ENUMERATING K-CLIQUES IN A LARGE NETWORK USING APACHE SPARK

A Thesis

Submitted to the Faculty

of

Purdue University

by

Raja Sekhar Rao Dheekonda

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2017

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

Dr. Mohammad Al Hasan

Department of Computer and Information Science

Dr. Rajeev R. Raje

Department of Computer and Information Science

Dr. Fengguang Song

Department of Computer and Information Science

Approved by:

Dr. Shiaofen Fang

Head of the Departmental Graduate Program

I dedicate my thesis, my work to the most important people in my life, my parents, whose love, constant support and guidance have always been my biggest motivation. I would also like to dedicate this work to Mary, to whom I would forever be grateful.

ACKNOWLEDGMENTS

I would like to take this opportunity to express my gratitude towards a few people without whom this work would not be a reality. First and foremost, I would like to thank my thesis advisor, Dr. Mohammad Al Hasan, for his valuable guidance and encouragement throughout the thesis process. Dr. Hasan patiently guided me and always encouraged me with a smile. During the tough times, he mentored me in a very calm and collected manner and explained the concepts with not only clarity but with simplicity. It has been both a great privilege and a model experience to study and work under him. This project would not be a reality without his mentoring and encouragement.

I am also grateful to the thesis committee members: Dr. Rajeev Raje, and Dr. Fengguang Song for giving me the opportunity to pursue my thesis.

I would like to thank my parents for their constant support and motivation throughout my Master′s education. They have always been my unfailing foundation and the reason for my achievements. I would also like to express my gratitude to Mary, to whose kindness and care I will eternally be grateful.

I would also like to thank my lab mates for their inspiration throughout the project. They have made my research memorable. Also, I would like to extend a heartfelt thanks to my friends who constantly supported and helped me during my project.

Lastly, I feel indebted to the knowledge and guidance given to me by IUPUI, through the courses I studied and with the professors who helped and guided me throughout my Master′s education. This project would not be a reality without it.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

HDD	Hard Disk Drive
SSD	Solid State Disk
KC-SPARK	k-clique enumeration using spark framework
RDD	Resilient Distributed Dataset
HDFS	Hadoop Distributed File System

ABSTRACT

Dheekonda, Raja Sekhar Rao M.S., Purdue University, May 2017. Enumerating k-cliques in a Large Network Using Apache Spark. Major Professor: Mohammad Al Hasan.

Network analysis is an important research task which explains the relationships among various entities in a given domain. Most of the existing approaches of network analysis compute global properties of a network, such as transitivity, diameter, and all-pair shortest paths. They also study various non-random properties of a network, such as graph densification with shrinking diameter, small diameter, and scale-freeness. Such approaches enable us to understand real-life networks with global properties. However, the discovery of the local topological building blocks within a network is an important task, and examples include clique enumeration, graphlet counting, and motif counting. In this paper, my focus is to find an efficient solution of k-clique enumeration problem. A clique is a small, connected, and complete induced subgraph over a large network. However, enumerating cliques using sequential technologies is very time-consuming. Another promising direction that is being adopted is a solution that runs on distributed clusters of machines using the Hadoop mapreduce framework. However, the solution suffers from a general limitation of the framework, as Hadoop's mapreduce performs substantial amounts of reading and writing to disk. Thus, the running times of Hadoop-based approaches suffer enormously. To avoid these problems, we propose an efficient, scalable, and distributed solution, KC-SPARK , for enumerating cliques in real-life networks using the Apache Spark in-memory cluster computing framework. Experiment results show that KC-SPARK can enumerate k-cliques from very large real-life networks, whereas a single commodity machine cannot produce the same desired result in a feasible amount of time. We also compared

KC-SPARK with Hadoop mapreduce solutions and found the algorithm to be 80-100 percent faster in terms of running times. On the other hand, we compared with the triangle enumeration with Hadoop mapreduce and results shown that KC-SPARK is 8-10 times faster than mapreduce implementation with the same cluster setup. Furthermore, the overall performance of KC-SPARK is improved by using Spark's inbuilt caching and broadcast transformations.

1 INTRODUCTION

Network analysis is an important research task which explains the relationship among various entities in a given domain. Most of the existing approaches of network analysis either compute some global properties of a network, such as transitivity, diameter, all-pair shortest paths, etc. or they study various non-random properties of a network, such as graph densification with shrinking diameter [1], small diameter [2], and scale-free-ness [3]. Unfortunately, global analysis of a network fails to capture the local topological building blocks of a network.

Discovery of local topological building blocks is important for understanding the formation of a network from the perspective of the vertices in the network, so there is a growing need to develop algorithms for enumerating and counting local topological building blocks of a network. In recent years, researchers have developed algorithms for enumerating and counting various kinds of local topological structures of a network, examples include clique enumeration [4–10], graphlet counting [11–17], and motif counting [18–21]. However, all these tasks are expensive and the cost of the cost increases exponentially with the number of vertices.

Among various local topological analysis of networks, enumeration of cliques provably received the most attention due to its wide-spread applications in various fields, such as, system science [22], social science [6] and bioinformatics [11]. In bioinformatics, clique enumeration has been used for protein-protein interactions [23], sequence clustering [23], and gene expression analysis [24]. In system science, cliques are used for circuit design [25], and in social science, cliques are also used for detecting communities [26]. In cheminformatics, cliques have been used for comparing the chemical properties of two compounds [27]. In machine learning and information retrieval, cliques are used for spam detection [28], and association rule mining [29].

Enumeration of cliques in a network are categorized into 3 forms, such as, maximal clique, maximum clique, and k-clique. For the first form, i.e., enumerating maximal cliques several works have been proposed by the researchers [5,30–39]. However, solving this task in large real-world graphs is difficult and expensive. In fact, Tomita et al. [5] have proven that the time complexity of maximal clique enumeration is exponential; if n is the number of vertices the complexity is $O(3^{n/3})$. For the second form, which is maximum clique enumeration several works have also been proposed [40–48]. One of the fastest among those is proposed by Robson [48] which runs in $O(2^{0.249n})$. Since the collection of of maximal cliques is a superset of the collection of maximum cliques, the latter form of clique enumeration is less interesting and easier to solve. Nevertheless, both the above forms are $\mathcal{NP}$-complete, so for large input networks enumeration of maximal or maximum cliques is very time-consuming. Besides, for many real-life applications of cliques (such as, community detection and spam detection), one may only need cliques upto a certain number of vertices only, resulting the third form of clique enumeration, namely k-clique enumeration. In this paper, my focus is to find an efficient solution of k-clique enumeration problem.

The brute-force time complexity of enumerating fixed size (say, k) clique is $O(n^k k^2)$; where n is the number of vertices and k is the desired clique size. Here, $O(n^k)$ is the number of potential subgraphs of size k in the network, and the time complexity to analyze whether a subgraph is a clique is $O(k^2)$. Various sequential clique enumeration algorithms have been proposed over the years. One of the fastest solutions among those is proposed by Virginia Vassilevska [49] which runs in $O(n^k/(\epsilon \log n)^{k-1})$ time and $O(n^\epsilon)$ space, for all $\epsilon > 0$, on networks with n nodes. In fact, this is the first solution that takes $o(n^k)$ time and $O(n^c)$ space, for some constant c independent of clique size k. MACE, introduced by Tomita et al. [5], is Another solution that gained a lot of prominence. However, in this approach, when the size of the input network increases, the computation can not be completed within reasonable amount of time on a single commodity machine. Another approach was put forward by Borgatti and

Everett et al. [6] who introduced the UCINET software. This software is not scalable as it cannot work on real-life networks with more than 2 million nodes.

Networks, in presents days are very large; specifically on-line social networks [50] and web networks [51] typically consist of millions and even billions of vertices and edges. None of the existing sequential k-clique enumeration methods perform well on such networks either due to memory limitation and/or due to very large running time. To overcome these limitations, scientists have proposed k-cliques enumeration methods which find approximation solutions. Another promising direction that is adopted is the solution that run on distributed clusters of machines.

A couple of approximation algorithms have been suggested by Lars Eilstrup et al. [4] and Shweta Jain et al. [7] that can be applied for 2 million nodes easily. But these algorithms have their limitations too and the maximum number of nodes they can handle is 100 million. Also, since they are approximations, they are not very ideal for large real-world networks. Other sequential algorithms have been discussed in detail in the Previous Research chapter 2.

A collection of parallel and distributed solutions are proposed in recent years by Suri and Vassilvitskii [8], Afrati et al. [9] and Finocchi et al. [10]. All these methods run on mapreduce based distributed platform. The earliest among them, Suri and Vassilvitskii's [8] algorithm enumerates triangles only, which are merely cliques of size three. Afrati's [9] method works for arbitrary k size cliques, but it is based on "multiway join" operation in mapreduce framework. Since join takes enormous amount of time, the performance of this method is poor for large real-life networks. Finocchi's [10] algorithm is the latest among the distributed solutions of k-clique enumeration. It is considered to be better than the above two algorithms. However, all the above methods suffer from a general limitation of mapreduce framework, i.e., they perform substantial amount of reading and writing to disk file, thus their running time suffers enormously.

To avoid the above mentioned limitations, researchers, in recent years, are adopting in-memory distributed platforms for large scale data processing, such as Spark [52].

Spark is an open source in-memory cluster computing framework that supports iterative and interactive applications with implicit data parallelism while retaining the Hadoop mapreduce scalability and built-in fault tolerance. This is achieved by the Resilient Distributed Datasets (RDDs) data structure in Spark. RDD is a read-only set of objects that is partitioned across the cluster of machines. Should there be a loss in any partition, it is rebuilt implicitly using Spark's in-built RDD lineage [52]. Also it is reported that Spark outperforms Hadoop by being 10 times faster on disk based applications and 100 times faster for in-memory applications [52].

In this work, we propose a new algorithm KC-SPARK , which enumerates all k-cliques for any given k value. Our algorithm runs on Spark, which makes it scalable, distributed and fault-tolerant. Experiment results show that KC-SPARK can enumerate k-cliques from very large real-life networks, where a single commodity machine cannot produce the desired result in feasible time. In our experiment results, of the network ca-HepPh, enumerating 5-cliques in a single machine with quad-core took 18 hours approximately and it took 53 minutes when executed in cluster of 10 machines. Execution running time comparison between standalone mode and cluster mode for different benchmark graphs are given in Table 5.8. We also compared our solution with the mapreduce framework proposed by Finocchi et al. [10] and Afrati et al. [9] and our algorithm has proven to be 80-100 percent faster than both of these methods. Execution running time comparison of our solution i.e., KC-SPARK with the afore-mentioned mapreduce solutions for different benchmark graphs are given in Table 5.3. Furthermore, we also ran our algorithm for k=3 to enumerate triangles only and compared this setup with Suri et al. [8] method. Our results show that KC-SPARK performed faster than their algorithm even with around 40% smaller number of machines. We also compared triangle enumeration using Hadoop mapreduce with the same cluster set up and results shown that KC-SPARK is 8-10 times faster than Hadoop mapreduce shown in Table 5.6

1.1 Thesis Structure

Chapter 1 provides the introduction of k-clique enumeration and the rationale for using in-memory solutions compared to sequential and Hadoop mapreduce frameworks. The major applications of using cliques especially in social and web networks have also been discussed. Chapter 2 provides details about previous research of different clique enumeration techniques using sequential algorithms as well as parallel solutions. Chapter 3 presents the background knowledge of graph mining along with Apache Spark technology which have been used for the algorithm implementation.

Chapter 4 describes the proposed work in a detailed manner by taking a small graph as an example. It also describes the broadcasting and caching of in-built transformations used to achieve better performance in execution time. Chapter 5 presents the experiments and results performed on benchmark graphs. The experiments are conducted on standard data sets taken from SNAP repository [53] and a comparison of execution times with the mapreduce implementations [8–10] for standard data sets has been done. Chapter 6.2 concludes the thesis with some suggestions of future works.

2 PREVIOUS RESEARCH

Clique enumeration problems are categorized into three groups:

1. **Maximal clique** : enumerating cliques that cannot be extended even after including one more vertex.

2. **Maximum clique**: finding the clique with the largest possible number of vertices in a given network.

3. **k-clique mining**: enumeration of all cliques of size k in a given network.

2.1 Review of sequential algorithms for k-clique enumeration

Several solutions are proposed for each of the above groups. Most of them are based on the maximal clique enumeration problem. However, solving this task in large real-world graphs is difficult and expensive. In fact, Tomita et al. [5] have proven that the time complexity of maximal clique enumeration is exponential; if n is the number of vertices the complexity is $O(3^{n/3})$. The most effective and well-known solution was proposed by Bron and Kerbosch et al. [30]. In fact, a faster algorithm has been suggested by Tomita et al. [5] and its time complexity is almost similar to Bron and Kerbosch et al. [30]. Tomita's [5] work was initially focused on enumerating maximal cliques, but it can also be used to enumerate k-cliques in a graph as well. Their implementation is called MACE which is available online.

The sequential algorithm that is the closest to our work, proposed by Virginia Vassilevska [49] for enumerating k-clique, which runs in $O(n^k/(\epsilon \log n)^{k-1})$ time and $O(n^\epsilon)$ space, for all $\epsilon > 0$, on networks with n nodes. In fact, this is the first solution that takes $o(n^k)$ time and $O(n^c)$ space, for some constant c independent of clique size k. The algorithm aims to deal with the complexity of gathering $(k-1)$-cliques in many

small subgraphs, where each subgraph is of size $(\log n)$, by attempting to complete these cliques by adding an extra vertex. This algorithm partitions the input vertices into $n/(\epsilon' \log n)$ parts, where $\epsilon' = \epsilon/2(k-1)$, and each part is of size $(\epsilon' \log n)$ nodes. The partition is considered according to the order of the columns present in the adjacency matrix. This makes sure that concatenation of k chunks, each of $(\epsilon' \log n)$, can be done in $O(k)$ time. It is a space efficient algorithm, where each iteration reuses the space used by the previous iterations.

Borgatti and Everett et al. [6] present a software called UCINET that is used to enumerate k-cliques in a network. This is one of the traditional sequential algorithms to analyze social network data. This package can also be utilized to visualize the input network with intermediate results. The limitation of this algorithm is that, it can only enumerate the cliques upto 2 million nodes. And hence, enumeration of cliques in large networks such as web, social etc is impossible.

Lars Eilstrup et al. [4] present an approximate solution for enumerating cliques in large networks. This algorithm is based on randomization and approximation. But their algorithm does not provide an exact count of the clique enumeration.

Shweta Jain and Seshadhri et al. [7] propose an approximation solution for counting cliques over a network with less than 2 % error. But this solution is limited to one hundred million edges and also, it is not scalable for large real-life graphs.

2.2 Review of Parallel algorithms for k-clique enumeration

Most of the solutions are exist for enumerating cliques in a single machine environment, but very few solutions are exist for distributed and parallel settings. Few of such are presented by Suri and Vassilvitskii [8], Afrati et al. [9], Tsourakakis et al. [15], and Finocchi et al. [10], all of which are primarily based on hadoop mapreduce distributed computing framework.

Suri and Vassilvitskii's [8] proposed the enumeration of triangle counting in a network, where the value of k=3 in k-clique. As mentioned before, this work is based

on mapreduce framework and enumerated triangles over real-life graphs. The main idea of this algorithm is performed in two rounds. In the first round, it computes all possible 2 length path in the network by pivoting on every vertex in a distributed manner. In the second round, it checks which of these 2 length path can be closed by an edge, which forms a triangle. Finally, they enumerate and count the triangles in the network. In fact, this algorithm takes $O(m^{3/2})$ space. They also provided a detailed research on the curse of map reducer, which is one of the biggest problems in distributed computing. They have performed experiments in a 16-36 node cluster. As the name suggests, it works only for 3-clique and they are not able to enumerate cliques for cliques of size k>3. Another limitation that comes from using hadoop mapreduce framework is the I/O bottleneck which is very time consuming process for enumeration in large real-life graphs.

Afrati et al. [9] proposed a distributed solution for enumerating complete sub-graphs using multiway joins with mapreduce framework. The basic idea of this algorithm is partition the input network into sub-graphs and by using sequential algorithms on each sub-graph to enumerate cliques. This method works for arbitrary k size cliques, but it is based on "multiway join" operation in mapreduce framework. Since join takes enormous amount of time, the performance of this method is poor for large real-life networks. In addition to the I/O bottleneck limitation of using Hadoop, their algorithm performed poor in execution times to enumerate cliques over large real time networks such as power-law random-graphs.

Tsourakakis et al. [15] proposed an algorithm for counting triangles using approximation based methods. They used mapreduce framework, but their algorithm is limited to triangle counting. Their algorithm has the same limitations as the triangle counting method suggested by Suri and Vassilvitskii et al [8]. Besides, this algorithm does not enumerate the cliques in a given network.

Finocchi et al. [10] also presented a solution for the enumeration of k-cliques in large-scale networks using mapreduce framework, with their implementation being able to handle cliques of size $k >= 3$. Total space consumed for k-clique enumeration

is $O(m^{3/2})$ and time complexity is given as $O(m^{k/2})$ where m is the number of edges in a given network and k is the clique size. They also provided a sampling based approach for enumeration of cliques in large networks with extremely accurate estimates and high speedups. Their approximation algorithm is specially useful when it is not feasible to use an exact algorithm for enumeration over large graphs. In such instances, exact algorithm can be used until a certain threshold, after which approximation can be implemented for effective time management. This is one of the fastest solutions for enumeration of k-cliques compared to the above solutions [8] [9]. The experiments are done over Amazon EC2 platform with 16 machines. Each machine has 4 virtual cores, 7.5 GB primary memory, and a 32 GB solid state disk. Although it can be said that this is the best solution so far, in this approach, intermediate outputs are stored in the solid state disk and the data is retrieved when required. This again leads to the I/O bottleneck problem rendered by hadoop.

To provide an effective solution to the above mentioned drawbacks of sequential and hadoop based implementations, we chose Spark in-memory cluster computing framework to enumerate cliques over large scale graphs and also to eliminate I/O bottlenecks which has been the primary issue of using hadoop based framework. We present a novel algorithm KC-SPARK to enumerate k-cliques in a graph using Spark in-memory cluster computing framework.

3 BACKGROUND

Section3.1 describes the domain knowledge required for this research. Section3.2 discusses the technological aspects of Spark distributed framework.

3.1 Notation

Consider $G = (V, E)$, which is a connected, undirected, unweighted graph where V refers to the set of vertices and E refers to the set of Edges. Edge e ∈ E and is represented using two vertices (v_i,v_j) which belongs to set V. In this report, the term graph is sometimes referred to as a network. A graph is said to be connected, only if there exists a path between any two vertices in the graph. For a connected network, there should not be any unreachable vertices. Otherwise, it is said to be disconnected. A simple graph can be connected or disconnected, is one in which there are no self loops and multiple edges present between any two vertices selected. An undirected network is a network, where all the edges are bidirectional. An unweighted graph does not have weights associated for an edge. Neighborhood for a vertex v is represented as $\mathcal{N}(v)$, where $\mathcal{N}(v)$ defines the set containing $(u \in V : (u, v) \in E)$. Neighborhood can also be referred as adjacency list in this report.

Graph $G' = (V', E')$ can be considered as a subgraph of graph G, if and only if $V' \subseteq V$ and $E' \subseteq E$. For a graph G, if $V' \subseteq V$ and $E' \subseteq E$ and $\{e = (v_a, v_b) : v_a, v_b \in V', e \in E, e \notin E'\} = \phi$, then $G' = (V', E')$ is defined as a vertex-induced subgraph. A vertex-induced subgraph $G' = (V', E')$ is a subset of the vertices of a graph $G = (V, E)$ together with any edges whose endpoints are both in this subset. A vertex induced graph is also referred to as an induced graph in this report.

Two graphs G and G' are isomorphic, if there exists a structure preserving (both adjacency and non adjacency preserving) bijection $f : V \rightarrow V'$; such a function f is

called an isomorphic function from G to G'. An embedding of a graph G' in graph G is a sub graph S of G, such that S and G' are isomorphic to each other. S is an induced embedding of G' in graph G, when the sub graph S is vertex induced sub graph of G .

A complete graph is an undirected graph where every pair of distinct vertices is connected by a unique edge. Any induced graph that is a complete graph forms a ***clique***. In graph domain, a clique is a small, connected, complete induced subgraph of a network. In this work, we work with enumeration of all possible cliques having k vertices; where k=3,4,5 etc. We refer to a clique with k vertices as k-clique; So, 1-clique is simply a vertex, 2-clique is an edge between two vertices, 3-clique is a triangle. For a k-clique, all k vertices are connected to each other, where the total number of edges are $\binom{k}{2}$ i.e., $(k * (k - 1))/2$.

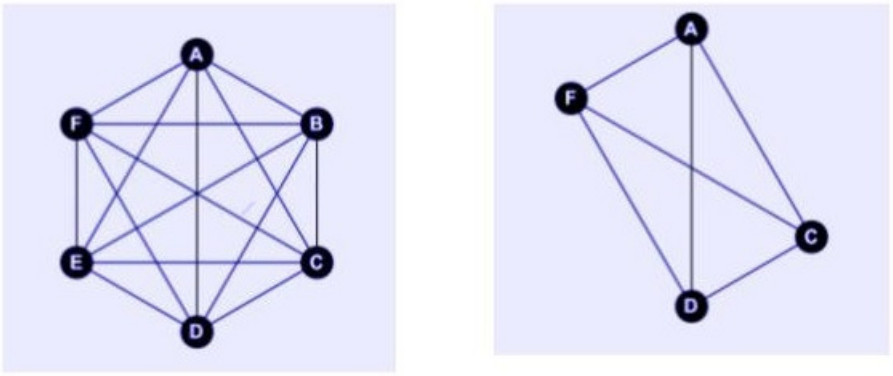

Figure 3.1. subgraph with set of vertices (A,C,D,F) induced from the complete graph k_6

The figure 3.1 demonstrates the subgraph induced from the complete graph k_6 by a set of vertices (A,C,D,F). Here, the induced subgraph forms a complete graph. Therefore, the induced subgraph is called as 4-***clique***, since it contains 4 vertices. The total number of edges present in the subgraph is 6; from the formula $\binom{4}{2} = (4 * (4 - 1))/2 = 6$.

3.2 Technology

3.2.1 Apache Spark

Apache Spark is an open source in-memory cluster computing platform [52]. It belongs to the Hadoop open-source community and is built on top of Hadoop Distributed File System (HDFS). It was developed to overcome the limitations in the mapreduce cluster computing paradigm. There are some similarities between Hadoop and Spark, but the later performs better in some specific types of applications specifically for iterative applications. Also it is reported that Spark outperforms Hadoop by being 10 times faster on disk based applications and 100 times faster for in-memory applications [52].

The main advantage of Spark is that it provides in-memory cluster computing which speeds up the execution of the iterative applications. The input data is loaded into the main memory which can be processed by multiple processes. As the data resides in the main memory, it takes constant time to access the data unlike from disk access and hence it completely removes the disk I/O time. Due to this advantage, it is well suited for iterative and machine learning algorithms. In addition, it is also efficient for processing large scale applications, such as, community detection and spam detection.

Apache Spark works with the master/worker model. The main operation in the Spark program is SparkContext, which is also called master (also referred to as driver in some cases) that manages workers, where executors run. Both, the Driver and the Executor execute in their own java processes [52]. Spark package must be installed across all the workers in the cluster. Every Spark program contains transformations and actions which is similar to methods in object-oriented terminology. Transformations are evaluated in lazy manner and actions are computed immediately, whenever action is invoked. If the program contains collect transformation, then SparkContext driver collects the results from all the workers that are associated with the task in the cluster and produces it to the master. The main responsibility of the SparkContext

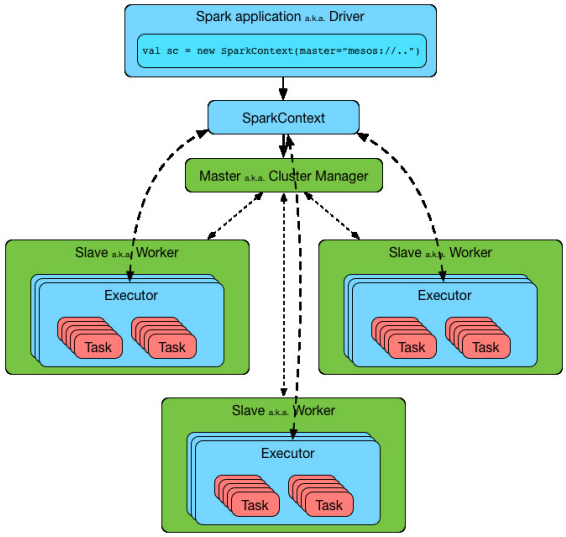

Figure 3.2. Spark Architecture in detail. Figure borrowed from [54]

is to make sure that the workers are not sitting idle, by assigning them tasks periodically. To achieve scalability, workers can be increased dynamically just by adding them to the cluster with Spark and Scala installed. The detailed architecture is shown in the figure 3.2

3.2.2 Specific Applications performed well by spark

Below are the drawbacks of Hadoop when compared to Spark.

1. **Iterative Jobs**:Many machine learning algorithms follow an iterative approach, which requires the same algorithm to be applied repeatedly. Now, each iteration can be viewed as a mapreduce job. The disadvantage here is that for every iteration the same data needs to be loaded from the disk. Hence this operation is expensive and is very time consuming.

2. **Interactive Analytics** : Hadoop is generally used to run queries on databases with large amount of data using Pig and Hive. Now, the user expect that the data is loaded once into the main memory and then queried multiple times.

But in Hadoop each query is executed as a separate job and hence each of them access the disk separately. This is not desirable as it increases the execution time and does not effectively utilize the resources.

To overcome the above limitation, Spark uses an abstraction called Resilient Distributed Data sets (RDDs). RDD is a read-only collection of objects which is partitioned over several machines. It also can be re-built if a partition accidentally crashes. It is reported that Spark is 10x faster when compared to Hadoop on disk and 100x faster on memory in performing iterative machine learning algorithms. It [52] also can be used to execute a query on a 39GB data set with sub-second response time.

3.2.3 Spark Programming Model

RDD is a read-only collection of objects which is partitioned over several machines. It can also be re-built if a partition crashes accidentally. The main advantage with RDD is that it does not have to reside in the physical memory, instead a handle to the RDD contains all the necessary information to construct the RDD from the data in the reliable storage. With this advantage it can re-construct any RDD if any node fails. As the data does not have to exist in the physical storage, this makes the iterative and interactive jobs to execute much faster compared to Hadoop. Each RDD is represented as a Scala object, if we use Scala as the programming language. There are four ways in which a RDD can be constructed,

1. **File**: Shared file system, for example, HDFS.

2. **Parallelizing**: In this procedure the array is divided into a number of slices and then each of these slices can be sent to multiple nodes.

3. **Transforming an existing RDD [52]**: In this procedure a dataset of type X can be transformed to a dataset of type Y. This transformation can be done by

using the flatMap operation, which scans each element of the dataset by using a user-defined function.

4. **Changing the persistence**: RDD are lazy when they are initialized and they are not stored in the disk immediately. They are stored in the main memory as long as enough space is available. They are only saved onto the disk if the action is invoked in the application. The persistence properties of the RDD is configurable and can be modified to cache or save action.

The cache action means that the data set is available in the cache for faster access in future i.e., it is kept in the memory. As the dataset is cached and not stored on the file system such as HDFS in mapreduce; execution takes very fast. This is the main advantage for the Spark where the program can access data locally. Now, the caution here is that enough memory should be available across all the machines. If the memory is not sufficient then Spark recomputes the data as and when it is required.

The save action means that the data set is saved and is written onto the file system. Now the saved file can be referred for all the future operations.

3.2.4 Parallel Operations

The main aim of parallel operations is to increase the speed of execution. Below are the parallel operations which are possible with RDDs,

1. **map** : It is a transformation which takes input as dataset element through a function and outputs a new RDD representing the results.

2. **Reduce**:The reduce operation is similar to the Hadoop reduce operation which performs action. This operation will combine the results from all the worker nodes and then gives a combined result to the master node.

3. **Collect**: This operation assists in getting the results from all the worker nodes i.e., collection of results. It also sends all the required information from the

data set to the user or driver program. For example, in case of processing an array, the user can map the array to all the available nodes. After the nodes have finished the processing, the collect operation can collect the results. The main advantage here is the single collect operation that gets the results from all the nodes.

4. **Foreach**: This transformation is used to iterate through the RDD elements stored in the spark RDD construct.

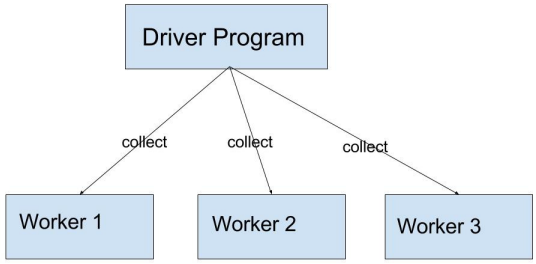

Figure 3.3. Spark collect action

As shown in the figure3.3, the elements of the data set are first mapped onto the available nodes and then the processing is performed. In this way we are parallelizing the processing of the data set between the nodes. The results are later collected by the user or driver program.

3.2.5 Shared member Variables

In Spark, transformations like map, reducebykey, groupbykey, filter are used in the program. We generally call these transformations by passing the functions as parameters to Spark. Therefore the variables used in these transformations should be

within the scope of functions where it gets executed by Scala user defined functions. Suppose if a slave node wants to execute a specific function, then all the corresponding variables required for that procedure needs to be copied into that particular slave machine. However, spark uses two special kind of shared variables called **Broadcast** and **Accumulators**. Due to these shared variables, we can just invoke these transformations by writing 2 lines of code in the program. It provides program level transparency, where Spark developers can concentrate on application logic rather than lower level implementation details.

1. **Broadcast Variables**: Broadcast variables allow the programmer to keep a read only variable on each machine that can be treated like cached object [52] rather than transporting a copy of the whole object. It uses efficient algorithms to communicate across the machines with reduced communication cost in the cluster. This variable is being invoked from the SparkContext package. Doing this invocation, the data will be cached across all the worker machines. Due to this, it will take constant time in order to access the data. Value can be returned using the in-built property called valueof.

2. **Accumulators**: This operation supports parallel implementations especially to implement sums and counters. This variable is fault tolerant and it uses mapreduce paradigm. We can define our own type of accumulators according to the scenario.

4 PROPOSED ALGORITHM

Section 4.1 provides the architecture and implementation details of KC-SPARK algorithm. We then explain the algorithm by executing it over a small graph. Section 4.2 provides the detailed information about the proposed algorithm, and also includes some sample code snippet. Section 4.3 provides the study of time complexity of KC-SPARK in the distributed cluster.

4.1 Architecture of KC-SPARK

The architecture of KC-SPARK is shown in Figure 4.1. The input of this algorithm is a graph G which is stored in a file as a list of edges. Each line of this file is a pair of numbers separated by a delimiter (we used tab a a delimiter) representing the source and destination vertices of one of G's edges. The edges are listed in the input file in an arbitrary order.

The main idea of KC-SPARK is to enumerate cliques of fixed size k in a network *iteratively*. For an input graph G, KC-SPARK finds all cliques of size l, from the list of all $l-1$ size cliques and the adjacency list of G for an l value, 2 unto k. Thus, the first iteration generates all size-2 cliques (edges), the second iteration generates all size-3 cliques (triangles), the third iteration generates all size-4 cliques, and the process continues until all k-cliques are generated.

A key idea of KC-SPARK in that in each of its iterations, each of the l-size cliques are enumerated exactly once, starting from $l = 2$. To achieve this, KC-SPARK view an undirected graph G as a DAG G^d, where each edge (u, v) is directed in a given precedence order of vertices, $u \prec v$. We consider $u \prec v$, if $d(u) < d(v)$ or $(d(u) = d(v)) \wedge (u.\text{id} < v.\text{id})$, here $d(\cdot)$ stands for the degree value of a vertex. Thus, in G^d, every edge is directed from a low degree vertex to a high degree vertex. In case an

edge connects two vertices of the same degree, the direction of the edge goes from the smaller id vertex to the higher id vertex. By using this DAG representation of G, each of the cliques will be enumerated only by the highest precedence vertex of that clique, so duplicate enumeration of cliques is entirely eliminated.

First iteration of KC-SPARK computes the adjacency list of G, the size of adjacency list of each vertex is the degree of the vertex in G. From the degree value we can easily obtain the precedence order of the vertices. Then the next iteration constructs the adjacency list of DAG G^d. For an undirected edge (u, v) of G, the adjacency list of u contains (u, v), and the adjacency list of v contains (v, u). However, for G^d, an edge between u and v is listed only in the adjacency list of either u or v (exclusively), depending on the precedence order. In subsequent discussion the adjacency list of G^d is called *filtered adjacency list* to distinct it from the original adjacency list of G and we use $\mathcal{N}(v)$ to represent the filtered adjacency list of v. KC-SPARK broadcasts and caches the adjacency list $\mathcal{N}(v)$ of DAG G^d across all the worker machines in the cluster. By using this, it eliminates the I/O bottleneck to/from the disk. Thus, $\mathcal{N}(v)$ acts as local or in-memory data structure during computation across all the machines in the cluster. After the above process, KC-SPARK enumerates l-cliques from the $(l-1)$-cliques, which we discuss in the subsequent paragraph.

In order to compute l-cliques from $(l-1)$-cliques, two operations are carried out: Extension and Completion. Extension takes two inputs: first is a $(l-1)$ sized clique (say, C_{l-1}), and the second is a list of vertices (say, E), which can be used to extend C_{l-1} to an l size clique, C_l. For a vertex $v \in E$ and for all vertices $u \in C_{l-1}$, (u, v) edge exists. For example, consider Figure 4.4; When we are extending the 2-clique $(1, 2)$, the extension list contains $\{3, 4\}$. The extension operation returns a tuple of size 3, which consists of the vertex that is being added into C_{l-1}, l-size clique C_l (list of vertices), and possible list of extensions of C_l. For the above example, the output of Extension would be $(3, (1, 2, 3), \{3, 4\})$. The third field of Extension output, which is an extension list of C_l is not necessarily valid, rather it is a super-set of the valid extension list. Therefore, we need to perform the Completion operation. Completion

takes two inputs: first, the output of Extension and the second is adjacency list of the "key" vertex of Extension. Completion returns C_l-clique (list of vertices) and its valid set of extensions. For our example the only valid extension for the clique $(1, 2, 3)$ is $\{4\}$. So the output of Completion is $(1, 2, 3), \{4\})$. From this the Extension of the next iteration to obtain a clique of size $l + 1$ can proceeds.

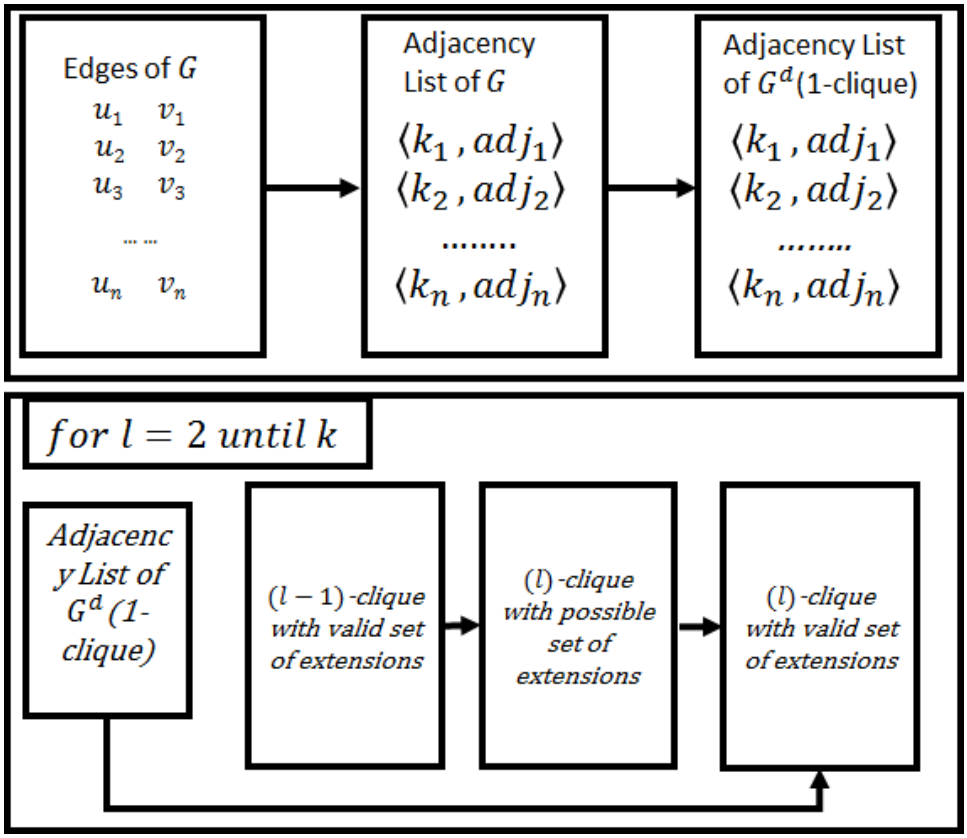

Figure 4.1. Architecture of KC-SPARK

4.2 KC-SPARK : Enumeration of k-cliques using Spark

In this section, we discuss KC-SPARK algorithm in a detailed manner. KC-SPARK is an iterative algorithm where k-clique enumeration is derived from $(k-1)$-clique. We first discuss an iterative KC-SPARK enumeration of cliques in Section 4.2.1. In

the subsequent sections, we provided a detailed explanation of all the steps that KC-SPARK performs.

4.2.1 Iterative KC-SPARK

Spark is based on mapreduce model and supports iterative computation. As it does not provide an iterative approach by default, designer is responsible to write the iteration logic for large networks. In an iterative Spark version, KC-SPARK generates 1-clique which contains the vertices and adjacency list of G_d of the network. Then, it generates 2-clique which are the edges of the network from the 1-clique RDD. This process is repeated until k-cliques are enumerated. The overall process of the iterative spark algorithm KC-SPARK is shown in Algorithm1.

Algorithm 1 Steps for KC-SPARK

1: Convert G into adjacency list format using Algorithm 2
2: Compute adjacency list of G^d from the adjacency list of G using Algorithm 3
3: Cache and broadcast adjacency list of G^d using Algorithm 4
4: Generate k-clique extension using Algorithm 5
5: Generate k-clique completion using Algorithm 6

4.2.2 Distributing the enumeration and sample code snippet

A clique of size k (for $k > 1$) can be generated from its sub-cliques. Starting from a vertex, we can extend it to an edge, then to a triangle, then to a 4-size clique all the way unto a k-size clique. For a k-size clique, there are $k!$ possible enumeration paths. Since we want a unique enumeration, we generate a k-size clique by adding the vertices of the clique in their precedence

by incrementally adding one of its vertices to and its relevant edges incrementally.

which are induced with the vertices that are already part of that clique.

from a clique of size $k-1$ by adding a neighboring node and its corresponding edges. This gives us a breadth first search (BFS) order exploration of clique enumeration (see Figure 4.2). In BFS order, all the cliques of size l are enumerated before any $l+1$ cliques.

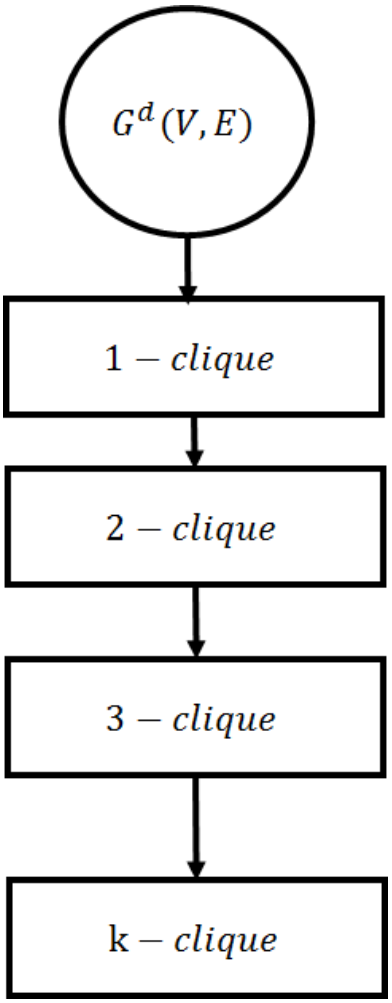

Figure 4.2. BFS exploration of KC-SPARK

For distributed clique enumeration, we use Spark computation framework. In this framework collection of records are stored as resilient distributed datasets known as RDD. Initially, we construct a RDD containing all 1-clique embedding, such that each clique is a record in the RDD. In the RDD, the embedding of 1-clique is represented

as a key-value pair: the key is the vertex id and the value is is the adjacency list of that vertex in G^d. For a size l, we have a distinct RDD, which we name RDD_l, containing all l-size cliques and their valid extensions. The following lemma holds.

Algorithm 2 Algorithm that generates adjacency list from the input graph G

1: **procedure** MAP(u, v) ▷ where u is source vertex and v is destination vertex
2: **Input file**: each line is represented as an undirected and unweighted edge
3: **emit** $\langle u, v \rangle$
4: **emit** $\langle v, u \rangle$
5: **end procedure**
6: **procedure** GROUPBYKEY$(input)$ ▷ where *input* is generated from the above Map
7: **Input** : List of the undirected edges $\langle u, v \rangle$ and $\langle v, u \rangle$
8: **emit** $\langle v, \mathcal{N}(v) \rangle$ ▷ where v is a node and $\mathcal{N}(v)$ is an adjacency list for the node (v)
9: **end procedure**

```
//Setting up Spark Configuration in the cluster
val Config:SparkConf = new SparkConf()
val SC = new SparkContext(Config)
//Reading the input file and convert it into adjacency list
val adjList = SC.textFile(args(0)).map(line ⇒ {
val innerLine = line.split(\t)
(innerline(0).toInt, innerline(1).toInt)
})
```

Figure 4.3. Snippet code for generating Adjacency List using Spark

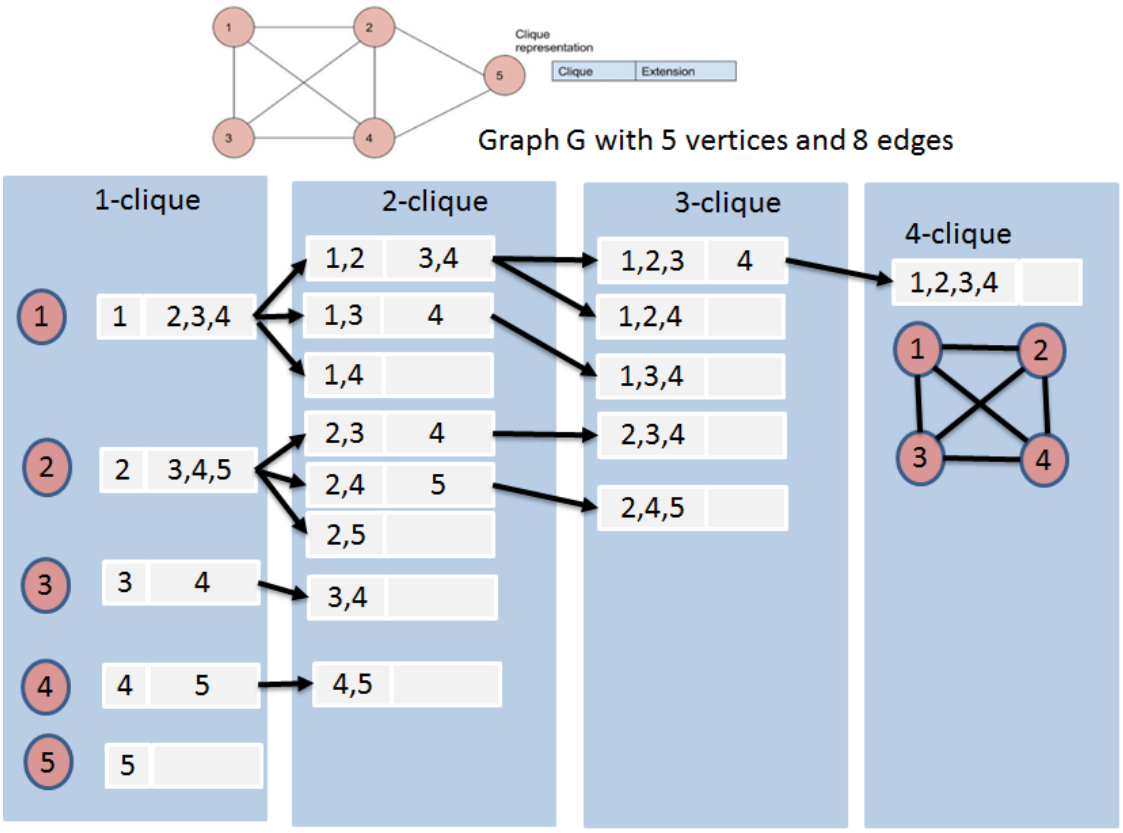

Figure 4.4. Enumeration of 4-clique for a small graph

Figure 4.3 provides the sample code snippet which produces the output as adjacency list by using map transformation that is described in the section1. In Spark, a user can provide the number of partitions so that a task can be divided across the worker nodes. This job of assigning the partitions is taken care by SparkContext which is an in-built function that acts as a driver for the program. We can also use Spark construct *RDD.repartition*, for equal distribution of clique records across the Spark cluster.

4.2.3 Adjacency List

We implemented each step of (Algorithm 1) and collected each k-clique information into the Spark RDD that can be used for further iterations of generating k-clique. (Algorithm 2) will provide adjacency list of each vertex for the input graph. This adjacency list information will be helpful in generating next l-cliques. Therefore, using map and groupbykey in-built transformation we can obtain adjacency list. Each line of the adjacency list is represented as a tuple of size 2 consisting of the node id and the neighbors associated with it.

Algorithm 3 Generate adjacency list of Graph G^d

1: **procedure** FILTERADJLIST($v, \mathcal{N}(v)$) ▷ where v is source vertex and $\mathcal{N}(v)$ is adjacency list
2: **Input** : Output from Algorithm 2
3: $filterAdjList \leftarrow \Phi$
4: **for** $i \in N(v)$ **do**
5: **if** $i > v$ **then**
6: $filterAdjList \cup i$
7: **end if**
8: **end for**
9: **emit** $\langle v, filterAdjList \rangle$ ▷ where v is a node and $filterAdjList$ is filtered adjacency list generated from $\mathcal{N}(v)$
10: **end procedure**

In Algorithm 3, we enumerated cliques of size 2 which is straightforward. After which we try to eliminate the duplicates (line no 5) by considering only nodes whose vertex id's are greater than the id of the node being considered. Since it does not consider duplicates, after this step, we will be enumerating the cliques only once. Typically, in mapreduce algorithms, we implement the reduce transformation imme-

diately after the map transformation. But it is not mandatory and hence we decided not to perform the reduce transformation step as we are not collecting any data.

Lemma 4.2.1 *Say,* $r = \langle key, value \rangle \in RDD_1$. *Then* $r.key$ *is a 1-clique and* $r.value$ *is valid extension of* $r.key$ *clique.*

Proof: Say, $u = r.key$. By construction, u is a vertex, which is a size-1 clique. $r.value$ is u's adjacency list, so each vertex $v \in r.value$ forms an edge with u. Besides, (u, v) is an edge of G^d, so $u \prec v$. Thus, the vertex v can be added with u to form a valid size-2 clique, namely (u, v). ■

Algorithm 4 Broadcasting filtered adjacency list

1: **procedure** BROADCASTADJLIST($filterAdjList$) ▷ where $filterAdjList$ is filtered adjacency list from the Algorithm 3
2: **Input** : Output from Algorithm 3
3: $localAdjList \leftarrow filterAdjList.collectAsMap()$ ▷ $localAdjList$ RDD is the cached object
4: SparkContext.broadcast($localAdjList$)
5: **end procedure**

4.2.4 Broadcasting Neighborhood Information

In the proposed distributed clique enumeration process, a clique embedding is represented as a record in RDD. Thus making the enumeration process easily distributed across all the workers in the cluster. The neighborhood information $\mathcal{N}(v)$ of the last added node v is the only piece of network topology information passed to the distributed methods (Lines 3-4 of Algorithm4). As we demonstrate in experimental results, the computational burden for clique enumeration in real world networks come from the exponential count of clique embedding and not from the input network size. Which means the clique counting algorithms often fail due to the number of embedding it needs to enumerate but not due to the size of the network. The information

regarding filtered adjacency lists of corresponding vertices will be cached and broadcasted to the other workers in the cluster by invoking the respective collectAsMap and broadcast transformations in the master node. This ensures that information is available in-memory for the workers, thereby making Spark is much more faster than Hadoop.

Starting from 3-cliques, we can repetitively apply Algorithms 5 and 6 to get RDDs with increasingly larger cliques.

Algorithm 5 Generating k-clique extensions from *(k-1)-clique*

1: **procedure** EXTENSION($(k-1)$-clique)
2: **Input file**:$(k-1)$-clique=(V,Extensions), where $(k-1)$-clique is a tuple of size 2, where $(k-1)$-clique.V represents set of nodes that are responsible for identifying $(k-1)$-clique and $(k-1)$-clique.Extensions is a set of vertices that are valid extensions for $(k-1)$-clique
3: posKCliExt $\leftarrow \Phi$
4: possibleExt $\leftarrow (k-1) - clique.Extensions$
5: **for** $i \in (k-1) - clique.Extensions$ **do**
6: **emit** $posKCliExt \cup (i,$(k-1)$-clique \cup (i), possibleExt)$
7: **end for**
8: **end procedure**

4.2.5 Clique Extension

For a record of size k clique embedding, the extension stage enumerates all intermediate extensions to give records of size $k+1$ cliques embedding (see Algorithm 5). Initially the function EXTENSION constructs an empty list *partialCliqueEmbedding* for gathering all cliques extended from the input arguments in (Algorithm 5). Enumerating k-cliques from $(k-1)$-cliques is done by the last node added to the *CompleteExtension* into the $(k-1)$-cliques (line no. 6) but this does not alter the extensions. All the new

Algorithm 6 Generating k-clique completion from $(k-1)$-clique

1: **procedure** COMPLETION($posKCliExt, localAdjList.value(posKCliExt.V)$) ▷ where $posKCliExt$=(v,clique,PossExt) is a 3-tuple generated from Algorithm 5 and localAdjList is generated from Algorithm 4

2: **Input file**: posKCliExt is a RDD from the algorithm 5 and $localAdjList.value(posKCliExt.v)$ is the filtered adjacency list for the node.

3: $CompleteExtensions \leftarrow \Phi$

4: $CompleteExtensions \leftarrow localAdjList.value(posKCliExt.v) \cap posKCliExt.PossExt$

5: **emit** $posKCliExt.clique, CompleteExtensions$

6: **if** $|posKCliExt.clique| == k$ **then**

7: **return** $posKCliExt.count()$

8: **else**

9: **repeat** Algorithm 5

10: **end if**

11: **end procedure**

records will be added to the *partialCliqueEmbedding* and the final list is returned as output which will be used to generate complete extensions of the k-cliques. The formal steps are given in (Algorithm 5).

4.2.6 Clique Completion

completion procedure (see Algorithm 6) is the responsible for re-evaluation of all the newly created records as a result of the extension step. At this stage all information necessary for clique type identification and further extensions of clique enumerations are gathered from the adjacency list $\mathcal{N}(v)$ of newly added vertex v. The function COMPLETION takes the tuple from the output of Algorithm 5 which consists of (v,clique,partialExtensions).

The steps for obtaining CompleteExtensions are described henceforth. First, we gather neighborhood information of the newly added vertex v (Line 4) from the Local Adjacency list map obtained after applying Algorithm 3. $LocalAdjacencyListRDD$ is a restricted adjacency list containing all neighboring nodes of v whose id is larger than the node considered in the embedding. $LocalAdjacencyListRDD$ is necessary to ensure the correctness of the algorithm, so that each clique is embedded only once. Complete extensions are computed by performing an intersection operation of neighborhood list and current partial extension list of vertex v obtained from (Algorithm 5). The function COMPLETION returns the complete clique embedding record (Line 5). Line 6-10 will compare the size of the clique obtained to k, if it matches then it returns the count of cliques, if it does not it will repeat the (Algorithm 5).For a better understanding, a small graph and the procedure needed to enumerate 4-clique has been provided in the Figure 4.4

Lemma 4.2.2 *Say, extension and completion operation takes* $r = \langle key, value \rangle \in RDD_{l-1}$. *Then it produces* $r' = \langle key, value \rangle \in RDD_l$ *such that* $r'.key$ *is a* l *size clique and* $r'.value$ *contain vertices which are valid extension of* $r'.key$ *clique.*

Proof: Say, $C = r.key$. By construction, C is a $(l-1)$-size clique and $r.value$ is a set of vertices which are C's valid extension, so each vertex $v \in r.value$ forms an edge with each vertex of C. Extension process takes a vertex $v \in r.value$ and adds v with C to form a clique of size l. Besides, for every vertex $u \in C$, (u, v) is an edge of G^d, so $u \prec v$. Thus, the vertex v can be added with C to form a valid size-l clique $C \cup \{v\}$, which is set as $r'.key$. Then the Completion process intersects the adjacency list of v with $r.value$ and set the intersection set as $r'.value$. Now, for any vertex $w \in r'.value$, w is adjacent to v, and it is also adjacent to all vertices of C. So, w is a valid extension for the clique $C \cup \{v\} = r'.key$. ■

Theorem 4.2.3 *For a given positive integer k and an input graph G,* KC-SPARK *enumerates all k-size cliques of G in RDD_k*

Proof: We will prove this theorem by induction on k. Using Lemma 4.2.1, RDD_1 contains all 1-size cliques with valid extension. So, the claim holds for $k = 1$.

Let's assume that the theorem holds for $k-1$, then by induction hypothesis, RDD_{k-1} holds all $(k-1)$-size cliques and their valid extension. Then, k'th iteration of KC-SPARK takes RDD_{k-1} and generates RDD_k. Using the Lemma 4.2.2, RDD_k holds cliques of size k and their valid set of extensions. Hence, proved. ■

4.3 Time Complexity Analysis

KC-SPARK enumerates all possible cliques of size k of a graph. The brute-force time complexity of enumerating fixed size (say, k) clique is $O(n^k k^2)$; where n is the number of vertices and k is the desired clique size. Here, $O(n^k)$ is the number of potential subgraphs of size k in the network, and the time complexity to analyze whether a subgraph is a clique is $O(k^2)$. Since, KC-SPARK enumerates cliques in a distributed manner for the larger input networks with multiple processors p. Hence, the time complexity of KC-SPARK with p processors available is given as $O(n^k k^2/p)$. The space complexity is $O(n)$ since Algorithm 4 collects neighbors into a Map which

will be in-memory for all the CPU's so that computations can be done in a faster manner rather than overhead in disk I/O calls.

5 EXPERIMENTS AND RESULTS

5.1 Data sets

We performed experiments on several graphs taken from the SNAP Stanford repository [53] and network repository [55]. We pre-processed the input graphs so that they are undirected, connected, and sorted according to their degree of distribution. Doing this we can achieve dynamic load balancing of the processes and eventually control the curse of the last reducer. Each line in the input represents an edge separated by the tab delimiter between the pair of vertices. In this paper, we present the experiment results for the real time web graphs (webBerkStan, webGoogle, web-baidu-baike), social graphs (soc-brightkite,socfb-CMU), infrastructure graphs (inf-road-usa,inf-roadNet-CA,inf-openflights,inf-italy-osm), and collaboration networks (ca-dblp-2012, ca-HepPh). Few main characteristics were tabulated and given in Table 1. We observed that as the number k in *k-clique* increases the *k-clique* count increased from one million to one billion as shown in the graph web-BerkStandir.

5.2 Platform

The experiments were carried out in a cluster of 10 machines and also on a stand alone machine. Our cluster consists of 10 machines each having a 14 GB RAM and 4 cores which are specially devoted for Spark jobs. So there are a total of 40 cores on our cluster. We also performed the same experiments on a single Ubuntu machine which has 8 GB primary memory (RAM) and 4 cores. Experimental results for stand alone mode and cluster mode are given in Table 1. Spark 2.0.1 and Scala 2.11.7 were

installed on all the 10 machines with the same configuration setting as in stand alone mode with 4 cores.

Experiments are performed for enumerating triangles using Hadoop mapreduce using the same cluster configuration.

5.3 Comparison to the mapreduce works [8–10]

In this section we tabulated the clique counts and compared our experiment results to benchmark data sets taken from the SNAP Stanford repository. Table5.1 provides properties such as nodes, edges, max degree and number of cliques up until size 5 of benchmark graphs taken from [53]. Our experiments were performed over our own cluster of 10 machines and each machine has been installed with Apache Spark 2.0.1. Table5.3 is the main outcome of this work, providing the execution time of our algorithm including comparisions with the works of Suri et al. [8] (triangle counting on a 1636 node cluster), Afrati et al. [9] (sub graph enumeration), and Finocchi et al. [10] (latest and fastest work on clique counting using mapreduce framework on a 16 node cluster) . Our solution ran on 10 machines with 40 cores and outperformed mapreduce solutions even though they used 16 machines with 64 processors. Detailed execution times are given in the table5.3. Running time of FFF_3 is much slower than SV from table5.3 on most of the graphs. To test our solution, we compared execution times of $ExactSpark_3$ with SV, FFF_3, AFU_3 and our solution has proved to be faster than benchmark data sets. And then we compared *4-clique* and *5-clique* execution times with the mapreduce solutions proposed by [9] and [10]. Figures5.3 and 5.4 showed the execution time comparison among KC-SPARK and mapreduce solutions [8–10].

We also compared the running times with large graphs provided in Table5.4. Our solution ran on 10 machines and outperformed mapreduce solutions even though [8–10] used a 1636 node cluster. In order to process large graphs, we require more primary memory to efficiently enumerate cliques of sizes k > 3. Figure5.5 showed

the execution time comparison of triangle counting among KC-SPARK and mapreduce solutions [8–10].

Table 5.1.
Benchmark graphs statistics: number of Nodes, number of Edges, max degree, avg degree, and number of cliques on k=3,k=4,k=5 nodes

Network	Nodes	Edges	AvgDeg	# 3-*Clique*	# 4-*Clique*	# 5-*Clique*
citPat	3,774,768	16,518,947	8	7,515,023	3,501,071	3,039,636
youtube	1,134,890	2,987,624	5	3,056,386	4,986,965	7,211,947
locGowalla	196,591	950,327	9	2,273,138	6,086,852	14,570,875
socPokec	1,632,803	22,301,964	27	32,557,458	42,947,031	52,831,618
webGoogle	875,713	4,322,051	11	13,391,903	39,881,472	105,110,267
webStan	281,903	1,992,636	16	11,329,473	78,757,781	620,210,972
asSkit	1,696,415	11,095,298	6.54	28,769,868	148,834,439	1,183,885,507

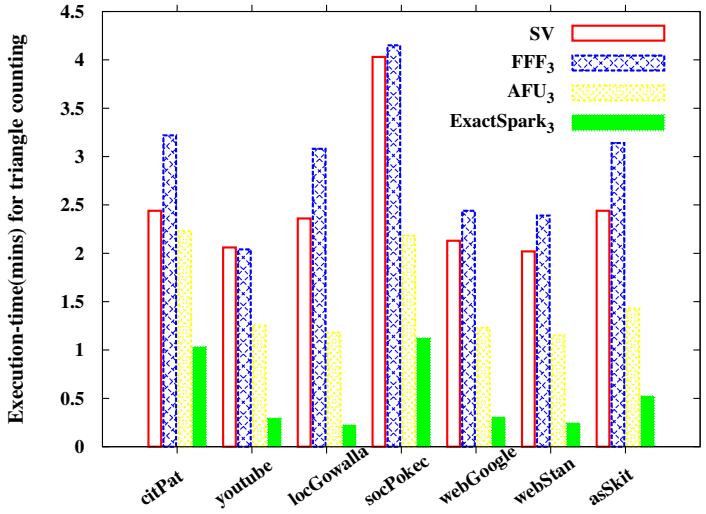

Figure 5.1. Execution time comparison of enumerating 3-clique for the solutions SV, FFF_3, AFU_3 [10] with $ExactSpark_3$

Table 5.2.
Running time comparison of KC-SPARK on a cluster of 10 machines with 40 total processors. Running times for the executions SV,FFF, and AFU in (minutes:Seconds) are taken from [10]. Comparison for k=3,4

Network	SV	FFF_3	AFU_3	$ExactSpark_3$	FFF_4	AFU_4	$ExactSpark_4$
citPat	2:44	3:22	2:23	1:03	3:11	3:11	1:08
youtube	2:06	2:04	1:25	0.29	2:39	1:41	0:32
locGowalla	2:36	3:08	1:18	0:22	3:04	1:21	0:24
socPokec	4:03	4:15	2:18	1:12	4:02	2:29	1:26
webGoogle	2:13	2:44	1:23	0:30	2:43	1:27	0:36
webStan	2:02	2:39	1:15	0:24	2:29	1:27	0:36
asSkit	2:44	3:14	1:43	0:52	3:17	2:59	1:30

Notations: (minutes:seconds)

Table 5.3.
Running time comparison of KC-SPARK on a cluster of 10 machines with 40 total processors. Running times for the executions SV,FFF, and AFU in (minutes:Seconds) are taken from [10]. For comparison we used k=5

Network	FFF_5	AFU_5	$ExactSpark_5$
citPat	3:13	2:18	1:13
youtube	2:34	1:33	0:35
locGowalla	3:02	1:30	0:28
socPokec	4:13	2:39	1:46
webGoogle	2:43	1:32	0:47
webStan	2:37	2:06	2:01
asSkit	3:18	5:34	5:48

Notations: (minutes:seconds)

Table 5.4.
Running time comparison of KC-SPARK on large graphs with the triangle counting times reported in [10] . For comparison we used k=3

Network	Nodes	Edges	$AvgClusCoeff$	# 3-$clique$
orkut	3,072,441	117,185,083	0.17	627,584,181
webBerkStan	685,230	7,600,595	0.60	64,690,980
comLiveJ	3,997,962	34,681,189	0.30	177,820,130
socLiveJ1	4,847,571	68,993,773	0.27	285,730,264

Table 5.5.
Running time comparison of KC-SPARK on large graphs with the triangle counting times reported in [10] . For comparison we used k=3

Network	SV	FFF_3	AFU_3	$ExactSpark_3$
orkut	30:07	24:00	8:21	14:53
webBerkStan	2:28	3:00	1:37	0:51
comLiveJ	5:31	5:31	2:53	3:21
socLiveJ1	6:36	6:33	3:14	5.07

Notations: Running time (minutes:seconds)

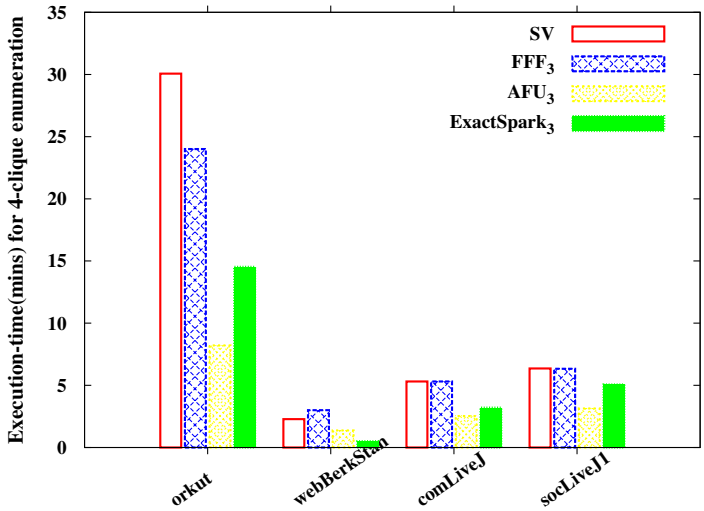

Figure 5.2. Execution time comparison of enumerating 3-clique on large graphs for the solutions SV, FFF_3, AFU_3 [10] with $ExactSpark_3$

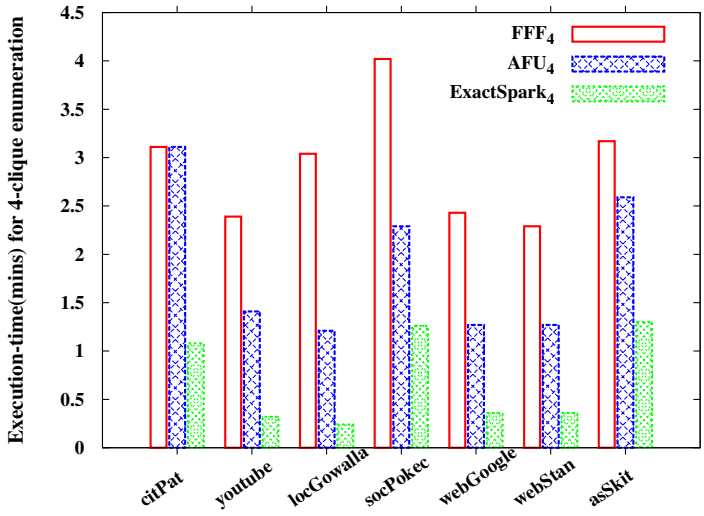

Figure 5.3. Execution time comparison of enumerating clique-4 for the solutions FFF_4, AFU_4 [10] with $ExactSpark_4$

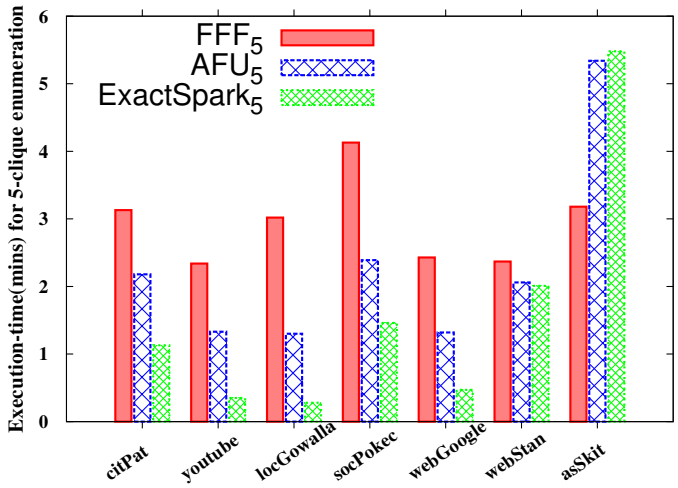

Figure 5.4. Execution time comparison of enumerating clique-5 for the solutions FFF_5, AFU_5 [10] with $ExactSpark_5$

5.4 Comparison of enumerating 3-clique between Hadoop mapreduce and KC-SPARK with the same cluster setup

We implemented enumaration of triangles in a network using Hadoop mapreduce with the same cluster setup and compared with the 3-clique from KC-SPARK . Table 5.6 provides the detailed comparison of experiment results.

5.5 Analysis of speedup of KC-SPARK between stand alone mode and cluster mode

We enumerated *k-cliques*, where the input size k=5 for all the graphs listed in Table5.8 using iterative Spark framework. The clique count results have been tabulated for input graphs as shown in Table 5.8. The number of cliques increased exponentially for the social networks socfb-CMU and soc-brightkite from 3-clique to 5-clique. In contrast, the number of cliques decreased for the infrastructure networks inf-roadNet-CA, inf-italy-osm etc. We observed that for large graphs like web-BerkStan-dir, the

Table 5.6.
Running time comparison of KC-SPARK on a cluster of 10 machines with 40 total processors. Running times for the executions Hadoop mapreduce and KC-SPARK in (minutes:Seconds). For comparison, we used k=3

Network	Nodes	Edges	# 3-*Clique*	mapreduceTime (min:sec)	$kc-spark$ (min:sec)
citPat	3,774,768	16,518,947	7,515,023	9:42	1:03
youtube	1,134,890	2,987,624	3,056,386	3:36	0:29
locGowalla	196,591	950,327	2,273,138	3:00	0:22
socPokec	1,632,803	22,301,964	32,557,458	22:42	1:12
webGoogle	875,713	4,322,051	13,391,903	4:24	0:30
webStan	281,903	1,992,636	11,329,473	4:01	0:24
asSkit	1,696,415	11,095,298	28,769,868	11:12	0:52
orkut	3,072,441	117,185,083	627,584,181	379:36	14:53
webBerkStan	685,230	7,600,595	64,690,980	11:37	0:51
comLiveJ	3,997,962	34,681,189	177,820,130	40:64	3:21
socLiveJ1	4,847,571	68,993,773	285,730,264	58:42	5:07

enumeration of cliques took longer duration and partitioned the input network into 400 partitions across all 40 cores.

5.5.1 Scalability and Performance evaluations with increasing # CPUs

Since Spark algorithms are parallel in nature, a simple question always asked is how their execution times are affected by the number of CPU's used i.e. the number of cores used. Figure5.5 and Figure5.6 depict the execution time of the Spark program in stand alone mode with 4 cores and in cluster mode with 40 cores. We have demonstrated experiments for cliques of size 5, where k=5 in all the executions ranging from small graphs to large graphs. Scalability can be obtained by increasing

Table 5.7.
Benchmark graphs statistics: number of Nodes, number of Edges, max degree, avg degree, and number of cliques on k=3,k=4,k=5 nodes

Network	Nodes	Edges	Max	# 3-*Clique*	#5-*Clique*
inf-openflights	2,939	15,677	242	72852	875543
ia-email-EU-dir	265,009	364,481	8K	267,313	1,101,520
inf-roadNet-CA	1,957,027	2,760,388	12	120,492	40
soc-brightkite	56,739	212,945	1,134	494,408	19.4M
web-google-dir	876K	5M	6K	13.3M	105M
socfb-CMU	6,621	249,959	840	32,788,398	1.4T
web-baidu-baike	2M	18M	98K	25M	24M
ca-dblp-2012	317,080	1,049,866	343	2,224,385	262M
inf-road-usa	23,947,347	28,854,312	9	50,135,113	384M
ca-HepPh	11,204	117,619	491	3,357,890	6.4B
web-BerkStan-dir	617,094	7,600,595	84K	64.7M	21.8B

Notations: K = 1000, M = 1000K, B = 1000M, T = 1000B,

Table 5.8.
Running time comparison of KC-SPARK on a cluster of 10 machines with 40 cores and standalone mode with 1 machine and 4 cores . For comparison we used k=5

Network	Nodes	Edges	Max	STANDALONE	CLUSTER
inf-openflights	2,939	15,677	242	13	17
ia-email-EU-dir	265,009	364,481	8K	26	23
inf-roadNet-CA	1,957,027	2,760,388	12	78	33
soc-brightkite	56,739	212,945	1,134	86	26
web-google-dir	876K	5M	6K	351	47
socfb-CMU	6,621	249,959	840	387 (6.45m)	42
web-baidu-baike	2M	18M	98K	982 (16.36m)	95 (1.58)m)
ca-dblp-2012	317,080	1,049,866	343	1,333 (22m)	99 (1.65m)
inf-road-usa	23,947,347	28,854,312	9	3190 (53.16m)	179 (2.95m)
ca-HepPh	11,204	117,619	491	63,716 (17.69h)	3,258 (54.3m)
web-BerkStan-dir	617,094	7,600,595	84K	$> 39h$	6420 (107m)

$K = 1000$, $M = 1000K$, $B = 1000M$, $T = 1000B$, $m \rightarrow min$, $h \rightarrow hour$ and $d \rightarrow day$

the number of CPUs in the cluster. Due to the broadcasting and caching mechanism exhibited by Spark, the performance increased significantly on the cloud as opposed to the execution time for the real time graph ca-dplp-2012

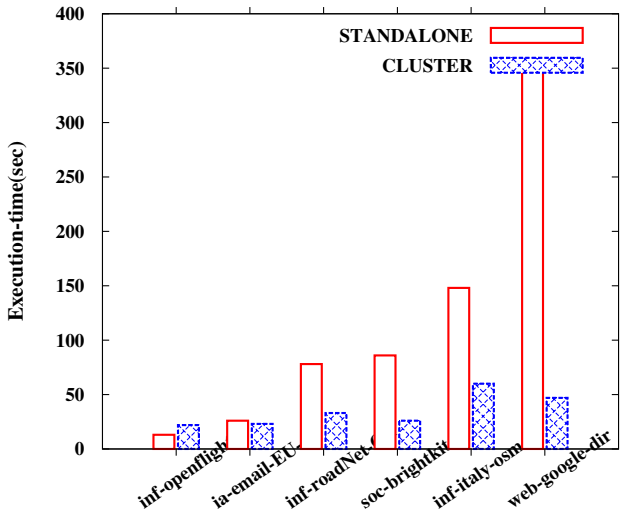

Figure 5.5. Execution time comparison between STAND ALONE mode and CLUSTER mode of KC-SPARK for medium size graphs

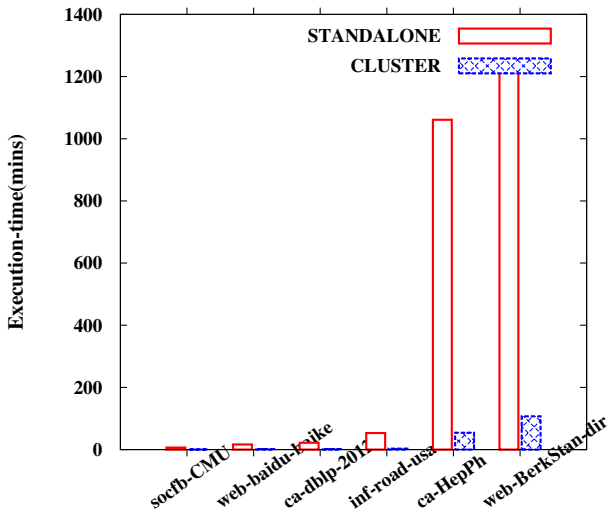

Figure 5.6. Execution time comparison between STAND ALONE mode and CLUSTER mode of KC-SPARK for large graphs

6 FUTURE WORK AND CONCLUSION

6.1 Future work

1. In this work, KC-SPARK has been implemented using the Spark distributed framework. For enumerating cliques efficiently on larger graphs, KC-SPARK can be extended to run on a greater number of machines with more primary memory due to the fact that Spark has been proven to work 100x faster in-memory compared to Hadoop mapreduce.

2. The KC-SPARK algorithm can be modified to enumerate maximal and maximum cliques on a network efficiently. To the best of my knowledge, there is no work proposed on maximal and maximum clique using the Spark framework. Maximal cliques can be obtained by clique decomposition of a graph. For maximum clique, we need to perform iterations with k value to be maximum degree in the Graph G^d. For maximal cliques, shrink the graph by removing all the edges from the input graph once we obtain maximal cliques of respective size till the k value to be 1.

3. Approximation algorithms can be proposed using the KC-SPARK approach which will be helpful for enumerating cliques if the graph size is too large.

4. KC-SPARK can be extended to work with multiple CPUs and GPUs. Since GPUs provide faster computation when compared to CPUs, we can change the KC-SPARK design accordingly for time efficient results.

6.2 Conclusion

In this work, we propose a KC-SPARK , an efficient distributed clique enumeration algorithm for large real-life networks using the Apache Spark distributed framework. This method harnesses the power of a distributed computing paradigm to give a scalable clique enumeration and counting mechanism in large real-life networks. Experiment results show that KC-SPARK can enumerate k-cliques from very large real-life networks, where a single commodity machine cannot produce the desired result in a reasonable amount of time. In our experiment results, enumerating 5-cliques for the network ca-HepPh in a single machine with a quad-core processor took approximately 18 hours. In contrast, the same computation took 53 minutes when executed in cluster of 10 machines using KC-SPARK . We also compared our solution with the mapreduce frameworks proposed by Finocchi et al. [10] and Afrati et al. [9] for clique enumeration. Experimental results show that KC-SPARK is 80-100 percent times faster in running times than both of these methods. On the other hand, we also compared with the triangle enumeration using mapreduce with the same cluster setup. Our results show that running times of KC-SPARK are 8-10x faster than Hadoop mapreduce implementation. Moreover, our system is fully fault-tolerant due to the inherent features of the Spark framework. If there is an arbitrary failure in the worker node, that particular node computation can be recomputed from the original fault tolerant RDD using the Apache Spark lineage property.

We conducted many experiments to show that KC-SPARK efficiently runs on any cloud platform using the Spark in-memory cluster computing framework. We have shown that KC-SPARK is fast, distributed, and scalable because it can handle large graphs that cannot be computed in a reasonable amount of time with a single commodity machine. In order to process larger graphs, KC-SPARK merely requires more primary memory to efficiently enumerate cliques.

REFERENCES

REFERENCES

[1] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proc. of the 11th ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, 2005.

[2] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998.

[3] Albert-Laszlo Barabasi and Reka Albert. Emergence of Scaling In Random Networks. *Science*, 286:509–512, October 1999.

[4] Lars E. Rasmussen. Approximately counting cliques. Technical report, Berkeley, CA, USA, 1996.

[5] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, October 2006.

[6] Stephen P. Borgatti, Ajay Mehra, Daniel J. Brass, and Giuseppe Labianca. Network analysis in the social sciences. *Science*, 323:892–895), 2009.

[7] Shweta Jain and C. Seshadhri. A fast and provable method for estimating clique counts using turán's theorem. *CoRR*, abs/1611.05561, 2016.

[8] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, pages 607–614, 2011.

[9] Foto N. Afrati, Dimitris Fotakis, and Jeffrey D. Ullman. Enumerating subgraph instances using map-reduce. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 62–73, Washington, DC, USA, 2013. IEEE Computer Society.

[10] Irene Finocchi, Marco Finocchi, and Emanuele G. Fusco. Counting small cliques in mapreduce. *CoRR*, abs/1403.0734, 2014.

[11] N. Przulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[12] Tijana Milenkovic and Nataa Przulj. Uncovering biological network function via graphlet degree signatures. *Cancer Inform*, 6:257–273, 2008.

[13] M. Rahman, M. A. Bhuiyan, and M. Al Hasan. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2466–2478, Oct 2014.

[14] Mahmudur Rahman, Mansurul Alam Bhuiyan, Mahmuda Rahman, and Mohammad Al Hasan. Guise: a uniform sampler for constructing frequency histogram of graphlets. *Knowledge and information systems*, 38(3):511–536, 2014.

[15] Charalampos E. Tsourakakis, U Kang, Gary L. Miller, and Christos Faloutsos. DOULION: Counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 837–846, 2009.

[16] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *Proceedings of the 15th IEEE International Conference on Data Mining (ICDM)*, pages 1–10, 2015.

[17] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgios Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph pattern mining. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Oct 2015.

[18] D. Marcus and Y. Shavitt. Efficient counting of network motifs. In *2010 IEEE 30th International Conference on Distributed Computing Systems Workshops*, pages 92–98, June 2010.

[19] Andrei Todor, Alin Dobra, and Tamer Kahveci. Counting motifs in probabilistic biological networks. In *Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB '15, pages 116–125, New York, NY, USA, 2015. ACM.

[20] Pinghui Wang, John C. S. Lui, Bruno Ribeiro, Don Towsley, Junzhou Zhao, and Xiaohong Guan. Efficiently estimating motif statistics of large networks. *ACM Trans. Knowl. Discov. Data*, 9(2):8:1–8:27, September 2014.

[21] Holger Dinkel, Kim Van Roey, Sushama Michael, Norman E Davey, Robert J Weatheritt, Diana Born, Tobias Speck, Daniel Krger, Gleb Grebnev, Marta Kuban, Marta Strumillo, Bora Uyar, Aidan Budd, Brigitte Altenberg, Markus Seiler, Luca B Chemes, Juliana Glavina, Ignacio E Snchez, Francesca Diella, and Toby J Gibson. The eukaryotic linear motif resource elm: 10 years and counting. *Nucleic acids research*, 42(Database issue):D25966, January 2014.

[22] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proc. of the conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '99, pages 251–262, 1999.

[23] Yu Chen and Gordon M. Crippen. A novel approach to structural alignment using realistic structural and environmental information. *Protein Science*, 14(12):2935–2946, 2005.

[24] I.J. Farkas B. Adamcsek, G. Palla and T. Vicsek. Cfinder: locating cliques and overlapping modules in biological networks. *Bioinformatics*, 22:1021–1023, 2006.

[25] Jiérâi Vlach and Kishore Singhal. *Computer Methods for Circuit Analysis and Design*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1993.

[26] Hiroo Saito, Masashi Toyoda, Masaru Kitsuregawa, and Kazuyuki Aihara. A large-scale study of link spam detection by graph algorithms. In *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web*.

[27] Masahiro Hattori, Yasushi Okuno, Susumu Goto, and Minoru Kanehisa. Development of a chemical structure comparison method for integrated analysis of chemical and genomic information in the metabolic pathways. *Journal of the American Chemical Society*, 125(39):11853–11865, 2003. PMID: 14505407.

[28] R. Baeza-Yates and B.Ribeiro-Neto. Modern information retrieval. *ACM press New York*, 1999.

[29] Mohammed J Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. Technical report, Rochester, NY, USA, 1997.

[30] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, September 1973.

[31] F. Cazals and C. Karande. A note on the problem of reporting maximal cliques. *Theor. Comput. Sci.*, 407(1-3):564–568, November 2008.

[32] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

[33] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Generating all maximal independent sets: Np-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, 1980.

[34] David Eppstein, Maarten Löffler, and Darren Strash. *Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time*, pages 403–414. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[35] David Eppstein and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. In *Proceedings of the 10th International Conference on Experimental Algorithms*, SEA'11, pages 364–375, Berlin, Heidelberg, 2011. Springer-Verlag.

[36] Ina Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theor. Comput. Sci.*, 250(1-2):1–30, January 2001.

[37] David S. Johnson and Christos H. Papadimitriou. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, March 1988.

[38] Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. pages 260–272. Springer-Verlag, 2004.

[39] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[40] Immanuel M Bomze, Marco Budinich, Panos M Pardalos, and Marcello Pelillo. The maximum clique problem. In *Handbook of combinatorial optimization*, pages 1–74. Springer, 1999.

[41] Randy Carraghan and Panos M Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9(6):375–382, 1990.

[42] Patric RJ Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1):197–207, 2002.

[43] Panos M Pardalos and Jue Xue. The maximum clique problem. *Journal of global Optimization*, 4(3):301–328, 1994.

[44] Roberto Battiti and Marco Protasi. Reactive local search for the maximum clique problem 1. *Algorithmica*, 29(4):610–637, 2001.

[45] Qingfu Zhang, Jianyong Sun, and Edward Tsang. An evolutionary algorithm with guided mutation for the maximum clique problem. *IEEE Transactions on Evolutionary Computation*, 9(2):192–200, 2005.

[46] Panos M Pardalos and Gregory P Rodgers. A branch and bound algorithm for the maximum clique problem. *Computers & operations research*, 19(5):363–375, 1992.

[47] Luana E Gibbons, Donald W Hearn, Panos M Pardalos, and Motakuri V Ramana. Continuous characterizations of the maximum clique problem. *Mathematics of Operations Research*, 22(3):754–768, 1997.

[48] J. Robson. Finding a maximum independent set in time O(2 (n/4)), January 2001.

[49] Virginia Vassilevska. Efficient algorithms for clique problems. *Inf. Process. Lett.*, 109(4):254–257, January 2009.

[50] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1361–1370, New York, NY, USA, 2010. ACM.

[51] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[52] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[53] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[54] jaceklaskowski. Spark architecture.

[55] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.