results for graph type 2: E=V

The graph used in this test has 113 edges and 113 nodes. Table 4.2 reports the average execution time and modularity after running original and modified Grappolo for 100 iterations

Table 4.2. Running time and modularity for graph type:2

Algorithm	Grappolo	Modified Grappolo
Average execution time	0.000727503	0.00120508
Average modularity	0.754993	0.754993

After conducting the t-test on the above results, the calculated p-value was 0.0001. With p-value < 0.05, we reject null hypothesis at a 5% level of significance. We conclude that the modified Grappolo algorithm has a larger execution time as compared to the original Grappolo.

4.4.6 Results for graph type 3: E<V

The graph used in this test has 40 edges and 38 nodes. Table 4.3 reports the average execution time and modularity after running the original and modified Grappolo implementations for 100 iterations.

After conducting a t-test on the above results, the calculated p-value was 0.0001. With a p-value < 0.05, we reject the null hypothesis at a 5% level of significance. We conclude that the modified Grappolo algorithm has a larger execution time as compared to the original Grappolo.

Table 4.3. Running time and modularity for graph type:3

Algorithm	Grappolo	Modified Grappolo
Average execution time	0.00051085	0.00073881
Average modularity	0.621537	0.593837

4.5 Resultant communities

The communities detected by both the algorithms were analysed to check to what degree they differ from each other. As seen in the above results, the modularity of the communities detected by both the algorithms remains similar. However, the communities might differ because of the introduction of random numeric label in the minimum label heuristics of the Grappolo algorithm. Figures 4.12 and 4.13 show the communities detected for a graph with a number of nodes=155 and number of edges=260. The modularity of the communities detected by modified Grappolo algorithm was 0.653913, and the modularity of the communities detected by the original Grappolo algorith was 0.648203.

The communities detected are differ to some extent. The differing communities are communityID = 3,9, and 10. The communities 3 and 9 are neighboring communities as the nodes 35 and 47 are in community 3 for the original Grappolo algorithm, but for the modified algorithm, they are in community 9. Similarly, communities 9 and 10 are neighboring communities as many vertices that are in community 9 for the original algorithm are in community 10 for the modified algorithm. The reason behind these difference is the random label that is assigned to each community, and based on this label, the destination neighboring community is decided.

	CommunityID															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Vertex	1	3	4	6	20	21	25	27	36	37	48	61	98	99	103	104
	2	5	9	35			26		67	65	49	64				
	42	7	11	47					68	66	70	74				
	50	8	15	51					75	89	71	76				
	62	10	17	52					83	100	72	78				
	63	12	22	53					90		73	79				
	69	13	23	54					91		101	80				
	77	14	24	55					92			81				
	95	16	28	56					93			82				
	96	18	29	57					97			84				
	102	19	33	58								85				
		30	34	59								86				
		31	39									87				
		32	41									88				
		38										94				
		40														
		43														
		44														
		45														
		46														
		60														

Figure 4.12. Communities detected by Grappolo algorithm

	CommunityID															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Vertex	1	3	4	6	20	21	25	27	36	35	37	61	98	99	103	104
	2	5	9	51			26		67	47	65	64				
	42	7	11	52					68	48	66	74				
	50	8	15	53					75	49	89	76				
	62	10	17	54					83	70	100	78				
	63	12	22	55					90	71		79				
	69	13	23	56					91	72		80				
	77	14	24	57					92	73		81				
	95	16	28	58					93	101		82				
	96	18	29	59					97			84				
	102	19	33									85				
		30	34									86				
		31	39									87				
		32	41									88				
		38										94				
		40														
		43														
		44														
		45														
		46														
		60														

Figure 4.13. Communities detected by modified Grappolo algorithm

CHAPTER 5. CONCLUSION AND DISCUSSION

This chapter briefly explains the research conducted. It also concludes the results of the research and lastly includes relevant discussion and future direction.

5.1 Conclusion

The principal focus of this thesis was to modify the parallel version of the Louvain algorithm to make the analysis of software networks for malware detection faster. Communities detected within a software network can be further analyzed to discover complex modules or important nodes within the graph. Another application of community detection in the context of software networks is influence maximization where the communities will be analyzed to find the most influential node. Identifying such influential nodes can take us one step towards accurately detecting malware as the hackers/adversaries will try to insert their malicious code in such nodes. Having information about the nodes in the network can further help in detecting or analyzing malware easily.

The literature states that Louvain algorithm is one of the faster algorithms and is able to handle large networks preserving the quality of the communities detected. A plethora of work has been done on making the Louvain algorithm faster. The researchers of Lu et al. (2015) have parallelized the Louvain algorithm. This thesis focuses on making the parallel implementation of the Louvain named Grappolo, faster. However, the resulting execution times of the modified Grappolo were observed to be greater than the original Grappolo. Chapter 4 gives details on the results obtained by executing the original Grappolo algorithm and the modified Grappolo on a set of inputs. The modularity was also observed to check the quality of the partitions. It was observed that the quality was preserved after making the modifications. Thus concluding that the added heuristics failed to reduce the execution time of the Grappolo but the quality of the communities detected by both the algorithms was similar.

5.2 Discussion

On thinking about parallel computing, it is important to acknowledge the following complications that occur due to parallel processing (Barney et al., 2010)

1. There does not exist a commonly agreed model of the parallel environment. An algorithm based on some model of parallelism will not necessarily have the same performance on two different architectures.
2. A better model should take into consideration the number of processors used as well as the run time. The tradeoffs need significant consideration as it may affect the performance of the algorithm.

Parallel overhead includes factors like (Mad, 2009):

1. Thread library startup overhead: This is a one time overhead when the code starts.
2. Thread start-up overhead: It is the time taken to create threads.
3. Per-thread overhead: Time spent by the threading library in scheduling chunks of work on each thread.
4. Synchronization: This includes time spent in controlling the concurrent execution of threads.
5. Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.

5.2.1 Speculation of results

After analysing the results obtained, few speculations about the increased execution time are as follows:

1. One of the challenges in parallel processing is deciding the granularity of task decomposition to get the best performance of the algorithm. In the modified code, the amount of speedup by the introduction of tasks depends on the ability of the max() function to speed up. However, it can be observed from the execution times of both the algorithms that in the implementation of the modified algorithm, time is spent in setting up the environment. One of the reason can be the overhead of creating a task is more than the work done by the task. In the modified implementation, the tasks are spawned for a small size data. To elaborate, the operations defined in the task construct work on calculating modularity gain for “n“ neighbors. After analyzing the input graphs, the maximum degree of a node is 100, i.e., that particular node has 100 neighbors. On randomly dividing the neighbors into two groups (odd and even) the idle number of vertices in one group will be 50, and the best case it will be 100. In both cases, it is small as compared to the size of the graph. Hence the tasks are spawned to calculate modularity gain for small “n“ (number of vertices) which is not justifying the overhead caused to create a thread.

2. Barrier: The performance of a parallel program greatly relies on the underlying synchronization mechanisms used for concurrency control. In the tasking model, task generation and execution are separate. Tasks execute at a task scheduling point. Also, threads may switch from one task to another. Hence, the synchronization of thread execution is required. In the implementation of the modified algorithm, a barrier: taskwait is used to synchronize the execution of tasks. This barrier can contribute to increased execution time because the master thread has to wait until all its child tasks have executed the tasks assigned to it.

5.2.2 Proposed changes based on speculations

An attempt to improve the execution time can be achieved by following proposed changes:

1. Have large parallel regions as they offer more scope for the threads to use the data available in cache and better compiler optimization.

2. In order to implement the modifications of the algorithm, the researcher added/modified the existing Grappolo code. Due to this, there were some implementation restrictions as the existing Grappolo code uses many data structures store the graph information, calculate the modularity gain, store cluster information, to store information about updated clusters, etc. All these data variables are interlinked. Hence to change one of them needs changes done to all the linked data variables. This posed a restriction as the researcher cannot change the majority of the logic of the code in terms of data variables. For example, to divide the variable that stores all the neighbors of a node, the researcher had to divide the neighbors and store in two different variables which can lead to unnecessary resource utilization. To reduce the execution time, the code can be modified to accommodate the changes like avoiding the use of barriers, use of data variables like multidimensional arrays to stores neighbors, and a label for each neighbor that signifies whether that neighbor belongs to the odd group or the even group. This will avoid the need for storing the neighbors in two different data variables.

5.3 Future Scope

A future direction for this study is to investiage deeper into why proposed heuristic failed to speedup the algorithm. The proposed heuristic resulted into larger execution times due to the overhead caused during the initial setup. Alternative constructs in OpenMP that would avoid such overheads can be investigated. Another direction can be to identify alternate heuristics to speedup the algorithm. This research majorly focuses on the execution time. A study can also be conducted to identify heuristics that improves the quality of the communities detected by keeping similar execution times.

REFERENCES

Adoption rate and popularity. (n.d.). Retrieved from https://usa.kaspersky.com/resource-center/threats/malware-popularity

Advanced computing, mathematics and data research highlights. (n.d.). Retrieved from https://www.pnnl.gov/science/highlights/highlight.asp?id=3887

Afsariardchi, N. (2012). *Community detection in dynamic networks*. Unpublished doctoral dissertation, McGill University.

Al Dallal, J. (2012). The impact of accounting for special methods in the measurement of object-oriented class cohesion on refactoring and fault prediction activities. *Journal of Systems and Software*, *85*(5), 1042–1057.

Arp, D. (n.d.). Retrieved from https://www.sec.cs.tu-bs.de/~danarp/drebin/download.html

Balaban, A. T. (1985). Applications of graph theory in chemistry. *Journal of chemical information and computer sciences*, *25*(3), 334–343.

Barabási, A.-L. (2016). *Network science*. Cambridge university press.

Barney, B., et al. (2010). Introduction to parallel computing. *Lawrence Livermore National Laboratory*, *6*(13), 10.

Bhowmick, S., & Srinivasan, S. (2013). A template for parallelizing the louvain method for modularity maximization. In *Dynamics on and of complex networks, volume 2* (pp. 111–124). Springer.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, *2008*(10), P10008.

Boppana, R. B. (1987). Eigenvalues and graph bisection: An average-case analysis. In *Foundations of computer science, 1987., 28th annual symposium on* (pp. 280–285).

Dietrich, J., Yakovlev, V., McCartin, C., Jenson, G., & Duchrow, M. (2008). Cluster analysis of java dependency graphs. In *Proceedings of the 4th acm symposium on software visualization* (pp. 91–94).

Du, Y., Wang, J., & Li, Q. (2017). An android malware detection approach using community structures of weighted function call graphs. *IEEE Access*, *5*, 17478–17486.

Dunn, R., Dudbridge, F., & Sanderson, C. M. (2005). The use of edge-betweenness clustering to investigate biological function in protein interaction networks. *BMC bioinformatics*, *6*(1), 39.

Fortunato, S. (2010). Community detection in graphs. *Physics reports*, *486*(3-5), 75–174.

Getting started androguard 3.3.5. (n.d.). Retrieved from `https://androguard.readthedocs.io/en/latest/intro/gettingstarted.html`

Grove, D., & Chambers, C. (2001). A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *23*(6), 685–746.

Huang, X., & Huang, W. (2015). Go: A cluster algorithm for graph visualization. *Journal of Visual Languages & Computing*, *28*, 71–82.

Kelly, G. (2014, Mar). *Report: 97% of mobile malware is on android. this is the easy way you stay safe.* Forbes Magazine. Retrieved from `https://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/#67a5eb9c2d4f`

Leskovec, J., Lang, K. J., & Mahoney, M. (2010). Empirical comparison of algorithms for network community detection. In *Proceedings of the 19th international conference on world wide web* (pp. 631–640).

Li, H., Zhao, H., Cai, W., Xu, J.-Q., & Ai, J. (2013). A modular attachment mechanism for software network evolution. *Physica A: Statistical Mechanics and its Applications*, *392*(9), 2025–2037.

Lu, H., Halappanavar, M., & Kalyanaraman, A. (2015). Parallel heuristics for scalable community detection. *Parallel Computing*, *47*, 19–37.

Mad. (2009, Jan). *Performance obstacles for threading: How do they affect openmp code?* Intel. Retrieved from `https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code`

Mcafee labs threats reports. (n.d.). Retrieved from `https://www.mcafee.com/enterprise/en-us/threat-center/mcafee-labs/reports.html`

Myers, C. R. (2003). Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, *68*(4), 046116.

Newman, M. E., & Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical review E*, *69*(2), 026113.

The open graph viz platform. (n.d.). Retrieved from `https://gephi.org/`

Paleari, R. (2011). Dealing with next-generation malware.

Pan, W., Li, B., Ma, Y., Liu, J., & Qin, Y. (2009). Class structure refactoring of object-oriented softwares using community detection in dependency networks. *Frontiers of Computer Science in China*, *3*(3), 396–404.

Qu, Y., Guan, X., Zheng, Q., Liu, T., Wang, L., Hou, Y., & Yang, Z. (2015). Exploring community structure of software call graph and its applications in class cohesion measurement. *Journal of Systems and Software*, *108*, 193–210.

Qu, Y., Guan, X., Zheng, Q., Liu, T., Zhou, J., & Li, J. (2015). Calling network: A new method for modeling software runtime behaviors. *ACM SIGSOFT Software Engineering Notes*, *40*(1), 1–8.

Shanbhaq, S. V. (2016). A faster version of louvain method for community detection for efficient modeling and analytics of cyber systems.

Staudt, C. L., & Meyerhenke, H. (2013). Engineering high-performance community detection heuristics for massive graphs. In *Parallel processing (icpp), 2013 42nd international conference on* (pp. 180–189).

Šubelj, L., & Bajec, M. (2011). Community structure of complex software systems: Analysis and applications. *Physica A: Statistical Mechanics and its Applications*, *390*(16), 2968–2975.

Valverde, S., & Solé, R. V. (2003). Hierarchical small worlds in software architecture. *arXiv preprint cond-mat/0307278*.

Wickramaarachchi, C., Frincu, M., Small, P., & Prasanna, V. K. (2014). Fast parallel algorithm for unfolding of communities in large graphs. In *High performance extreme computing conference (hpec), 2014 ieee* (pp. 1–6).

APPENDIX A. CODES

A.1 Compiling and Running Grappolo

The source code of Grappolo was retrived from the following site http://hpc.pnl.gov/people/hala/grappolo.html

The source code contains multiple .cpp files which are compiled together using a makefile. Following is the makefile.

```
#GCC Compilers:
CC  = gcc
CPP = g++
#CFLAGS   = -g -O3 -fopenmp -std=c99
CFLAGS   = -g -Ofast -fopenmp -std=c99
#CPPFLAGS = -g -O3 -fopenmp
CPPFLAGS = -g -Ofast -fopenmp -std=c++0x

#Intel Compilers:
#CC  = icc
#CPP = icpc
#CFLAGS = -fast -O2 -axT -openmp # -std=c99              #Intel Opt
#CPPFLAGS = -fast -O2 -axT -openmp # -std=c99            #Intel Opt

#METIS_HOME = $(HOME)/metis-5.0.2
METIS_HOME = /afs/msrc.pnl.gov/files/home/hala533/metis-5.0.2
METIS_INCLUDE = -I$(METIS_HOME)/include
METIS_LIB = -L$(METIS_HOME)/libmetis -lmetis -lm

LDFLAGS  = $(CPPFLAGS)
INCLUDES = . $(METIS_INCLUDE)
LIBS     = -lm


TARGET_1 = driverForGraphClustering
TARGET_2 = convertFileToBinary
TARGET_3 = convertFileToEdgeList
TARGET_4 = driverForColoringExperiments

TARGET_5 = driverForRmat
TARGET_6 = driverForRGG
TARGET_7 = driverForPartitioningWithMetis

TARGET_8 = convertSnapFileToBinary

#TARGET   = $(TARGET_1) $(TARGET_2) $(TARGET_3)
$(TARGET_4) $(TARGET_5) $(TARGET_6)

#TARGET = $(TARGET_1) $(TARGET_2) $(TARGET_3) $(TARGET_4)

TARGET = $(TARGET_1)

```

```
43
44 OBJECTS  = RngStream.o utilityFunctions.o parseInputFiles.o \
45           writeGraphDimacsFormat.o buildNextPhase.o \
46           coloringDistanceOne.o utilityClusteringFunctions.o \
47           parallelLouvainMethod.o parallelLouvainWithColoring.o \
48       louvainMultiPhaseRun.o parseInputParameters.o
49       vertexFollowing.o
50
51 all: $(TARGET) message
52
53 $(TARGET_1): $(OBJECTS) $(TARGET_1).o
54 $(CPP) $(LDFLAGS) -o $(TARGET_1) $(TARGET_1).o $(OBJECTS) $(LIBS)
55
56 $(TARGET_2): $(OBJECTS) $(TARGET_2).o
57     $(CPP) $(LDFLAGS) -o $(TARGET_2) $(TARGET_2).o $(OBJECTS) $(LIBS)
58
59 $(TARGET_3): $(OBJECTS) $(TARGET_3).o
60     $(CPP) $(LDFLAGS) -o $(TARGET_3) $(TARGET_3).o $(OBJECTS) $(LIBS)
61
62 $(TARGET_4): $(OBJECTS) $(TARGET_4).o
63     $(CPP) $(LDFLAGS) -o $(TARGET_4) $(TARGET_4).o $(OBJECTS) $(LIBS)
64
65 $(TARGET_5): $(OBJECTS) $(TARGET_5).o
66     $(CPP) $(LDFLAGS) -o $(TARGET_5) $(TARGET_5).o $(OBJECTS) $(LIBS)
67
68 $(TARGET_6): $(OBJECTS) $(TARGET_6).o
69     $(CPP) $(LDFLAGS) -o $(TARGET_6) $(TARGET_6).o $(OBJECTS) $(LIBS)
70
71 $(TARGET_7): $(OBJECTS) $(TARGET_7).o
72     $(CPP) $(LDFLAGS) -o $(TARGET_7) $(TARGET_7).o
73     $(OBJECTS)
74     $(METIS_LIB)
75
76 $(TARGET_8): $(OBJECTS) $(TARGET_8).o
77     $(CPP) $(LDFLAGS) -o $(TARGET_8) $(TARGET_8).o
78     $(OBJECTS)
79     $(METIS_LIB)
80
81 .c.o:
82     $(CC) $(CFLAGS) -c $< -I$(INCLUDES) -o $@
83
84 .cpp.o:
85     $(CPP) $(CPPFLAGS) -c $< -I$(INCLUDES) -o $@
86
87 clean:
88     rm -f $(TARGET).o $(OBJECTS)
89
90 wipe:
91     rm -f $(TARGET).o $(OBJECTS) $(TARGET) *~ *.bak
92
93 message:
94     echo "Executables: " $(TARGET) " have been created"
```

Listing A.1: Makefile

To compile the code execute following commands on the terminal.

```
96 $make
97 echo "Executables: " driverForGraphClustering " have been created"
98 Executables:  driverForGraphClustering  have been created
```

```
$./driverForGraphClustering sample.net -f 3 -v -c -o
```

Listing A.2: Compile and running commands in Terminal

The description of command line arguments is as follows

1. sample.net: Input file in Pajek format.
2. -f specifies the file type
3. The options for file types are as follows
   (a) 1- Matrix-Market
   (b) 2- DIMACS#9
   (c) 3- Pajek (each edge once)
   (d) 4- Pajek (twice)
   (e) 5- Metis (DIMACS#10)
   (f) 6- Simple edge list twice
   (g) 7- Binary format
4. -v: Signifies activation of vertex following
5. -c: Signifies activation of coloring
6. -o: Signifies saving the cluster information in an output file

A.2 Interpretation of Output

On running the algorithm, it outputs the information about the run on terminal. Following is an example of the output

```
*****************************************
Input Parameters:
*****************************************
Input File: sample.net
File type : 3
Threshold  : 1e-06
C-threshold: 0.0001
-----------------------------------------
Coloring   : TRUE
Strong scaling : FALSE
VF         : TRUE
Output     : TRUE
*****************************************
```

```
Within displayGraphCharacteristics()
*****************************************
General Graph: Characteristics :
*****************************************
Number of vertices   :  34
Number of edges      :  78
Maximum out-degree is:  17
Average out-degree is:  4.588235
Expected value of X^2:  35.647059
Variance is          :  14.595156
Standard deviation   :  3.820361
Isolated vertices    :  0 (0.00%)
Degree-one vertices  :  1 (2.94%)
Density              :  6.747405%
*****************************************
Vertex following is enabled.
Time to determine number of vertices (numNode) to fix: 0.000021
Graph will be modified -- 1 vertices need to be fixed.
Within renumberClustersContiguously()
Time to renumber clusters: 0.000026
Within buildNewGraphVF(): # of unique clusters= 33
Actual number of threads: 16
Time to initialize: 0.000
NE_out= 77   NE_self= 1
These should match: 155 == 155
Time to count edges: 0.000
Time to build the graph: 0.000
Total time: 0.000
Graph after modifications:
Within displayGraphCharacteristics()
*****************************************
General Graph: Characteristics :
*****************************************
Number of vertices   :  33
Number of edges      :  78
Maximum out-degree is:  17
Average out-degree is:  4.696970
Expected value of X^2:  36.696970
Variance is          :  14.635445
Standard deviation   :  3.825630
Isolated vertices    :  0 (0.00%)
Degree-one vertices  :  0 (0.00%)
Density              :  7.162534%
*****************************************
Within algoDistanceOneVertexColoringOpt()
Actual number of threads: 16 (requested: 16)
Vertices: 33  Edges: 78
Within generateRandomNumbers() -- Number of threads: 16
Each thread will add 2 edges
Results from parallel coloring:
*********************************************
** Iteration : 0
Time taken for Coloring:  0.000095 sec.
Conflicts          : 0
Time for detection : 0.000010 sec
*********************************************
Total number of colors used: 4
Number of conflicts overall: 0
Number of rounds           : 1
Total Time                 : 0.000105 sec
```

```
**********************************************
Check - SUCCESS: No conflicts exist

================================
Phase 1
================================
Within algoLouvainWithDistOneColoring()
Actual number of threads: 16 (requested: 16)
Time to initialize: 0.000
=================================================
=================================================
Itr      E_xx            A_x2           Curr-Mod
Time-1(s)       Time-2(s)        T/Itr(s)
=================================================
=================================================
1        56      3238    0.225920        0.000   0.000
           0.000
2        84      4898    0.337196        0.000   0.000
           0.000
3        86      4982    0.346565        0.000   0.000
           0.000
4        86      4982    0.346565        0.000   0.000
           0.000
=================================================
=================================================
Total time for 4 iterations is: 0.001064
=================================================
=================================================
Within renumberClustersContiguously()
Time to renumber clusters: 0.000003
Within buildNextLevelGraphOpt(): # of unique clusters= 7
Actual number of threads: 16 (requested: 16)
Time to initialize: 0.000
Time to count edges: 0.000
Time to build the graph: 0.000
Total time: 0.000
================================
Phase 2
================================
Within parallelLouvianMethod()
Actual number of threads: 16 (requested: 16)
Time to initialize: 0.000
=================================================
=================================================
Itr      E_xx            A_x2           Curr-Mod
Time-1(s)       Time-2(s)        T/Itr(s)
=================================================
=================================================
1        86      4982    0.346565        0.000   0.000
         0.000
2        114     7592    0.418803        0.000   0.000
         0.000
3        114     7592    0.418803        0.000   0.000
         0.000
=================================================
=================================================
Total time for 3 iterations is: 0.000185
=================================================
=================================================
Within renumberClustersContiguously()
```

```
Time to renumber clusters: 0.000002
Within buildNextLevelGraphOpt(): # of unique clusters= 4
Actual number of threads: 16 (requested: 16)
Time to initialize: 0.000
Time to count edges: 0.000
Time to build the graph: 0.000
Total time: 0.000
================================
Phase 3
================================
Within parallelLouvianMethod()
Actual number of threads: 16 (requested: 16)
Time to initialize: 0.000
=================================================
=================================================
Itr       E_xx            A_x2           Curr-Mod
    Time-1(s)       Time-2(s)        T/Itr(s)
=================================================
=================================================
1         114     7592    0.418803        0.000   0.000
         0.000
2         114     7592    0.418803        0.000   0.000
         0.000
=================================================
=================================================
Total time for 2 iterations is: 0.000107
=================================================
=================================================
Within renumberClustersContiguously()
Time to renumber clusters: 0.000001
*****************************************
*********    Compact Summary   *************
*****************************************
Total number of phases          : 3
Total number of iterations      : 9
Total time for clustering       : 0.001356
Total time for building phases  : 0.000190
Total time for coloring         : 0.000105
*****************************************
TOTAL TIME                      : 0.001651
*****************************************
Cluster information will be stored in file: sample.net_clustInfo
```

Listing A.3: Information from complete run

The information about the clusters is stored in the file *sample.net_clustInfo*. The file contains the clusterId for each vertex (the line number is the implicit id for a vertex.

A.3 Run Grappolo on Rice

To run the above code on Rice. First, the job submission file should be created. The contents of the job submission file is as follows:

```
#!/bin/sh -l
```

```
#PBS -l walltime=01:00:00
#PBS -q datalab

cd $PBS_O_WORKDIR
module load intel
export OMP_NUM_THREADS=20

cd grappolo_4
make
./driverForGraphClustering App362.net -f 3 -v -c -o
```

Listing A.4: Job submission file

To schedule a job following command is used.

```
$qsub jobsubmissionfilename
```

Listing A.5: Scheduling a job on Rice

A.4 Modifications on Grappolo

The modifications were done to the following files in the original Grappolo code

1. utilityClusteringFunctions.h
2. parallelLouvainMethod.cpp
3. utilityClusteringFunctions.cpp

In *utilityClusteringFunctions.h* the function definitions of *buildLocalMapCounter()* and *max()* are changed. The changes are as follows

```
#include "defs.h"

using namespace std;

void sumVertexDegree(edge* vtxInd, long* vtxPtr, long* vDegree,
long NV, Comm* cInfo);

double calConstantForSecondTerm(long* vDegree, long NV);

void initCommAss(long* pastCommAss, long* currCommAss, long NV);

long buildLocalMapCounter(long adj1, long adj2,
map<long, long> &clusterLocalMap_odd,
map<long, long> &clusterLocalMap_even,vector<double> &Counter,
edge* vtxInd, long* currCommAss, long me);

long max(map<long, long> &clusterLocalMap, vector<double> &Counter,
long selfLoop, Comm* cInfo, long degree, long sc, double constant,
double& maxGain);
```

Listing A.6: Changes in utilityClusteringFunctions.h

In *parallelLouvainMethod.cpp*, modifications were done for the functions *buildLocalMapCounter()* and *max()* . The modiifications were done from line 133. Following is the code snippet of the modified code for original Grappolo

```
omp_set_nested(1);
#pragma omp parallel for
    for (long i=0; i<NV; i++) {
      long adj1 = vtxPtr[i];
      long adj2 = vtxPtr[i+1];
      long selfLoop = 0;
      double maxGain_even = 0;
      double maxGain_odd = 0;
      //Build a datastructure to hold the cluster structure of its
      //neighbors

      //Map each neighbor's cluster to a local number
      map<long, long>::iterator storedAlready;

      //Number of edges in each unique cluster
      vector<double> Counter;
      map<long,long> clusterLocalMap_odd;
      map<long,long> clusterLocalMap_even;

      //Add v's current cluster:
      if(adj1 != adj2)
      {
          clusterLocalMap_even[currCommAss[i]] = 0;
          clusterLocalMap_odd[currCommAss[i]] = 0;

          //Initialize the counter to ZERO (no edges incident yet)
          Counter.push_back(0);

          //Find unique cluster ids and #of edges incident (eicj)
          selfLoop = buildLocalMapCounter(adj1, adj2,
          clusterLocalMap_odd,clusterLocalMap_even, Counter, vtxInd,
          currCommAss, i);

          // Update delta Q calculation
          clusterWeightInternal[i] += (long)Counter[0]; //(e_ix)
          long maxIndex_even;
          long maxIndex_odd;

          if (!clusterLocalMap_even.empty())
          {
          #pragma omp task shared(maxIndex_even,maxGain_even) untied
              {
                  maxIndex_even = max(clusterLocalMap_even, Counter,
                  selfLoop, cInfo, vDegree[i], currCommAss[i],
                  constantForSecondTerm,maxGain_even);
              }
          }

          if(!clusterLocalMap_odd.empty())
          {
          #pragma omp task shared(maxIndex_odd,maxGain_odd) untied
              {
                  maxIndex_odd = max(clusterLocalMap_odd, Counter,
                  selfLoop, cInfo, vDegree[i], currCommAss[i],
```

```
                    constantForSecondTerm,maxGain_odd);
                }
            }

            //Wait for the task to complete their execution
            #pragma omp taskwait
            {
                if(maxGain_even>maxGain_odd)
                {
                    targetCommAss[i]=maxIndex_even;

                }
                else
                {
                    targetCommAss[i]=maxIndex_odd;
                }
            }
            } else {
            targetCommAss[i] = -1;
            }
```

Listing A.7: Modifications in parallelLouvainMethod.cpp

In *utilityClusteringFunctions.cpp* the functions *buildLocalMapCounter()* and *max()* are changed. The changes are as follows:

```
long buildLocalMapCounter(long adj1, long adj2, map<long, long>
&clusterLocalMap_odd, map<long, long> &clusterLocalMap_even,
vector<double> &Counter, edge* vtxInd, long* currCommAss, long me) {
    map<long, long>::iterator storedAlready_odd;
    map<long, long>::iterator storedAlready_even;
    long numUniqueClusters = 1;
    int label=0;
    long selfLoop = 0;
    for(long j=adj1; j<adj2; j++) {
        if(vtxInd[j].tail == me) {  // SelfLoop need to be recorded
          selfLoop += (long)vtxInd[j].weight;
        }
        //Check if it already exists
        storedAlready_even = clusterLocalMap_even.find(
        currCommAss[vtxInd[j].tail]);
        storedAlready_odd = clusterLocalMap_odd.find(
        currCommAss[vtxInd[j].tail]);
        //Already exists
        if( storedAlready_even !=
        clusterLocalMap_even.end() ) {
        //Increment the counter with weight
          Counter[storedAlready_even->second]+= vtxInd[j].weight;
        }
        //Already exists
        else if( storedAlready_odd != clusterLocalMap_odd.end() ) {
        //Increment the counter with weight
          Counter[storedAlready_odd->second]+= vtxInd[j].weight;
        }

        else {
            label=rand();
            if(label%2==0)
            //Does not exist, add to the map
```

```
415                 clusterLocalMap_even[currCommAss[vtxInd[j].tail]] =
416                 numUniqueClusters;
417               else
418                 clusterLocalMap_odd[currCommAss[vtxInd[j].tail]] =
419                 numUniqueClusters;
420
421             Counter.push_back(vtxInd[j].weight); //Initialize the count
422
423             numUniqueClusters++;
424           }
425
426         }//End of for(j)
427
428   return selfLoop;
429 }//End of buildLocalMapCounter()
430
431 long max(map<long, long> &clusterLocalMap, vector<double> &Counter,
432          long selfLoop, Comm* cInfo, long degree, long sc,
433          double constant,double& maxGain) {
434
435     map<long, long>::iterator storedAlready;
436     //Assign the initial value as self community
437     long maxIndex = sc;
438     double curGain = 0;
439     double eix = Counter[0] - selfLoop;
440     double ax = cInfo[sc].degree - degree;
441     double eiy = 0;
442     double ay = 0;
443
444     storedAlready = clusterLocalMap.begin();
445     do {
446         if(sc != storedAlready->first) {
447         // degree of cluster y
448             ay = cInfo[storedAlready->first].degree;
449             eiy = Counter[storedAlready->second];
450             //Total edges incident on cluster y
451             curGain = 2*(eiy - eix) - 2*degree*(ay - ax)*constant;
452
453             if( (curGain > maxGain) ||
454                ((curGain==maxGain) && (curGain != 0) &&
455                (storedAlready->first < maxIndex)) ) {
456                 maxGain = curGain;
457                 maxIndex = storedAlready->first;
458             }
459         }
460         storedAlready++; //Go to the next cluster
461     } while ( storedAlready != clusterLocalMap.end() );
462
463     if(cInfo[maxIndex].size == 1 && cInfo[sc].size ==1 &&
464     maxIndex > sc) { //Swap protection
465     maxIndex = sc;
466   }
467   return maxIndex;
468 }//End max()
```

Listing A.8: Modifications in utilityClusteringFunctions.cpp