A DISTRIBUTED GRAPH MINING FRAMEWORK BASED ON MAPREDUCE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF COMPUTER ENGINEERING DEPARTMENT
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SERTAN ALKAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY 2010

Approval of the thesis:

A DISTRIBUTED GRAPH MINING FRAMEWORK BASED ON MAPREDUCE

submitted by **SERTAN ALKAN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Müslim Bozyiğit _____
Head of Department, **Computer Engineering**

Assist. Prof. Dr. Tolga Can _____
Supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Prof. Dr. Volkan Atalay
Computer Engineering Department, METU _____

Assist. Prof. Dr. Tolga Can
Computer Engineering Department, METU _____

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Uğur Doğrusöz
Computer Engineering Deparment, Bilkent University _____

Dr. Cevat Şener
Computer Engineering Department, METU _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: SERTAN ALKAN

Signature :

ABSTRACT

A DISTRIBUTED GRAPH MINING FRAMEWORK BASED ON MAPREDUCE

Alkan, Sertan
M.S., Department of Computer Engineering
Supervisor : Assist. Prof. Dr. Tolga Can

January 2010, 47 pages

The frequent patterns hidden in a graph can reveal crucial information about the network the graph represents. Existing techniques to mine the frequent subgraphs in a graph database generally rely on the premise that the data can be fit into main memory of the device that the computation takes place. Even though there are some algorithms that are designed using highly optimized methods to some extent, many lack the solution to the problem of scalability. In this thesis work, our aim is to find and enumerate the subgraphs that are at least as frequent as the designated threshold in a given graph. Here, we propose a new distributed algorithm for frequent subgraph mining problem that can scale horizontally as the computing cluster size increases. The method described here, uses a partitioning method and Map/Reduce programming model to distribute the computation of frequent subgraphs. In the core of this algorithm, we make use of an existing graph partitioning method to split the given data in the distributed file system and to merge and join the computed subgraphs without losing information. The frequent subgraph computation in each split is done using another known method which can enumerate the frequent patterns. Although current algorithms can efficiently find frequent patterns, they are not parallel or distributed algorithms in that even when they partition the data, they are designed to work on a single machine. Furthermore, these algorithms are computationally expensive but not fault tolerant and are not designed to work on

a distributed file system. Using the Map/Reduce paradigm, we distribute the computation of frequent patterns to every machine in a cluster. Our algorithm, first bi-partitions the data via successive Map/Reduce jobs, then invokes another Map/Reduce job to compute the subgraphs in partitions using CloseGraph, recovers the whole set by invoking a series of Map/Reduce jobs to merge-join the previously found patterns. The implementation uses an open source Map/Reduce environment, Hadoop. In our experiments, our method can scale up to large graphs, as the graph data size gets bigger, this method performs better than the existing algorithms.

Keywords: frequent subgraph mining, graph partitioning, closed pattern, MapReduce, distributed computing

ÖZ

EŞLE/İNDİRGE YÖNTEMİ ÜZERİNE KURULU DAĞITIK BİR AĞ MADENCİLİĞİ GERÇEKLEMESİ

Alkan, Sertan
Yuksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Yrd. Doç. Dr. Tolga Can

January 2010, 47 sayfa

Bir ağ içinde saklı olan ve sık geçen desenler incelenen bu ağ için önemli bilgiler sunar. Bir ağ veritabanındaki sık geçen daha küçük ağları bulmak için varolan teknikler genelde tüm verinin tek bir makina belleğine sığacağını göz önüne alarak tasarlanmışlardır. Her ne kadar belli bir miktara kadar oldukça eniyilenmiş yöntemler olsa da, ölçeklenebilme problemine çözüm geliştirememişlerdir. Bu tez çalışmasında, amacımız verilen bir ağ içinde en az daha önceden belirlenen eşik değerde tekrarlanan küçük ağları bulmaktır. Burda, sık geçen desenleri bulabilmek için, çalıştırılan küme boyutu ile orantılı biçimde ölçeklenebilen dağıtık bir yöntem sunuyoruz. Bu yöntemin merkezinde, dağıtık dosya sistemi üzerine veriyi bölmek için ve hesaplanan desenleri bilgi kaybetmeden birleştirmek için varolan bir ağ parçalama yöntemini kullanır. Sık desenlerin her parçada bulunması bilinen bir başka yöntemi kullanarak yapılır. Bilinen diğer yöntemler sık geçen desenleri bulabildikleri halde, veriyi parçalasalar bile tek bir makinada çalışmak üzere tasarlandıklarından dağıtık yöntemler değillerdir. Üstelik, bu yöntemler hesaplama yoğunluklu olmalarına rağmen hata toleransları yoktur ve hiçbiri dağıtık bir dosya sistemi üzerinde çalışmak üzere tasarlanmamıştır. Sunulan yöntemde ise, Eşle/İndirge yöntemini kullanarak, desenlerin hesaplanması kümedeki her makina üzerine dağıtılmaktadır. Yöntem; önce, ardışık gönderilen ve her seferinde veriyi birbirine en az bağlı olan iki düğüm

setine ayıran eşle/indirge işleriyle veriyi böler, bir başka eşle/indirge işiyle her bir parçadaki desenleri hesaplar ve bulunan desenleri birleştirip tüm ağ içindeki desenleri bulabilmek için ardışık eşle/indirge işleri gönderir. Gerçekleme için açık kaynak kodlu bir eşle/indirge ortamı, Hadoop, kullanıldı. Deneylerde, yöntemin büyük ağlara ölçeklenebildiği ve veri miktarı büyüdükçe varolan yöntemlerden daha yüksek başarımla çalıştığı gözlemlendi.

Anahtar Kelimeler: ağlarda desen madenciliği, ağ parçalama, kapalı desen, eşle/indirge, dağıtık programlama

To my family

ACKNOWLEDGMENTS

I want to start by thanking my supervisor Asst. Prof. Dr. Tolga Can. His support and advice were invaluable during the course of this thesis.

I want to thank all my friends and colleagues, especially to Doğacan Güney, Gencay Evirgen and Enis Söztutar, for their constant help and understanding throughout the writing of this thesis.

I would also like to thank all the jury members for their insightful comments and suggestions on this thesis.

Finally, I would like to thank my family for all the encouragement they provided. Without them, this thesis would never be finished.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ vi

DEDICATON viii

ACKNOWLEDGMENTS ix

TABLE OF CONTENTS x

LIST OF TABLES xiii

LIST OF FIGURES xiv

CHAPTERS

1 INTRODUCTION 1

1.1 Problem Definition and Motivation 1

1.2 Related Work 2

1.2.1 Candidate Generation-And-Test Approaches 3

1.2.2 Pattern Growth Models 4

1.2.3 Partition Based Methods 6

1.2.4 Distributed Approaches 7

1.3 Overview Of The Proposed Method 8

1.4 Contributions 9

1.5 Thesis Outline 10

2 BACKGROUND 11

2.1 Graph Preliminaries 11

2.1.1 Graphs 11

2.1.2 Frequent Patterns 12

2.1.3 Closed Patterns 13

2.2 Biology Preliminaries . . . 13
2.2.1 Proteins . . . 13
2.2.2 Protein-Protein Interaction Networks . . . 14
2.3 Distributed Programming Preliminaries . . . 15
2.3.1 Distributed File System: GFS and HDFS . . . 16
2.3.2 MapReduce . . . 17
2.3.3 Distributed Key-Value Store: BigTable and HBase . . . 19
3 MATERIALS AND METHODS . . . 23
3.1 Data Sets . . . 23
3.1.1 Protein-Protein Interaction Networks . . . 23
3.1.2 Gene Ontologies . . . 24
3.2 Computing Environment . . . 25
3.3 Overview . . . 25
3.4 Representation Of Graphs . . . 26
3.5 Dividing The Graph Into Partitions: *MRGraphPart* . . . 27
3.6 Calculation Of Subgraphs . . . 30
3.6.1 CloseGraph Computation . . . 31
3.6.1.1 DFS Notation . . . 31
3.6.1.2 Lexicographic Ordering . . . 32
3.6.1.3 Rightmost Extension . . . 33
3.6.1.4 Early Termination . . . 33
3.7 Merge-Join Operation . . . 34
3.7.1 A Simple Tree Scheduler . . . 35
4 RESULTS . . . 37
4.1 Local Experiments . . . 38
4.2 Distributed Runs . . . 40
4.3 The Limit Of Parallelism . . . 41
5 CONCLUSION AND FUTURE WORK . . . 43
5.1 Conclusions . . . 43
5.2 Future work . . . 44

REFERENCES 46

LIST OF TABLES

TABLES

Table 2.1 Logical View Of A HBase/BigTable Table 21

Table 2.2 Physical View Of A HBase/BigTable Table (1) 21

Table 2.3 Physical View Of A HBase/BigTable Table (2) 21

Table 2.4 Physical View Of A HBase/BigTable Table (3) 22

Table 3.1 A Sample Graph Table. 27

LIST OF FIGURES

FIGURES

Figure 2.1 An example of a very simple directed labelled graph. 12
Figure 2.2 A 3D structure of myoglobin protein showing coloured alpha helices. . . . 14
Figure 2.3 Protein-Protein Interaction Network of yeast organism. 15
Figure 2.4 HDFS Architecture. 17
Figure 2.5 MapReduce architecture and execution sequence. 19

Figure 3.1 Overview of our proposed algorithm (MR Graph Mine). 26
Figure 3.2 GraphPart partitioning method illustrated. 28
Figure 3.3 A sample graph and its possible DFS trees. 32
Figure 3.4 Early termination of rightmost extension. 34

Figure 4.1 Runtime vs graph size. 38
Figure 4.2 Runtime vs Min Support. 39
Figure 4.3 Runtime vs Cluster Size. 40
Figure 4.4 Runtime vs Cluster Size in relation to #Partitions. 41

CHAPTER 1

INTRODUCTION

1.1 Problem Definition and Motivation

Graphs are widely used data structures that represent objects and relations ranging from web pages, maps, social networks to biological networks in a cell. The patterns embedded in these graphs are subject to many interesting problems. The patterns in a web graph can reveal high level clustering and category information between the web pages, the ones embedded in a social network can indicate the friend cliques whereas the patterns found in a power grid can show the cluster of the flow of consumption of the power resource. In biological networks, on the other hand, the frequent subgraphs can help understand different functions and relations. In a Protein-Protein Interaction Network (PPI), a frequently repeated pattern could help identify the unknown function of a protein by looking at the pattern it resides in. In a Gene Regulatory Network, the patterns might prove useful to understanding the function of a gene expression and the conditions of its inhibition and activation.

Frequent pattern mining is the process of choosing subgraphs that at least exceed a certain support threshold from graph data set. It is the base for the research studies in graph clustering, graph classification, graph query and restricted graph mining. However, traditional mining methods that are based on non-structured data such as frequent-item set mining and sequential-pattern mining are different from data mining approaches that work on semi-structured and structured data such as graph. Graphs are inherently more complex data types to process compared to other structures like trees or lists.

In this thesis, our goal is to devise a scalable and distributed algorithm to extract the significant frequent patterns from a data set of labelled graphs. Given a graph dataset D =

$\{G_1, G_2, ..., G_N\}$ $support(g)$ denotes the total number of times the subgraph g exists in any of the graphs $G_i \in D$. The problem of frequent subgraph mining is to find any subgraph g s.t. $support(g) >= MIN\_SUPPORT$. In our study, however, we search for patterns in a graph rather than a set of graphs, as it is a more basic problem.

There are a number of methods to find subgraphs with a support higher than a designated threshold in a given graph database. All these methods use a similar process where finding repeated patterns in a graph data structure is divided into two distinct problems;

1. Subgraph generation: Enumerating the candidate patterns and pruning the unpromising ones from the search space.

2. Graph isomorphism: Calculating the support of a subgraph by counting the repeated occurences of it.

Mining large graph patterns is challenging due to exponential number of possible different subgraphs. The running time performance of the above two directly effects the overall performance of the algorithm. Many of the methods that will be described here, tries to reduce the search space by avoiding the generation of the patterns whose support will be expected to be less than the given threshold. In our study, instead of finding all subgraphs we propose to mine the *closed frequent graph patterns*[23]. A subgraph is called a closed pattern if there are no supergraphs of that pattern with the same support in the database. Although our method can be plugged with a different subgraph extraction algorithm, we find that in most cases non-closed patterns may reveal the same level of information and trying to find closed patterns does not only dramatically reduces the search space but it also substantially increases the efficiency of the mining.

1.2 Related Work

The study on efficient frequent pattern mining in graphs are generally based on two distinct approaches. The first method extends Apriori-based candidate generation-and-test approach to graph mining. The detailed algorithms of this approach use vertices, edges or edge-disjoint paths. The second, on the other hand, adopts a pattern-growth process by growing patterns from a single graph directly. In addition to these two, some algorithms try to iterate over

the search space by partitioning the graph where the general idea is to be able to reconstruct the graph with little or no data loss. In conjuction with above algorithms some distributed methods are also suggested to enable processing of large graph sets.

1.2.1 Candidate Generation-And-Test Approaches

One interesting approach is taken by *Kuramochi and Karypis* when they devised a scalable heuristic algorithm called *GREW*[11]. GREW discovers frequent subgraphs in an iterative fashion. During each iteration, GREW identifies vertex-disjoint embeddings of subgraphs that were determined to be frequent in previous iterations and merges certain subgraphs that are connected to each other via one or multiple edges. This iterative frequent subgraph merging process continues until there are no such candidate subgraphs whose combination will lead to a larger frequent subgraph. The method's efficiency is that it maintains the location of the embeddings of the previously identified frequent subgraphs by rewriting the input graph. As a result of this graph rewriting, the vertices involved in each particular embedding are collapsed together to form a new vertex (referred to as multi-vertex), whose label uniquely identifies the particular frequent subgraph that is supported by them.

This type of graph mining has been applied to the problem of graph isomorphism as well. In 2004, Cordella *(et al.)* implemented *VF2*[17] to check the equivalence of two graphs using state space representation *(SSR)* in large graphs. This representation of mappings along with the five feasibility rules introduced, allowed them to prune the search tree.

Kuramochi and Karypis later expanded their study on graphs with the introduction of *FSG*[10]. The algorithm finds frequent subgraphs using the level-by-level expansion strategy adopted by Apriori and uses a sparse graph representation to minimize both storage and computation. It works by increasing the size of frequent subgraphs by adding one edge at a time, allowing it to generate the candidates relatively efficiently. In order to be able to scale to large data sets, it incorporates some optimizations for candidate generation and frequency counting. FSG uses a canonical labelling method to dodge having to resort to computationally expensive graph and subgraph isomorphism. Authors further picked up their study on FSG and developed two better performing algorithms $HSIGRAM^2$ and *VSIGRAM*[12]. In both algorithms, the frequent patterns are conceptually organized in a form of a lattice that is referred to as the lattice of frequent subgraphs. The k^{th} level of this lattice contains all frequent subgraphs with k edges

(i.e., $size - k$ subgraphs), and a node at level k, representing a subgraph G_k, is connected to at most k nodes at level $k-1$, each corresponding to a distinct connected $size - (k-1)$ subgraph of G_k. The goal of both HSIGRAM and VSIGRAM is to identify the various nodes of this lattice and the frequency of the associated subgraphs. The difference between them, however, is the method they use to generate the nodes of the lattice. HSIGRAM follows a horizontal approach and discovers the nodes in a breadth-first fashion, whereas VSIGRAM follows a vertical approach and discovers the nodes in a depth-first fashion.

Kuramochi and Karypis have also studied on geometric graphs and developed *gFSG*[13], a computationally efficient algorithm for finding frequent patterns corresponding to geometric subgraphs in a large collection of geometric graphs. gFSG is an incremental improvement over their previous method, FSG and it is able to discover geometric subgraphs that can be rotation, scaling, and translation invariant, and it can accommodate inherent errors on the coordinates of the vertices. They used a different topological mechanism to check the isomorphism of two graphs by identifying the possible set of geometric transformations that map the vertices of one graph within an r distance of the vertices of another, and then checking each one of them to see if it preserves the topology (and the vertex and edge labels) of the two graphs.

Zeng *(et al.)* proposed another closed pattern discovery method for quasi-cliques, called *Cocain*[25]. The method employs a labelling scheme for graphs and several pruning mechanism to reduce the search space. In order to compute the valid extensible candidates for a subgraph instance, the set of neighbouring vertices of the subgraph is built and this set is refined based on the combination pruning technique. Then each discovered extensible candidate set is conjoined to get the global extensible candidates. Late pruning techniques of vertex connectivity pruning, critical connectivity pruning and subgraph connectivity pruning is applied to get the final set of valid extensible candidates. From this set, the vice frequent extensible labels are calculated. For each such label, the algorithm discovers the descendant of the subgraph and checks whether the candidate is closed.

1.2.2 Pattern Growth Models

One of the most widely used algorithms, *gSpan*[24], uses a lexicographic ordering method, called *DFS*, to overcome the difficulty of testing graph isomorphism. By defining a right-

most extension rule, this algorithm builds a linear code for a labelled graph, unique in a graph dataset. The method maps each graph to a minimum DFS code building a novel lexicographic ordering among these codes and constructing a search tree based on this lexicographic order. GSpan starts its search over this tree repeatedly for every subgraph which has the pattern *(A, a, B) where A,B* $\in$ {*set of vertex labels*} *and a* $\in$ {*set of edge labels*} and counts the number of occurrences of this graph by looking at the children of a specific node in the search tree.

A year later, the authors of *gSpan*, Yan and Han, extended their implementation by developing the algorithm called *CloseGraph*[23] which basically searches for closed patterns. As noted on their paper, their experiments show that significant patterns are often closed ones. The intuition behind this algorithm states that if the supergraph of a frequent pattern has the same support as the frequent pattern itself, then the supergraph is the cause of frequency rather than the smaller subgraph. By this logic, this algorithm uses the same foundation in *gSpan* but prunes the search space as it walks through the minimum DFS code tree. In this thesis, we use this method to calculate the patterns in the partitions of the main graph.

Later, M.Koyutürk *(et al.)* have developed another approach called *Biological Pathway Mining*[9]. This method is tailored towards the biological networks as it searches for the frequent metabolic pathways. Mining metabolic pathways helps discover common motifs of enzyme interactions that are related to each other. Pathways are modelled with simple directed graphs that are capable of capturing the interaction information efficiently. Upon each invocation, the recursive algorithm tries to extend the edgeset (subgraph) by all edges in the candidate set one by one. If the extended edgeset is frequent, then the procedure is again invoked for the extended edgeset. The algorithm stops whenever an edgeset cannot be extended further.

Aimed at the same problem, Peng *(et al.)* published *MaxFP*[18] algorithm in 2008 to find the closed frequent patterns in biological networks. *MaxFP* transforms metabolic pathways to a group of connectivity graphs and creates the FP-tree for these graphs, thereby transforming the mining frequent pattern problem in graph database to a mining FP-tree problem. Then it constructs their conditional pattern bases by identifying the initial suffix pattern and creates the condition FP-tree for the conditional pattern base and mine this tree recursively. Generation of the frequent pattern is done by mining the condition FP-tree. The patterns are grown by connecting the suffix pattern to the frequent patterns.

In 2008, Wu and Chen proposed *FSMA*[21] method which uses incidence matrix to represent

the labelled graphs and to detect their isomorphism. Starting from the frequent edges from the graph database, the algorithm searches the frequent subgraphs by adding frequent edges progressively. By normalizing the incidence matrix of the graph, the algorithm can reduce the computational cost of verifying the isomorphism of the subgraphs.

Yan *(et al.)* published a new algorithm, called *LEAP (Descending Leap Mine)*[22], to exploit the correlation between structural similarity and significance similarity in a way that the most significant pattern could be identified quickly by searching dissimilar graph patterns. Two novel concepts, structural leap search and frequency descending mining, are proposed to support leap search in graph pattern space. They later claim that the widely adopted branch-and-bound search in data mining literature is indeed not the best, thus sketching a new picture on scalable graph pattern discovery. *LEAP* develops a horizontal pruning mechanism by employing a structural proximity measure between the siblings in the search tree. as opposed to the traditional branch-and-bound algorithms which only prunes vertically in the search tree. LEAP is developed to extract significant graphs not necessarily the frequent ones, but it uses frequency association to deduce the significance of a subgraph.

1.2.3 Partition Based Methods

Wang *(et al.)* proposed a mining framework based on partitioning the graph data, *PartMiner*[20] in 2006. The PartMiner algorithm finds the frequent subgraphs by dividing the database into smaller and more manageable units, mining frequent subgraphs on these smaller units and finally combining the results of these units to losslessly recover the complete set of subgraphs in the database. The algorithm works by dividing the graphs into smaller, more manageable subgraphs until these subgraphs can fit into the main memory. With this approach, existing memory-based graph mining algorithms can be utilized to discover frequent patterns in these subgraphs. The discovered patterns are joined via a merge-join operation to recover the final set of frequent patterns that exist in the original graphs. Authors then extend PartMiner to handle updates in the dynamic environment. In our study, we use the partitioning and merging methods introduced in *PartMiner* to partition the graph data and run the merge-join jobs on computed patterns. *PartMiner* is based on a graph partitioning but it is not a distributed algorithm. In our thesis, we use the partitioning and merging methods defined in PartMiner but we devise a distributed algorithm.

In 2008, Nguyen *(et al.)* published a study which uses another horizontal partitioning scheme to mine graph data. Their method, *PartGraphMining*[16] divides the graph data set into fragments. Each fragment consists of one or more graphs from the set. When partitioning the graph data set, the method adds graphs with minimum similarity to the same fragment and tries to balance the size of each fragment. To calculate the minimum similarity, the algorithm takes a graph and a centroid of a fragment and computes the intersection. Using either *Gaston* or *gSpan* memory based methods, frequent local patterns are extracted in these fragments. *PartGraphMining* then merges the sets, checks every local frequent subgraph for the global frequency, and builds the resulting set.

1.2.4 Distributed Approaches

Very recently in 2009 Kang, Tsourakakis and Faloutsos devised a generic representation of a graph and some very common operations on it. Their framework, *Pegasus*[7] exploits the idea of unification of seemingly different graph mining tasks via a generalization of matrix-vector multiplication. The method, GIM-V, is implemented on open source *MapReduce*[2] library, *Hadoop*[3], thus the computation of matrix operations could be distributed among a cluster of machines, providing theoretically unbounded scalability. The framework includes the GIM-V implementation of some of the well known graph algorithms such as PageRank calculation, finding connected components, random walk with restart and graph diameter estimation. The study is promising in that it provides massive scalability via the use of MapReduce, but it is not practical to use since it requires a complete rewrite of the memory based algorithms, where the operations provided by GIM-V may fail to be the complete set of methods needed to implement those algorithms. In our thesis we employ a well known simple graph representation and operations on that representation which would allow using existing techniques. Even when the existing methods could be defined using GIM-V, we avoid having to rewrite those algorithms in our method focusing on the effectiveness.

Liu *(et al.)* proposed another distributed algorithm based on MapReduce to find the motifs in prescription compatibility networks, again in 2009. Their method, *MRPF*[14], distributes the computation of frequent patterns exhaustively. For every pattern in parallel, the matching subgraphs are found, they are extended with neighbouring vertices and the resulting new pattern is added to the set. Then the frequency of the patterns are computed in a parallel

fashion as the matching subgraphs for that patterns are already known from the previous step. This method is extremely distributed and it may scale well but it uses a very basic pattern generation method; it does not choose to employ many of the pruning methods already discovered. In this study, however, we incorporate partitions as the unit of distribution rather than the vertices.

In August 2009, Malewicz, Austern, Bik, Dehnert, Horn, Leiser, Czajkowski from Google announced their distributed large scale graph computation framework, *Pregel*[15]. Although not much is revealed, Pregel is not founded on MapReduce parallelism, rather it is built on another method called *Bulk Synchronous Parallel*[19]. BSP uses the notion of supersteps where each superstep consist of a local computation, messaging between nodes and a barrier synchronization. It has been noted that Pregel uses vertices as the unit of distribution. BSP seems to suit graph problems better than MapReduce (MR) in that sending successive MR jobs as many graph problems may require could easily be simulated by many supersteps in one BSP computation. Although there are some libraries which provide BSP functionality, there has been no production ready, fault-tolerant and widely used frameworks around. Our method is not based on BSP but on MapReduce; as explained in Section 5.2 future direction may divert using BSP but for now there is not well known production ready computing environments for that method.

1.3 Overview Of The Proposed Method

In this thesis, we devise a scalable and distributed method to mine the frequent subgraphs. We use MapReduce programming paradigm and existing partitioning and memory based mining algorithms. Given a graph, the number of times it should be partitioned and the minimum support value, our method sends iterative MapReduce jobs each of which bi-partitions the data. The algorithm then starts computing the frequent patterns in all the partitions. A MapReduce job runs CloseGraph on each partition in parallel. All partitions form a tree structured directory. In merge-join step a simple tree scheduler sends successive MapReduce jobs each of which merge-joins the calculated subgraphs of two partitions.

A more detailed overview of the method is described in Chapter 3.

1.4 Contributions

Our contributions in this thesis are:

1. We propose a new scalable algorithm that can distribute the computations of frequent patterns to individual machines. For this method, we make use of an existing and production ready high level distributed computing environment, MapReduce, and its popular open source implementation.

   (a) We propose that our method could scale up to great numbers of edges and vertices as the graph data is kept in a distributed file system and the computation runs on the partitions of the big data most of the time.

   (b) We show that since the algorithm is implemented using MapReduce paradigm, the performance of the method could scale horizontally as the size of the machine cluster increases.

   (c) We also show that the implementation of this method is inherently reliable as it is based on Hadoop/HBase fault tolerant computation model.

2. Even though we use a specific pattern extraction method for our study, we develop a method that can be plugged in with different algorithms.

3. When compared to other methods that are built only upon local and sequential computations, we assert that our method performs better when the size of the graph data is increased.

4. In this study, we propose that our method, when compared to other distributed graph mining frameworks, generally tends to have less extra computation stemming from the fact that instead of working on a vertex based distribution, we partition the graph and try to find the patterns in these subgraphs. Instead of processing every vertex in parallel and generating subgraphs starting from that vertex, we accumulate the patterns found in partitions and build the whole set.

1.5 Thesis Outline

This thesis is organized as follows: In Chapter 2, we provide the necessary background knowledge to understand the problem domain and the solutions. A general view on distributed systems, MapReduce and its implementation along with the intuition behind closed frequent pattern mining is provided. In Chapter 3, datasets and clusters are described and technical details of the proposed methods are given. The algorithm is detailed and the completeness of the solution is addressed. In Chapter 4, the experiments and their results which demonstrate the utility of the proposed methods are shown. In Chapter 5, the thesis is concluded with a summary and future directions; the possible expansion route of this kind of distribution model is mentioned.

CHAPTER 2

BACKGROUND

2.1 Graph Preliminaries

In the following sections (2.1.1, 2.1.2, 2.1.3), basic data structures based on graphs which are used throughout the project are explained.

2.1.1 Graphs

A graph is a widely used data structure to represent an abstract set of objects where some pairs of the objects are connected by links. Typically the interconnected objects, vertices or nodes, hold the data object while the links that connect some pairs of vertices, edges, denote the relations between the nodes. The edges may have orientation, they can be directed (one way) or undirected. Edges can be given weights to designate the degree of the relation between the nodes. Additionally nodes and/or edges of a graph can be labelled as shown in Figure 2.1[1].

[1] http://upload.wikimedia.org/wikiversity/en/thumb/1/10/Simple_directed_graph.svg

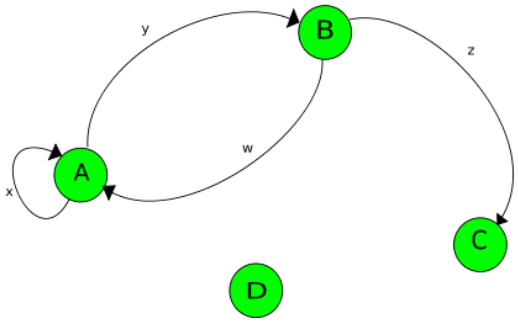

Figure 2.1: An example of a very simple directed labelled graph.

Graphs can be represented as a matrix where rows and columns are the nodes and each cell of the matrix C_{ij} holds the edge information (e.g weight) between nodes i and j. Although this representation is very straightforward, it is not space efficient for sparse graphs (graphs that have relatively less number of edges). A more compact way to store graphs is to use adjacency lists. An adjacency list is a map from vertices to the list of nodes they are connected to. This way no extra space is needed for a non-existent edge.

2.1.2 Frequent Patterns

A frequent subgraph in a graph is a subgraph which repeats at least as many as the given threshold. Support of a subgraph is the number of occurrences of that subgraph in graph data. Some of the definitions of frequency takes a designated number as the minimum support while others uses percentage as the minimum criteria.

A frequent pattern, on the other hand, is a pattern (mostly consisting of vertex and edge labels) whose matching subgraphs together are frequent in graph data. Support of a pattern is the sum of supports of matching subgraphs of that pattern. In this thesis we try to find frequent patterns which has a numerical minimum support threshold.

2.1.3 Closed Patterns

A closed subgraph in a graph is a subgraph which has no other supergraphs with the same support it has. Closed pattern is a pattern which has no other supergraphs (patterns) with the same support.

Closed frequent patterns are important since they are maximum frequent patterns.

2.2 Biology Preliminaries

In the following two sections, the biological structures that are given input to the methods derived in this thesis work are overviewed.

2.2.1 Proteins

Proteins are organic compounds and essential parts of organisms and participate in virtually every process within cells. They are formed of linear chains of amino acids and folded into a globular form. Chains of amino acids are held together via the peptide bonds. The sequence of amino acids in a protein is defined by the sequence of a gene, which is encoded in the genetic code.

There are 20 known amino acids and a protein is represented basically by a string encoding of this 20 letter alphabet. The amino acid sequence is not a complete representation as a protein's function or its interaction with other proteins are also defined by their structures (Figure 2.2[2]), their physiochemical properties and locations in the living cell, among other things.

[2] http://upload.wikimedia.org/wikipedia/commons/6/60/Myoglobin.png

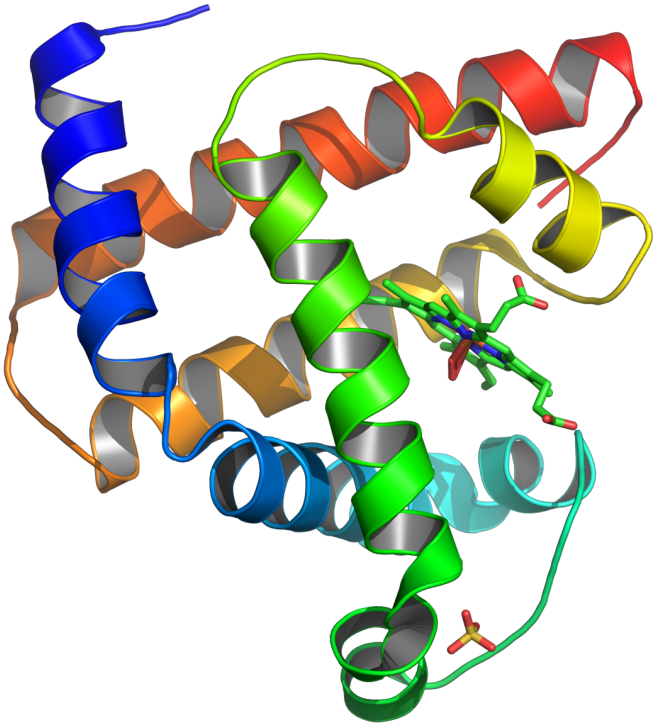

Figure 2.2: A 3D structure of myoglobin protein showing coloured alpha helices.

2.2.2 Protein-Protein Interaction Networks

Protein-protein interaction networks are graphs that represent the interactions of protein pairs in a genome. In a protein interaction network the protein pairs that are connected by an edge interacts with each other while for the pairs that have no edge, there is no interaction between the proteins. Since the extraction of this network for an organism is hard as there is an uncertainty in determining interactions between proteins, protein-protein interaction networks may be weighted and often undirected where the weight of the edge indicate the confidence value for the interaction between proteins. Edges may also be labelled to indicate the type of interaction. A typical protein-protein interaction network forms a very sparse graph as only a tiny portion of all possible edges is present in the network.

The interactions between proteins are important for very numerous biological functions such as signal transduction, i.e the fact that signals from the exterior of a cell are mediated to the inside of that cell by protein-protein interactions of the signalling molecules. This process, for instance, plays a fundamental role in many biological processes and in many diseases (e.g. cancers). Another biological function would be interaction of a protein with another

protein for long time to form a protein complex. In addition to this, a protein may interact with another for a very short time and causing a modification in that protein.

For our study, we input our experiments with labelled protein-protein interaction networks and try to find frequent patterns in these networks. The labels in protein interaction networks are taken from the gene ontology annotations of yeast (see Section 3.1.2). Below, Figure 2.3[3] shows a PPI for yeast organism.

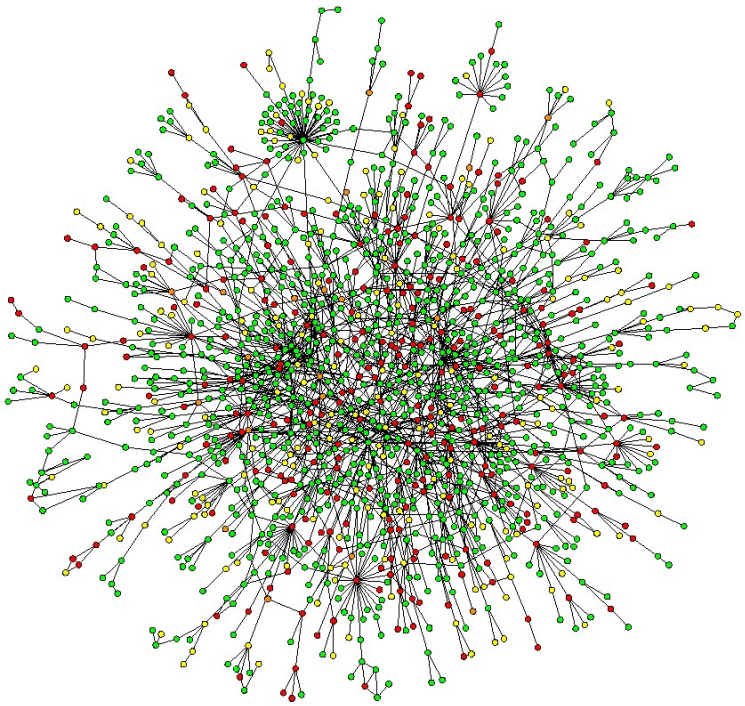

Figure 2.3: Protein-Protein Interaction Network of yeast organism.

2.3 Distributed Programming Preliminaries

The following sections overview the distributed programming methods and their implementations in this thesis work. Section 2.3.1 2.3.3 explains the distributed storage system while the distributed computation framework is explained in Section 2.3.2.

[3] http://www.bordalierinstitute.com/images/yeastProteinInteractionNetwork.jpg

2.3.1 Distributed File System: GFS and HDFS

The *Google File System (GFS)*[6] is a distributed file system developed by Google. Ghemawat *(et al.)* have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients. GFS is optimized for large amounts of data (in the scale of petabytes) and it provides a reliable efficient access to the data using massive clusters of computers.

GFS files are only extremely rarely overwritten, or shrunk and are usually appended to or read and file system holds a rather large block size for a file, typically 64 megabytes. Each block of a file is replicated among the cluster. The replication factor may be configured to large numbers but it is usually 3-5. When a disk fails or a computer gets unresponsive, the replication factor is first automatically decreased for all the blocks in that disk or machine and then system starts copying the lost blocks around to preserve the reliability. Unless there is simultaneous failure of computers at least as many as the replication factor, the system is guaranteed to serve all the blocks.

A typical GFS cluster consists of a *master node* and many slave nodes, i.e *chunk servers*. Master node does not store the actual data but is responsible for keeping the metadata associated with the chunks like tables mapping the 64-bit labels to chunk locations and the files they make up, the locations of the copies of the chunks, what processes are reading or writing to a particular chunk, or taking a ”snapshot” of the chunk pursuant to replicating it. Metadata is kept up to date by master server getting heart-beat updates from chunk servers. One of the main features of this file system is that no data transmission flows through master. When a read or write request is invoked to the master node, it returns the addresses of the chunks that the process should connect, then all data is transmitted directly between the client and the chunk server.

Chunk servers, on the other hand, store the data in 64 bit addressable chunks. Each chunk is replicated several times throughout the network, with the minimum being three, but even more for files that have high demand or need more redundancy.

Hadoop[3] is an open source project from Apache Software Foundation. Hadoop is a collection of projects which aim to provide reliable and scalable distributed computing. Hadoop

has three main subprojects, namely *Common*, *HDFS*, *MapReduce* and an echosystem provided along with these subsystems. In our thesis, all our implementation is done using the open source projects of Common, HDFS, MapReduce and HBase.

Hadoop subproject, *HDFS*[4], designed after GFS. The basic I/O operations are shown in Figure 2.4[5]. Although some minor differences exist, most notably the naming - chunks servers are called *DataNode*s and master server is called *NameNode*. HDFS is an open source implementation of GFS, and is used as the base level storage for this thesis work.

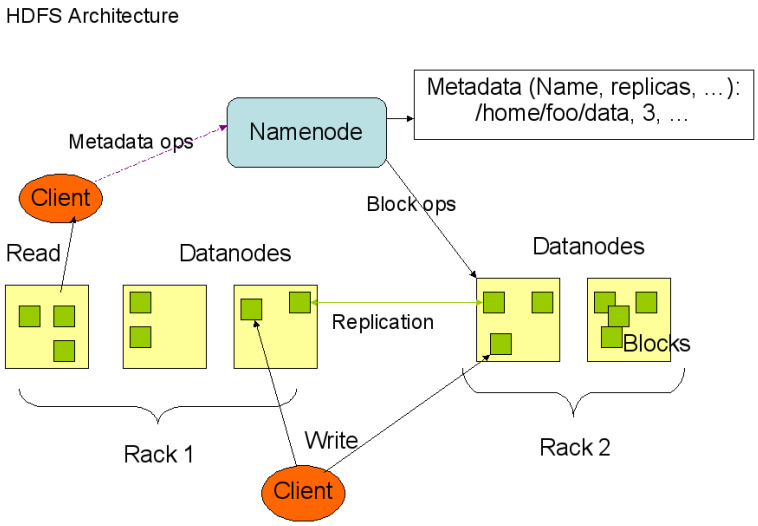

Figure 2.4: HDFS Architecture.

2.3.2 MapReduce

MapReduce is a distributed computation framework developed by Google. Basically it is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. The framework is inspired by map and reduce functions commonly used in functional programming.

[4] http://hadoop.apache.org/hdfs/
[5] http://hadoop.apache.org/common/docs/current/hdfs_design.html

A typical MapReduce cluster consists of a *master* node and many *worker* nodes. While both steps of the actual computation is carried out in worker nodes, the client only interfaces with the master node, thereby facilitating the client API.

The programs written in MapReduce are composed of two steps, *map* and *reduce*. In the map step, the master node tries to divide the data by calculating the input splits and assigns this information to worker nodes. Each worker node, in this step, computes the given function and emits and intermediate key/value pairs. In the reduce step, the values for the same key are presented to the same node via a partitioning and shuffling phase. A partition method is called for every key that a map emits to define the reducer it will be sent. Then every worker, given a key and a set of values, computes the function and emits the result.

A running MapReduce job process is shown in Figure 2.5[6]. User program is submitted to *master*, it starts the *worker* nodes with the split information. Worker nodes emit intermediate key/value pairs to local disks and the pairs are hashed to reducers according to the partition function. *Master* then assigns *worker*s as reducers. The reduce phase is finished when reducers emit the output.

Provided each mapping operation is independent of the other, all maps can be performed in parallel. By the same principle, a set of 'reducers' can perform the reduction phase; all that is required is that all outputs of the map operation which share the same key are presented to the same reducer, at the same time. For this property, MapReduce is very suitable for embarrassingly parallel applications on large data. A large server farm can use MapReduce to sort a petabyte of data in only a few hours.

The parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation. The failure policy in MapReduce is to schedule the failing task to another worker. Assuming that the input is still intact and there is no programmatic error, if one mapper or reducer fails, this policy mostly guarantees the result as the only other possibility is an hardware problem (a disk or a machine failure) during the execution.

[6] http://www.eyaloren.org/slides/2008/02/sw-hadoop/mapreduce.jpg

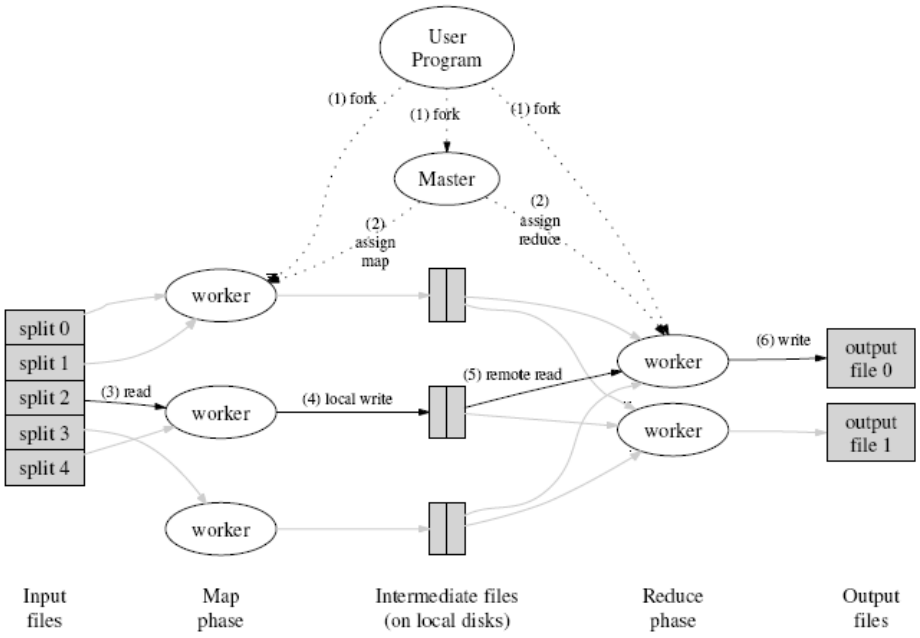

Figure 2.5: MapReduce architecture and execution sequence.

A sample program in MapReduce could be counting the words in a set of documents. Each map takes a document and for every word w in that document emits $(w, 1)$ pairs. The reducers then get a list of $(w, [1, 1, 1, 1...])$ pairs as input and emits the $(w, count)$ for every word w where count is the number of times that word appeared in all documents.

Hadoop[3] has another subproject, *MAPREDUCE*[7] which is an open source implementation of Google's MapReduce. Hadoop is written in Java and has some differences from Google's implementation, like names - workers are *tasktracker*s while master is *jobtracker* - among other things. Our thesis work uses MapReduce as the principal distribution mechanism.

2.3.3 Distributed Key-Value Store: BigTable and HBase

Since GFS is not geared towards random access to the data but it is more tailored for sequential access and streaming, a highly immutable type of data structure like matrix is best represented using a higher level API, a key/value store. In our thesis work we use the distributed key/value store, HBase[5], for keeping our data and running MapReduce jobs on it. HBase, another

[7] http://hadoop.apache.org/mapreduce/

Hadoop subproject, is an open source implementation of Google's *BigTable*[1].

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. However, it departs from the typical convention of a fixed number of columns, instead described by the authors as *a sparse, distributed multi-dimensional sorted map*, sharing characteristics of both row-oriented and column-oriented databases.

Tables are optimized for GFS by being split into multiple tablets which are segments of the table as split along a row chosen such that the tablet will be $\tilde{2}00$ megabytes in size. When sizes threaten to grow beyond a specified limit, the tablets are compressed and with algorithms *BMDiff*[8] and *Zippy*[9] which is described as less space-optimal variation of *LZO*[10] but more efficient in terms of computing time. The locations in the GFS of tablets are recorded as database entries in multiple special tablets, which are called *META1* tablets. META1 tablets are, again, found by querying the single *META0* tablet. Like GFS's master server, the META0 server is not generally a bottleneck since the processor time and bandwidth necessary to discover and transmit META1 locations is minimal and clients aggressively cache locations to minimize queries.

Although the client sees a table as a collection of rows where each row has predefined *column families*, the actual storage is quite different than that of a traditional DBMS. A column family is divided into columns which can be added or deleted dynamically. Each cell is addressed with a key composed of row, column family, column and timestamp.

These key/value stores are inherently reliable as HBase is built upon HDFS while BigTable is based on GFS. The following conceptual views of tables (Tables 2.1, 2.2, 2.3 and 2.4) are compiled from the canonical examples in HBase architecture wiki[11].

[8] http://norfolk.cs.washington.edu/htbin-post/unrestricted/colloq/details.cgi?id=437
[9] http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=755678
[10] http://www.oberhumer.com/opensource/lzo/
[11] http://wiki.apache.org/hadoop/Hbase/HbaseArchitecture

Table 2.1: Logical View Of A HBase/BigTable Table

Row Key	Timestamp	Column *contents:*	Column *anchor:*		Column *mime:*
com.cnn.www	t9		anchor:cnnsi.com	CNN	text/html
	t8		anchor:my.look.ca	CNN.com	
	t6	*html content*			
	t5	*html content*			
	t3	*html content*			

At physical level, each column family is held as a separate file indexed with row keys. Since non-existing columns does not claim any space, a sparse data structure can be designed to fit very efficiently using this data model.

Table 2.2: Physical View Of A HBase/BigTable Table (1)

Row Key	Timestamp	Column *contents:*
com.cnn.www	t6	*html content*
	t5	*html content*
	t3	*html content*

Table 2.3: Physical View Of A HBase/BigTable Table (2)

Row Key	Timestamp	Column *anchor:*	
com.cnn.www	t9	anchor:cnnsi.com	CNN
	t8	anchor:my.look.ca	CNN.com

Table 2.4: Physical View Of A HBase/BigTable Table (3)

Row Key	Timestamp	Column *mime:*
com.cnn.www	t6	text/html

Because of very aggressive caching of tables, HBase, for instance, most of the time serves all the data directly from the memory which allows random accesses and updates to existing data very efficiently.

CHAPTER 3

MATERIALS AND METHODS

3.1 Data Sets

In this section, we represent the data sets used in this thesis and the tools we use to measure the performance of our method. The implementation of our study is evaluated on three protein-protein interaction networks of *S. cerevisiae (baker's yeast)*. These three labelled graphs are tested on three different cluster configurations. Our method is tested along with PartMiner[20] and CloseGraph[23] methods on applicable configurations on these clusters.

3.1.1 Protein-Protein Interaction Networks

A protein-protein interaction network may be represented as a labelled graph (directed or undirected) by using known information about the proteins as the labels of the nodes. Any graph mining method when applied to these networks reveals interesting correlations about the living organism. For instance, motifs in protein-protein interaction networks could help understanding how an essential function in a living cell works and which proteins takes part in carrying out that function.

In our thesis, to be able to test our method we use three different protein-proteion interaction networks of yeast organism. Because it is readily available and easy to culture, baker's yeast has long been used in chemical, biological, and genetic research. In 1996, after 6 years of work, *S. cerevisiae* became the first eucaryote to have its entire genome sequenced. Our smallest PPI network for this organism contains experimentally verified interactions down-

loaded from the Database Of Interacting Proteins (DIP)[1] and consists of little more than 3K vertices and around 20K edges. The other PPI networks, PIN-STRING and WI-PHI[8] are greater in both number of vertices and edges. PIN-STRING has 4K vertices and 50K edges. WI-PHI has 5K vertices and 55K edges. These two networks contain computationally predicted interactions. We used highly confident subset of the predicted interactions. Using these networks, we test the performance of our method in comparison to the other methods. As noted earlier, PPI networks often form very sparse matrices. Considering the DIP network of yeast, out of possible 4.5M (undirected) interactions there are only 20K in the network, around 0.004% of the whole interaction space.

3.1.2 Gene Ontologies

The Gene Ontology[2] (GO) is a bioinformatics initiative which aims to maintain and further develop its controlled vocabulary of gene and gene product attributes, to annotate genes and gene products and to assimilate and disseminate annotation data.

GO annotations have many uses, for example, they can be utilized to predict protein-protein interactions. However, since GO annotations are not likely to be available for newly-sequenced organisms, we do not use GO annotations for PPI prediction in our thesis. Still, our test organism (baker's yeast) is well-studied and well-annotated. But even for well-studied organisms such as baker's yeast, a significant portion of the PPI network is not yet validated by wet-lab experiments. Because of this, for a given GO cluster, it is possible that there are other proteins that are in the same functional path but not yet GO-annotated. For this reason some of the proteins are not annotated in the network, to be able to deduce to a generic algorithm in our thesis work, we simply ignore them as they cannot be assigned to be a part of a pattern.

We downloaded a recent yeast GO annotations file. The nodes in the PPI networks are annotated using this file.

[1] http://dip.doe-mbi.ucla.edu/dip/Main.cgi
[2] http://www.geneontology.org/

3.2 Computing Environment

The algorithms *CloseGraph*, *PartMiner* and our implementation are tested on three machine clusters with different configurations. Since *CloseGraph* and *PartMiner* methods are not distributed, they are tested on a modest commodity computer. As we implemented our algorithm in MapReduce, it is possible to run the job as a local computation as opposed to a distributed one without changing a line in the source code. In order to get a controlled comparison, we test our method as a local MR job.

Our implementation is also tested with two distributed cluster configurations, one with a 3-machine and one with a 6-machine cluster. Both clusters are actually a one 6 machine Hadoop HDFS/MapReduce cluster but we force the partitions, the number of mappers and reducers so that each time only the designated number of machines are used.

3.3 Overview

This section details the overview of the algorithm that we use in this thesis. Individual steps in the algorithm is described in the following sections in detail. Our method works as follows;

1. Given a graph, the number of times it should be partitioned and the minimum support value the algorithm sends iterative MapReduce jobs each of which bi-partitions the data.
2. The algorithm then starts computing the frequent patterns in all the partitions. A MapReduce job runs CloseGraph on each partition in parallel.
3. All partitions form a tree structured directory. In merge-join step a simple tree scheduler sends successive MapReduce jobs each of which merge-joins the calculated subgraphs of two partitions.

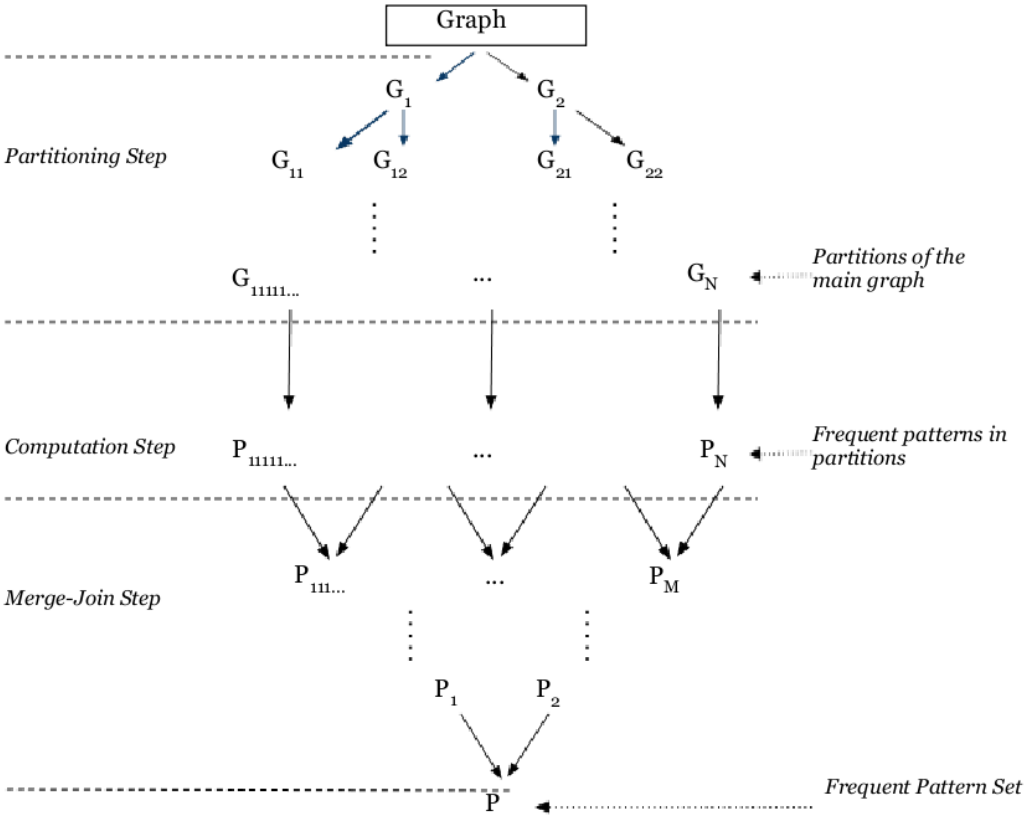

Figure 3.1: Overview of our proposed algorithm (MR Graph Mine).

3.4 Representation Of Graphs

Graphs are hold as adjacency matrices on HBase. Each graph is stored as a table whose columns are vertex ids. A graph table has two column families, *neighbors* and *label*. Any neighboring vertex is added as a column in *neighbors* column family. Vertex label and neigbors are held as separate column families avoiding the costly skip of the unneccessary information when a program tries to get only one of the information, an adjacent vertex or a label. Adjacent vertices of a node can be dynamically added/removed, and there may be different number of neigbors for each vertex. A conceptual view of the HTable for a graph is illustrated in Table 3.1.

Table 3.1: A Sample Graph Table.

Vertex ID	Family *neighbors:*				Family *label:*
YJR091C	YJL034W=1	YGR166W=1			3729
YJL034W	YJL034W=1		YPL187W=1		16887
YDL108W				YER171W=1	8353

As can be seen from the table, the node in question has all the information in one row. The *neighbors* column family stores the edge information, the column identifier denotes the id of the neighbor whereas the value of the cell is the weight of the edge.

3.5 Dividing The Graph Into Partitions: *MRGraphPart*

The motivation behind the partition based approach is to effectively reduce the computation space by dealing with smaller graphs. Using this approach, we can decrease the graph complexity and size but they later need to be combined to compile the overall result. In order to make the merge-join operations fast, division should partition the graph into two near distinct graphs. Partitioning according to least connectivity yields to have less number of units involved in the merge-join operation.

This method is based on the algorithm defined in the *PartMiner* paper[20]. The method described in PartMiner uses two criteria while deciding on the vertices set that should be separated;

1. Minimizing the connectivity: Graph should be partitioned into two least connected vertex sets.
2. Isolationg the updated items: The vertices and edges that are frequently updated should be isolated from the rest of the subgraph.

PartMiner uses *isolated updates* criteria as it is designed to deal with frequently updated graph data sets. In our case, the input data that we use rarely changes. In fact, in every calculation we assume that the graph input is static and the updates are done prior to running of the algorithm.

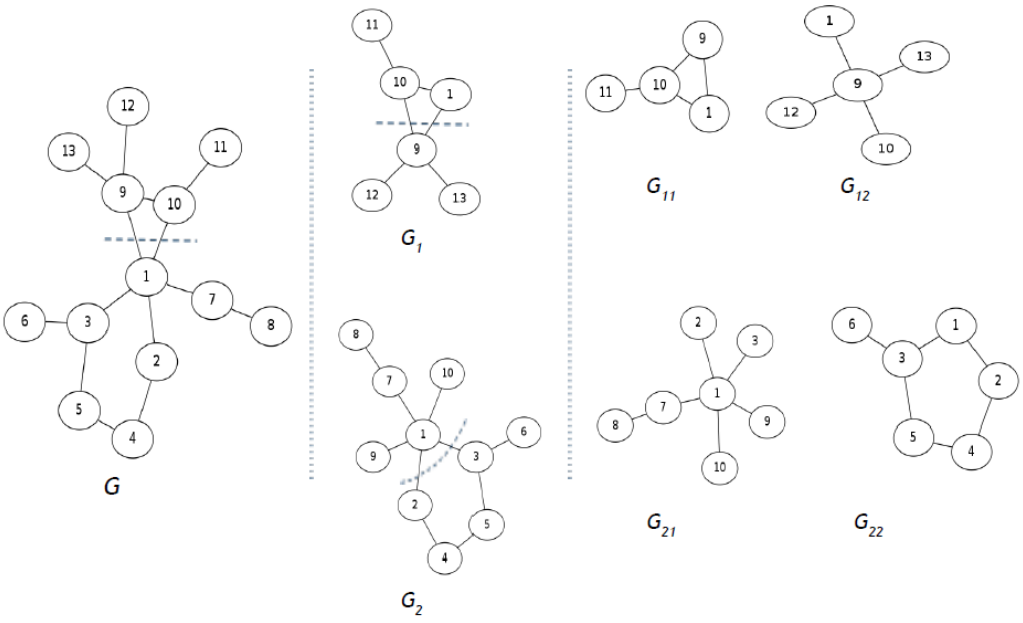

Figure 3.2: GraphPart partitioning method illustrated.

Partitioning process is illustrated in Figure 3.2. Note that, division process copies the connected edges to both subgraphs. In the first bi-partition, for instance, G_1 and G_2 both reserves the edges *(1, 9)* and *(1, 10)*.

GraphPart uses a weight function to identify when to separate a vertex set into a new graph whilst dividing. If we have two sets of vertices V_1 and V_2 during a partitioning process, the value of the weight function for that partitioning scheme is defined by $\omega(V_1) = \lambda_1 \frac{\sum_{v_i \in V_1} v_i ufreq}{|V_1|} - \lambda_2 |E_{V_1,V_2}|$. This weight function, defined in PartMiner[20], basically decreases with the increasing number of edges between the two sets and has a higher value when the vertices have higher frequency of update. The algorithm of GraphPart, starts from each vertex and builds a set by depth-first traverse of the graph. This process is stopped when 1) either there is no connected edge that can be traversed starting from that vertex, 2) or half of the vertices are visited in this graph, which is needed to keep the partitions similar in size. For each vertex, GraphPart builds two sets, one with the vertices that are visited and the other with the rest of the graph. The partitioning process ends when the algorithm finds two sets that maximizes the value of the weight function.

Our method uses a modified version of the weight function and changes the GraphPart so

that each iteration of depth-first search is done in parallel. Our weight function is simply the negation of the number of edges between two vertex sets;

$\omega(V_1) = -|E_{V_1,V_2}|$.

Algorithm 1 MRGraphPart

$G = \{V, E\}$; {Input graph, composed of vertices and edges}

map(V_i, G); //Map Phase

$VS_i \Leftarrow DFSScan(V_i, G)$;

$score \Leftarrow scoreOf(VS_i, G)$;

$GObject \Leftarrow \{score, VS_i\}$;

$emit(KEY, GObject)$;

combine$(KEY, [GObject_1, GObject_2, ..., GObject_N])$; //Combine Phase

$max \Leftarrow -\infty, maxG \Leftarrow \emptyset$

for all G such that $G \in [GObject_1, GObject_2, ..., GObject_N]$ **do**

if$(G.score > max)$

$maxG \Leftarrow G$;

$emit(KEY, maxG)$;

reduce$(KEY, [GObject_1, GObject_2, ..., GObject_N])$; //Reduce Phase

$max = -\infty, maxG = \emptyset$

for all G such that $G \in [GObject_1, GObject_2, ..., GObject_N]$ **do**

if$(G.score > max)$

$maxG \Leftarrow G$;

$G_1 \Leftarrow maxG.graph$ //Construct the subgraph in maxG

$G_2 \Leftarrow G - maxG.graph$ //Construct the subgraph of the rest

$write(G_1), write(G_2)$ //Write two partitions to disk

The division method in this thesis, *MRGraphPart*, the distributed counterpart of *GraphPart*, is a MapReduce job to partition the graph. Above, in the Algorithm 1, the idea is detailed. Each mapper takes a vertex and starts the depth-first traverse of the graph from that vertex, and calculates the score of that partitioning. Mappers emit $(CONSTANT\_KEY, GraphObject)$ as the intermediate key/value pair where GraphObject is a simple object that has a score and a list of vertex ids. We keep the key the same for all the pairs so that every key/value pair

is hashed to the same combiner and reducer. After the map phase, combiners, which are basically mini reducers that work only on one machine with one map output[3], take all the GraphObject which is emitted by one mapper and outputs one GraphObject with the highest score. Basically combiners take *(CONSTANT_KEY, [GraphObject$_1$, GraphObject$_2$, ..., GraphObject$_N$])* and emit ($CONSTANT\_KEY, GraphObject_i$) where $GraphObject_i$ has the maximum score. Reducers work the same way combiner does. As there is only one key emitted by mappers and combiners, all the intermediate key/value pairs are collected in one reducer. This reducer simply selects the GraphObject with the highest score. The output of the job, and the reducer is simply two graphs, one graph with the vertices that are in the selected *GraphObject*, the other with the rest of the vertices present in the main graph. Our method takes a graph with a number indicating bi-partitionings and it iteratively calls MRGraphPart, first on the main graph then on the output graphs of the previous job.

3.6 Calculation Of Subgraphs

In this step, we simply plug in an already existing memory based frequent subgraph mining technique. As noted earlier, we chose CloseGraph since it is founded on a widely known method, *gSpan*[24] and it is used to find more relevant patterns as it searches for closed patterns. In the following section 3.6.1, CloseGraph algorithm is overviewed.

Our implementation of distributed calculation, *MRFreq*, uses CloseGraph and runs on every partition in parallel.

Algorithm 2 MRFreq

$S = [G_{P_1}, G_{P_2}, ..., G_{P_N}]$; // Input the partition list

map($\_, G_{P_i}$); // Map Phase

$P_i \Leftarrow CloseGraph(G_{P_i})$; // Compute frequent patterns

$write(P_i)$ // Persist pattern list

$emit(KEY, \emptyset)$; // Emit nothing

Despite being a MapReduce job, *MRFreq* does not define a combiner nor a reducer as it is shown in Algorithm 2. After the calculation step, the mappers write out the resulting patterns for that partition and do not emit anything. This kind of implementation, a map-only-

[3] http://wiki.apache.org/hadoop/HadoopMapReduce

MapReduce-job, avoids the partitioning and shuffling steps as well as the reducer initialization. In actual implementation, though, mappers emit a place holder but program is run with a zero reducer configuration, thus the same effect is experienced. The output of each computation is a file which contains a list of computer patterns (subgraphs) and their frequencies.

3.6.1 CloseGraph Computation

This section provides the techniques used in CloseGraph to efficiently mine the graph data. At the heart of the algorithm, *DFS subscripting* method is defined to represent and extend a graph. A graph is mapped to its unique DFS code and a lexicographic ordering among all the codes for subgraphs is introduced.

CloseGraph scans the graph data or data set for a subgraph g. For every edge e that g can be extended to $g\Delta_x e$, CloseGraph counts the equivalents of $g\Delta_x e$. If the subgraph is frequent, it is added to the resulting frequent subgraph set.

The implementation of CloseGraph starts from the frequent 1-edge graphs and extends them. If a graph is frequent, it is added to the list of patterns that will be output.

3.6.1.1 DFS Notation

CloseGraph uses the same notation *gSpan*[24] does to map a graph to a unique notation. By visiting the vertices of a graph in a depth-first traversal, this method constructs a depth-first search tree. In a such DFS tree for a graph, the first node (the node which is visited first during the traversal) is called the *root* and the last node is called the *rightmost* node of the DFS tree. Since a graph can be traversed many ways starting from a different vertex, in order to be able map a graph to a unique and consistent DFS code, an ordering between the different DFS codes of a graph is introduced.

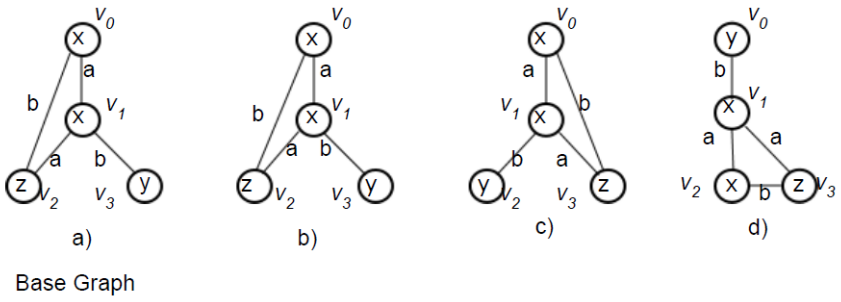

Figure 3.3: A sample graph and its possible DFS trees.

A *DFS code* is generated for each DFS tree traversals of a graph. DFS code is basically an edge order. Each edge is defined as a *5-tuple* $(V_{from}, V_{to}, L_{from}, L_{edge}, L_{to})$ where V's are the vertices of the edge and L's are labels of vertices and the edge itself. The order edges are listed is ruled by the following:

1. Edges are listed sorted according to the time they are visited.
2. For a vertex v all the backward edges should appear before its forward edges.
3. If a vertex v does not have any forward edges, the backward edge is placed right after the edge where v is the second vertex.

By the above definition, the graph in Figure 3.3b) has a DFS code of $((0, 1, x, a, x)\ (1, 2, x, a, z)\ (2, 0, z, b, x)\ (1, 3, x, b, y))$ and the graph in Figure 3.3c) has the DFS code of $((0, 1, x, a, x)\ (1, 2, x, b, y)\ (1, 3, x, a, z)\ (3, 0, z, b, x))$.

3.6.1.2 Lexicographic Ordering

A graph may have many different DFS tree representations and different DFS codes for each tree, in order to choose one as the unique representation for the graph, a lexicographical order is introduced. Among all the DFS codes that a graph can generate, the one with the minimum lexicographic value is chosen. This DFS code, when expressed as a string, is the first value alphabetically and is called the *Minimum DFS Code* of the graph. gSpan and CloseGraph use this string as the notation of a graph.

3.6.1.3 Rightmost Extension

In both gSpan and CloseGraph, the extension of a graph is done in two ways. When a graph g and its DFS tree T are provided; a new edge is added 1) either from rightmost node to one of the nodes in rightmost path *(backward extension)* 2) or from a node in rightmost path to a new vertex *(forward extension)*. This ensures that the extension yields a valid DFS code. Any other extensions of the graph would result in a different DFS code which cannot be a *child* node of the extended graph according to Definition 3.6.1. The *gSpan*[24] paper shows that this extension method, *rightmost extension*, covers all the frequent subgraphs and guarantees the correct mining result.

Definition 3.6.1 (Parent-Child Relation Of DFS Codes) *All the DFS codes form a DFS code tree in a graph such that a node at this tree with a DFS code* $P = (a_0, a_1, a_2, a_3, ..., a_m)$ *is called the parent of the children nodes with codes* $C = (a_0, a_1, a_2, a_3, ..., a_m, b)$*. This also states that* $\forall c_i \in C$ *s.t* C *is the set of children nodes of* P*,* c_i *is a rightmost extension of* P*.*

gSpan and CloseGraph both use the DFS codes of the initial subgraphs and build this tree by *rightmost* extending the patterns. CloseGraph differs from gSpan in that it early terminates on the generation of new DFS codes.

3.6.1.4 Early Termination

Basic gSpan algorithm, as noted earlier, selects the initial 1-edge subgraphs, extends them on the DFS tree using rightmost extension and calculates the occurrences. CloseGraph adds early termination to this algorithm as it searches only for the closed patterns. Recall that a closed pattern is a pattern which has no other supergraphs with the same support. So in CloseGraph algorithm, when a subgraph g is being extended in any way (in rightmost extension rule), if one of the extensions has the same number of occurrence with the original subgraph; the extension process is terminated. Since one of the supergraphs of g has the same support as g does, g cannot be a closed pattern. So the process continues by growing the supergraph and the unnecessary generation of other extensions of g is avoided.

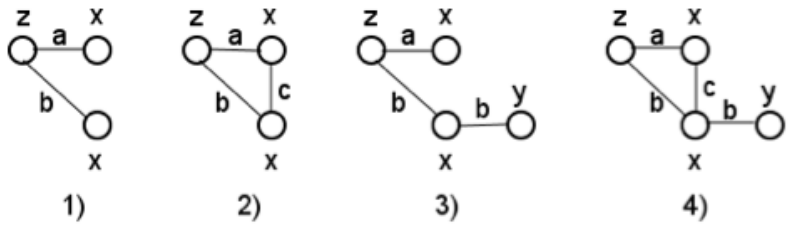

Figure 3.4: Early termination of rightmost extension.

As illustrated in Figure 3.4, when extending first graph (1) g, if the extension in (2) g' has the same support as g, then another possible extension (3) g'' is not generated. Since g is not a closed pattern it is omitted and the extension process continues from extending (2) g' into (4).

3.7 Merge-Join Operation

After the calculation step *MRGraphMine* algorithm starts the merge-join operation as defined in the PartMiner paper. For two calculated pattern sets U_1, U_2 and their $k - edge$ frequent patterns as $P^k(U_i)1$ and $P^k(U_2)$, the algorithm is illustrated in Algorithm 3.

Algorithm 3 MergeJoin

1: U_1, U_2; // Input the calculated patterns
2: $P^1(U) \Leftarrow P^1(U_1) \cup P^1(U_2)$ // 1-edge frequent patterns
3: $P^2(U) \Leftarrow P^2(U_1) \cup P^2(U_2)$ // 2-edge frequent patterns
4: $C^3 \Leftarrow join(P^2(U_1), P^2(U_2))$ // 2-edge frequent patterns which share an edge
5: $F^3 \Leftarrow checkFrequency(C^3, sup)$
6: **for** $k = 3; F^k \neq \emptyset; k++$ **do**
7: $P^k(U) = P^k(U_1) \cup P^k(U_1) \cup F^k$
8: $C_1^{k+1} = join(P^k(U_1), F^k)$
9: $C_2^{k+1} = join(P^k(U_2), F^k)$
10: $C_3^{k+1} = join(F^k, F^k)$
11: $C^{k+1} = C_1^{k+1} \cup C_2^{k+1} \cup C_3^{k+1}$
12: $F^{k+1} \Leftarrow checkFrequency(C^{k+1}, sup)$
13: **end for**
14: $P(U) \Leftarrow \cup P^k(U)$ // Resulting frequent pattern set

The merge-join operation of two sets of frequent patterns is a MapReduce job where map function executes the algorithm. The algorithm starts with merging 1-edge frequent patterns from both sets. Since these patterns cannot share an edge they are simply copied to result set as shown in Algorithm 3 line (2). Then 2-edge frequent patterns from the both sets are added to the result set. Since 2-edge graphs could share an edge that is broken whilst partitioning, a join operation is carried out and resulting 3-edge graphs are added to a set C^3 and the frequent patterns in C^3 is added to F^3 (lines 3-5). From this point on, algorithm merges $k - edge$ patterns by merging $k - edge$ patterns from both sets and invokes 3 join operations to build the edge-sharing $(k + 1) - edge$ patterns: both $k - edge$ patterns from the two sets are joined with the frequent edge sharing pattern set of previous step F^k. F^k is then joined with itself.

Completeness of the merge-join operation is detailed in PartMiner paper. It should be noted that the only tricky part to the merging operation is that after a join operation the set of edge-sharing graphs are checked for their frequency. Since we work on the partitions, we cannot use the global $MINIMUM\_SUPPORT (MIN\_SUP)$ as it will cause false mining results. Instead, we divide the MIN_SUP each time we partition the data. This way a partition is assigned a support predicate to check for.

3.7.1 A Simple Tree Scheduler

The parallelism in the computation of merge-join operation does not come from the MapReduce implementation as the merging two graphs is a map-only-MapReduce job which works only on one mapper. However, since we can merge two different nodes' children independent from each other, we distribute the computation of merge-join by a very simple tree scheduler.

Since the partitioning method iteratively divides the data into two parts, the patterns should be merged the same but reverse way as the partitioning of the graph. Our algorithm works on the partitioning tree and for every node, if the children nodes have their pattern set constructed, it submits a *MRMergeJoin* MapReduce job. MRMergeJoin merges the two pattern sets and writes the result set to the filesystem. If a child node has no pattern set constructed, the algorithm recurses to that child node and starts checking its children nodes. This ensures that children of two different nodes could be merge-joined provided that they are not hierarchically dependent on each other in the tree. This way, as we walk through the partitioning tree, we recursively merge-join the pattern sets in parallel starting from the leaves (the sets

computed via CloseGraph) to the root (the whole graph). In Figure 3.1, merge-join operation is illustrated.

CHAPTER 4

RESULTS

In this chapter, we present the experimental results of our method. As noted earlier, we run CloseGraph, PartMiner and our method (MRGraphMine) on 3 different graphs. All the methods are run on these input graphs on a single machine. Then we submit our method with distributed runs on two different cluster configurations.

The implementation of CloseGraph algorithm is loosely based on one of the existing implementations of gSpan provided by ParSeMis project[1]. We change the gSpan and add basic early termination conditions to get a working CloseGraph implementation. PartMiner algorithm, on the other hand, is coded from scratch basically. The implementation of memory based frequent pattern mining algorithm used in PartMiner default implementation, *Gaston* is again taken from the ParSeMis project. There are no optimizations or hooks introduced in both of these implementations and no performance analysis is reflected in their implementation. There may be more efficient implementations present for both algorithms but our use for them is very limited and they are mostly generated for the testing purposes.

MRGraphMine method is coded using HBase v0.20.2 and Hadoop v0.20 libraries. Like other algorithms, this method is also coded in Java 1.6. It uses the *concurrent* API defined in JDK in merge-join phase. The graphs are loaded to HBase tables using utility methods. In the experiments, only the computation related measurements are taken into account.

[1] http://www2.informatik.uni-erlangen.de/EN/research/ParSeMiS/index.html

4.1 Local Experiments

In the first experiment, all three algorithms are tested on three graphs to determine the scalability of the algorithms. The computations are done on a very modest single computer with 4GB memory and 3.0Ghz dual core CPU.

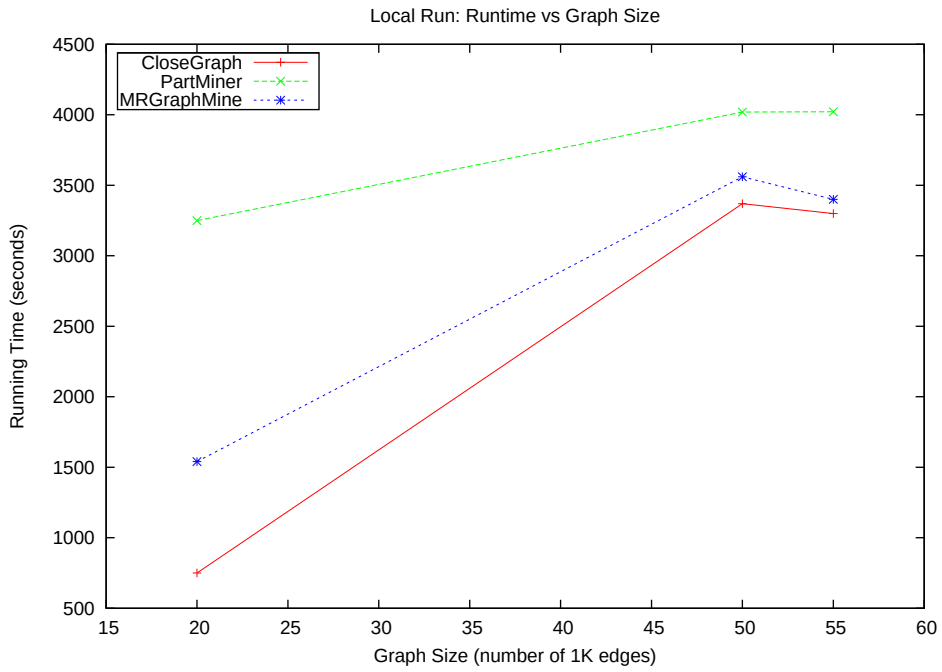

Figure 4.1: Runtime vs graph size.

In the experiment above (Figure 4.1), MRGraphMine and PartMiner run with 4 partitions. The minimum support value for all algorithms is set to 10%. In this setup, the memory based model CloseGraph is better performing on both of the runs. This is mainly because when compared to other direct computation models, MapReduce programs tend to have a constant overhead stemming from the fact that even though a MR program could be run locally without a change in the source code, running it locally produces unnecessary map and reduce steps. An algorithm which does the same steps of computation without map, combine, shuffle, partition or reduce phases could run slightly faster in practice, though, their algorithmic complexity would be the same. Apart from the inherent slight disadvantage, our algorithm partitions the data and merges the results. CloseGraph, in this case, avoids both the constant overhead by the MR paradigm and graph partitioning/merging, therefore it performs

better than the other two algorithms. PartMiner, on the other hand, performs the worst as it searches for all frequent patterns as opposed to closed patterns.

One thing to note about the experiment in Figure 4.1 is that when the graph size is nearly doubled, CloseGraph runtime increases more than our method (MRGraphMine). Increasing graph size creates a larger search space but PartMiner and MRGraphMine gets affected less as by dividing the graph they handle a portion of the size increase at a time.

On the same machine setup, the effect of minimum support to the runtime is analysed. In the experiment below (Figure 4.2) the minimum support are gradually increased from 5% to 20%. Having a less minimum support, all the algorithms run longer as they find more patterns 'frequent'.

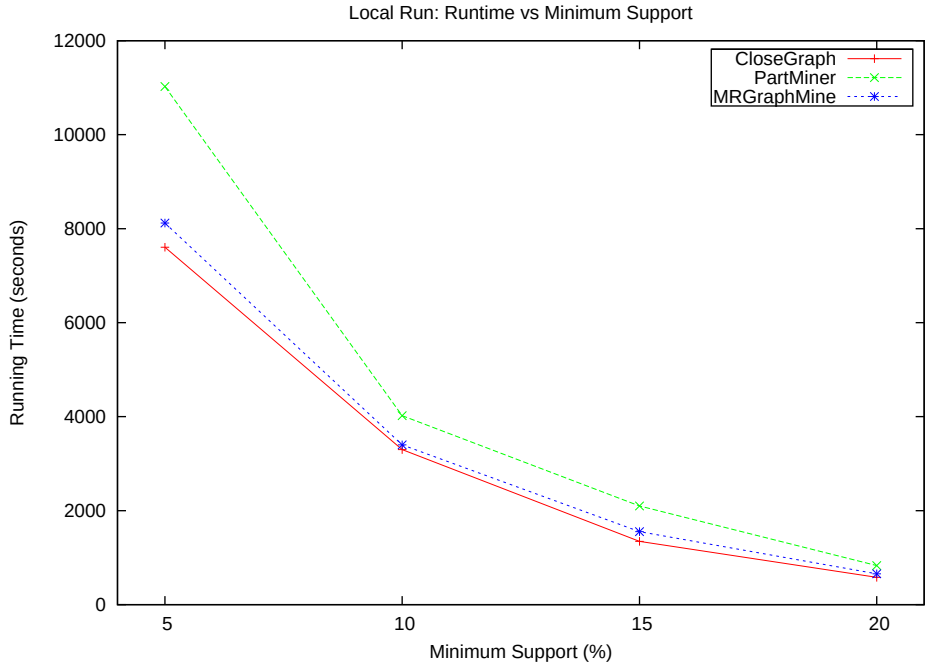

Figure 4.2: Runtime vs Min Support.

The results of the second experiment is somewhat similar to the first experiment. CloseGraph still works better performant than the other two. PartMiner, in this setup, gets a big negative impact on each decrease in the minimum support as the number of all frequent patterns gets very big whereas the number of closed patterns gets affected less. This second experiment is done using the bigger WI-PHI network as input (5K vertices, 55K edges).

4.2 Distributed Runs

The distributed environment is composed of six commodity computers, each of which has 4GB memory and 3.0 GHz dual core CPU. They are connected to each other via a gigabit LAN. Hadoop and HBase is installed to every computer in the cluster. There is one namenode for HDFS, one jobtracker for MapReduce, one HMaster for HBase master server. These three master servers are installed on different computers on the cluster. Apart from this, every computer serves as an HDFS datanode, a MapReduce tasktracker and an HBase region server.

On these clusters, we test the performance of our method. In the first experiment, we use the three graphs and use 3 of the servers in the cluster. The 3-server mini cluster is simply a cluster in which 3 of the tasktrackers are shut down. We did not shut down the HDFS datanodes or HBase region servers on these machines as they are not related to the computation but rather to I/O.

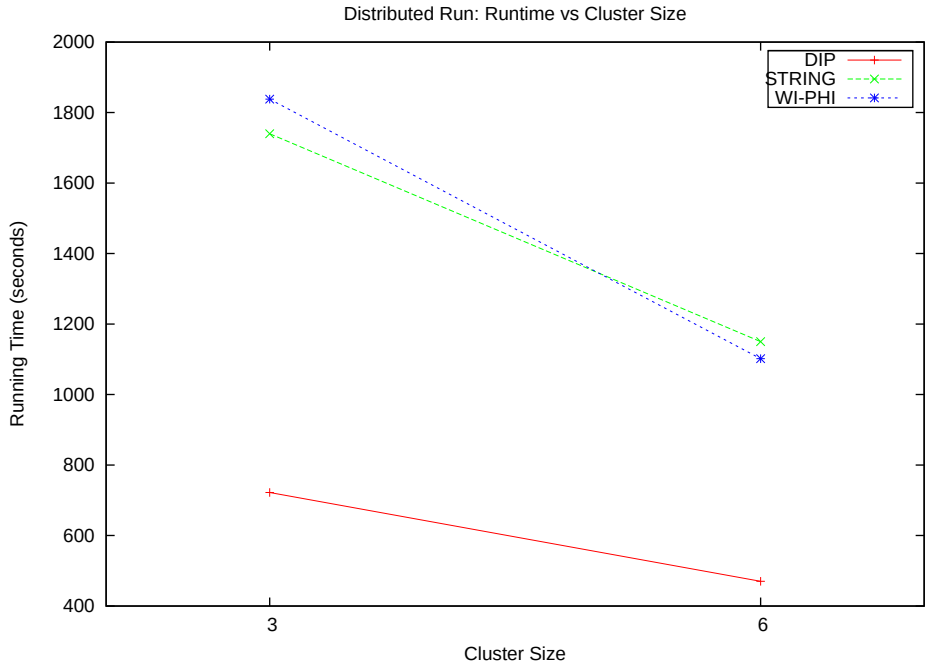

Figure 4.3: Runtime vs Cluster Size.

As shown in Figure 4.3, increasing cluster size for the same graph results in shorter running times. Although the relation between cluster size and running time is not exactly linear, the running time is reversely proportional to the size of the cluster. This proposes the horizontal

scalability in the mining process. In this setup, the number of partitions is set to 8 whereas the minimum support is given as 10%.

In the above experiment, it is also shown that with a cluster distribution MRGraphMine can outperform CloseGraph. Considering CloseGraph worsens the computation time as the graph size increases, MRGraphMine is efficient in handling large data in distributed runs.

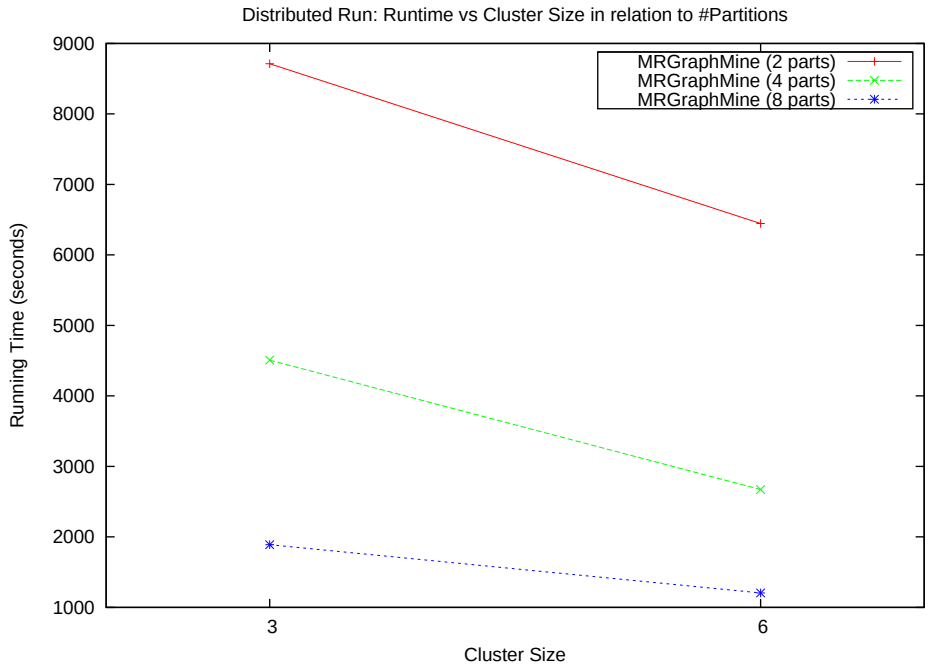

Figure 4.4: Runtime vs Cluster Size in relation to #Partitions.

In the last experiment in Figure 4.4, we try to relate the partitioning criteria to the cluster size. This setup uses the bigger WI-PHI network. If the number of partitions is less than the number of available tasktrackers, MRGraphMine cannot utilize the cluster fully. When, however, the number of partitions exceeds the cluster size (or the total number of tasks that a tasktracker can accept), MRGraphMine consumes the resources in an efficient manner.

4.3 The Limit Of Parallelism

Our method is heavily dependent on a well defined graph partitioning method. When the number of partitions increases for a given graph, the size of each partition shrinks and it gets easier to mine the individual partitions. But more partitions also mean more merge-join

operations. The parallel speedup here is bounded by these two constraints. The study of how much to compromise between the merge-join operations and individual partition size is left for future investigations.

This should also be noted that our method merge-joins the computed patterns in individual partitions. Since merge-join step does not include another invocation of pattern finding algorithm, the overall frequent patterns are comprised of what has been computed at partition level. So, in theory and in practice, our method cannot find any frequent pattern whose size is bigger than the smallest partition.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusions

Frequent pattern mining in graph data remains an interesting problem in many domains and disciplines. Because of the observation that the complexity of graph data is generally much greater than other abstract representations and the fact that the inherent problems in processing graphs such as graphs isomorphism and enumeration stands as a blocker for developing efficient methods, the need for a distributed and scalable graph processing is in high demand. As graphs are very expressive in showing many relations between countless types of objects, mining graphs will help discover unknown but crucial information in every discipline.

In our thesis, we devise a new distributed and scalable algorithm for mining graph data. We use labelled protein-protein interaction networks as our input and try to find the set of frequent patterns described in labels.

Distribution of our method is heavily dependent on the Map/Reduce paradigm. We use open source libraries to implement the algorithm detailed in this thesis work. Using Hadoop[3] and HBase[5], we manage to create a reliable and scalable computing environment. We expressed our data in Hbase as tables and run Hadoop Map/Reduce jobs on this data for several operations. As the system is built on top of a distributed file system, the reliability of data and horizontal scalability to large data size is done inherently. The algorithms defined in this thesis, i.e. partitioning, mining and merging, are all written as Map/Reduce jobs which helps scaling the computations horizontally with the cluster size.

At the core of our implementation we use an existing bi-partitioning method for dividing the main graph into smaller and more manageable units. Later in the mining process we

merge-join the calculated patterns from these graphs and construct them losslessly. To mine the individual partitions of the main graph, we utilize a well known pattern mining method CloseGraph.

We show that although for relatively small graphs our method does not outperform the memory based CloseGraph on a single machine, as the data size is increased our method shows more increasing performance compared to memory based methods. We also propose that when invoked on a medium sized computing cluster, our method outperforms the existing single machine counterparts; CloseGraph and PartMiner.

5.2 Future work

There are basically two popular approaches dealing with the distribution of graph computation. Being a very high level and easy distributed computing environment and having a production ready environment, Map/Reduce sticks out as a very competitive contender. In one of the popular areas, matrix computation, there has been considerable amount of work in distributing the workload using MR. For instance, the computation of a matrix multiplication operation;

$$A_{i,k} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,j} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i,1} & a_{i,2} & \cdots & a_{i,j} \end{pmatrix} x \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,k} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{j,1} & b_{j,2} & \cdots & b_{j,k} \end{pmatrix}$$

can be very easily parallelized. The resulting matrix's cells are just independent multiplications and sums of rows and columns of two input matrices. So any unit (a map task or a reduce task) could take the calculation of one such cell. Since matrix operations can be efficiently expressed as MR jobs and graphs are basically matrices, a memory based graph mining algorithm working on matrices could thus be transformed into a MR program. An open source project from Apache Foundation, *Hama*[4] provides necessary data types and operations for matrices using Map/Reduce and HBase. Existing methods to mine incident matrices are left as a future work for this type of mining.

Another parallelization approach has surfaced recently for graphs. A method called *Bulk Synchronous Parallel (BSP)*[19] is gaining popularity. A BSP cluster is basically a group

of processors (computers) connected with a communication network. Each computer should have a fast processor and a good amount of local memory as the idea of BSP is that the computation runs only on local data. A BSP program is simply a list of supersteps where a superstep is defined as the following three steps:

1. every computer processes local data,
2. computers wait for a barrier synchronization,
3. nodes communicate with each other.

Theoretically, a many-Map/Reduce-job program could be written as a single BSP program that runs many supersteps. This is suitable for graph processing as MR is more tailored towards one-shot computation of embarrassingly parallel problems. However, graph processing often requires back and forth data flow which is why MR programs may need to run multiple jobs when dealing with graphs. We do not choose BSP in our study as there is no production ready and fault tolerant open source implementation. Using BSP as the distributed computation framework is left as a future work.

REFERENCES

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, 51:107–113, 2006.

[2] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, 2008.

[3] A. S. Foundation. Hadoop core project. http://hadoop.apache.org/, last visited on 20/01/2010.

[4] A. S. Foundation. Hama project. http://hadoop.apache.org/hama, last visited on 20/01/2010.

[5] A. S. Foundation. Hbase project. http://hadoop.apache.org/hbase, last visited on 20/01/2010.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[7] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. *ICDM '09: Proceedings of the 2009 IEEE International Conference on Data Mining*, 2009.

[8] L. Kiemer, S. Costa, M. Ueffing, and G. Cesareni. Wi-phi: A weighted yeast interactome enriched for direct physical interactions. *Proteomics - Clinical Applications*, 7:932–943, 2007.

[9] M. Koyutürk, A. Grama, and W. Szpankowski. An efficient algorithm for detecting frequent subgraphs in biological networks. *Bioinformatics*, 20:200–207, 2004.

[10] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Trans. on Knowl. and Data Eng.*, 16:1038–1051, 2004.

[11] M. Kuramochi and G. Karypis. Grew-a scalable frequent subgraph discovery algorithm. *ICDM '04: Proceedings of the Fourth IEEE International Conference on Data Mining*, pages 439–442, 2004.

[12] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph*. *Data Min. Knowl. Discov.*, 11:243–271, 2005.

[13] M. Kuramochi and G. Karypis. Discovering frequent geometric subgraphs. *Inf. Syst.*, 32:1101–1120, 2007.

[14] Y. Liu, X. Jiang, H. Chen, J. Ma, and X. Zhang. Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. *APPT '09:*

Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies, pages 341–355, 2009.

[15] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, 51:6–6, 2009.

[16] S. N. Nguyen, M. E. Orlowska, and X. Li. Graph mining based on a data partitioning approach. *ADC '08: Proceedings of the nineteenth conference on Australasian database*, pages 31–37, 2007.

[17] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26:1367–1372, 2004.

[18] J.-y. Peng, L.-m. Yang, J.-x. Wang, Z. Liu, and M. Li. An efficient algorithm for detecting closed frequent subgraphs in biological networks. *BMEI '08: Proceedings of the 2008 International Conference on BioMedical Engineering and Informatics*, pages 677–681, 2008.

[19] U. University. Bsp worldwide. http://www.bsp-worldwide.org/, last visited on 20/01/2010.

[20] J. Wang, W. Hsu, M. L. Lee, and C. Sheng. A partition-based approach to graph mining. *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 74, 2006.

[21] J. Wu and L. Chen. A fast frequent subgraph mining algorithm. *ICYCS '08: Proceedings of the 2008 The 9th International Conference for Young Computer Scientists*, pages 82–87, 2008.

[22] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 3:433–444, 2008.

[23] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295, 2002.

[24] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining*, page 721, 2002.

[25] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Coherent closed quasi-clique discovery from large dense graph databases. *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 797–802, 2006.