ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

DISTRIBUTED BIPARTITE GRAPH CLUSTERING

M.Sc. THESIS

Resul TUGAY

Department of Computer Engineering

Computer Engineering Programme

JUNE 2018

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

DISTRIBUTED BIPARTITE GRAPH CLUSTERING

M.Sc. THESIS

Resul TUGAY
(504151522)

Department of Computer Engineering

Computer Engineering Programme

Thesis Advisor: Prof. Dr. Şule GÜNDÜZ ÖĞÜDÜCÜ

JUNE 2018

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

DAĞITIK İKİ PARÇALI ÇİZGE DEMETLEME

YÜKSEK LİSANS TEZİ

Resul TUGAY
(504151522)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Prof. Dr. Şule GÜNDÜZ ÖĞÜDÜCÜ

HAZİRAN 2018

Resul TUGAY, a M.Sc. student of ITU Graduate School of Science Engineering and Technology 504151522 successfully defended the thesis entitled "DISTRIBUTED BIPARTITE GRAPH CLUSTERING", which he/she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Prof. Dr. Şule GÜNDÜZ ÖĞÜDÜCÜ** Istanbul Technical University	.....
Jury Members :	**Assoc. Prof. Dr. Berk CANBERK** Istanbul Technical University	.....
	Asst. Prof. Dr. Nagehan İLHAN Harran University	.....
		.....

Date of Submission : **04 May 2018**
Date of Defense : **07 June 2018**

To my spouse,

FOREWORD

MSc. at Istanbul Technical University was an incredible experience for me. It was a milestone to be a part ITU Comp.Eng just after bachelor. I am very thankful to the faculty of ITU for believing in me from the beginning. Lots of important things have changed in my life after getting into MSc. I got married with my lovely wife Burcu and I had the opportunity to become research assistant at ITU. Being a research assistant added a lot to me. Teaching recitations and researching deeply were the most valuable experiences. The most instructive and funny projects were Database Management System and Operating System term projects. I would like to say that was great experiencing to be teaching assistant of these lectures.

I am grateful to my advisor Prof. Dr. Şule Gündüz Öğüdücü. She patiently guided me whenever l lost in somewhere during research and also gave me an opportunity working with real world problems with big companies.

June 2018

Resul TUGAY
(Research Assistant)

TABLE OF CONTENTS

Page

ABBREVIATIONS

SVD	**:** Singular Value Decomposition
PCA	**:** Principal Component Analysis
TF	**:** Term Frequency

SYMBOLS

$\mathbf{G}$	: Graph
$\mathbf{V}$	: Vertex Set
$\mathbf{E}$	: Edge Set
$\mathbf{A}$	: Adjacency Matrix
$\mathbf{F}$	: Matrix
$\mathbf{X}$	: Left Side of Bipartite Graph
$\mathbf{Y}$	: Right Side of Bipartite Graph
$\mathbf{W}$	: Symmetric Adjacency Matrix
$\mathbf{DM}$	: Diagonal Matrix of Symmetric Adjacency Matrix
$\mathbf{w}_{i,j}$	: Weight of Edge Between Vertex i in X and Vertex j in Y
$\mathbf{L}$	: Laplacian Matrix
C_1,C_2	: Sub Graphs After Bipartite Partitioning
$\mathbf{d}$	: Rank of a matrix
$\mathbf{P}$	: Proxy Matrix
$\mathbf{M}$	: Number of Rows
$\mathbf{N}$	: Number of Columns
$\mathbf{D}$	: Number of Blocks
$\mathbf{u}_i$	: ith Singular Left Vector
$\mathbf{v}_i$	: ith Singular Right Vector
Σ	: Singular Values
$\mathbf{S}_i$	: Constraint Set of vertex i

LIST OF TABLES

Page

LIST OF FIGURES

Page

DISTRIBUTED BIPARTITE GRAPH CLUSTERING

SUMMARY

Bipartite graphs are graphs whose vertices can be divided into two independent sets (X and Y) such that no two graph vertices within the same set are adjacent. There are many real-world examples which display natural bipartite structure. For instance, a bipartite graph where the distinct set of vertices are people and books and edges represent buying relationship between them. Another prominent example is a bipartite graph where the vertex sets are candidates and jobs and if a candidate applies to job, there is an edge between the corresponding vertices.

In this thesis, we implement a distributed bipartite clustering method for large and sparse data. There are several approaches to cluster data which is bipartite in nature. For example k-means clustering method can be applied to the both sides of the graph separately. But bipartite clustering requires a clustering method that clusters data simultaneously. Spectral clustering is a way to cluster bipartite data simultaneously which means assigning vertices from both sides into the same coherent cluster. We use multi bipartitioning clustering algorithm which is based on spectral clustering in this thesis along with some other bipartite partitioning algorithms such as Bifennel, Aweto and etc.

There are some steps that need to be done before bipartite graph is clustered. Firstly, bipartite graph is converted into adjacency matrix where rows represent one set of vertices of bipartite graph and columns represent the other set of vertices. If there is an edge between two vertices, then the value of the corresponding cell of the adjacency matrix is assigned to 1, otherwise it is assigned to 0. After having the adjacency matrix, the Laplace form of the adjacency matrix is created. This matrix is also known as degree matrix. Degree matrices namely D_1 and D_2 are constructed for X and Y respectively. Then adjacency matrix is multiplied with D_1 and D_2. Lastly, the spectral algorithm runs Singular Value Decomposition (SVD) on the formed adjacency matrix and finds second singular left and right vectors. After finding second singular right and left vectors, k-means clustering algorithm is run on these vectors to cluster data. This method only partitions the data into two cluster. But finding other singular vectors (2.,3.,...) other than only the second one, more clusters can be obtained.

In this thesis, we used distributed SVD algorithm to find singular left and right vectors of adjacency matrix when the matrix is large and sparse. Distributed SVD algorithm divides the matrix into column-wise sub matrices and calculates SVD on these sub matrices independently. It is capable of finding the singular values, left and right vectors when the matrix is rectangular (number of rows are much more smaller than number of columns or vice versa) and dense. In other words, the rank of the sub matrices must be equal to the rank of the adjacency matrix itself. Because of the sparsity of the data, the rank that is less in sub matrices than the adjacency matrix

causes to find singular values and vectors with high error. We proposed an algorithm, called *Ranky*, to handle this problem. Ranky algorithm detects the rows which cause the rank of the sub matrices less then the rank of the adjacency matrix and fills accordingly in order to make them equal. Despite the Ranky algorithm can handle this problem, balanced clusters can not be obtained because of the sparse matrix. After that, we used partitioning algorithms to compress the adjacency matrix into dense matrix and then applied spectral clustering. There are several partitioning algorithms such as Hash, Grid, Aweto, Bifennel etc. But only Aweto and Bifennel take the advantage of bipartite graphs as we know. We use Bifennel partitioning algorithm and Distributed Bifennel algorithm to compress the data due to their efficiency and less replication factor. Lastly, we decompress the partitions to get final clusters.

The evaluation criteria is the replication factor that measures partitioning algorithm performance in terms of the number of replicated vertices. The experimental results on a real world data set show that Distributed Bifennel algorithm has outperformed Bifennel algorithm in terms of execution time, while both algorithms produce equally likely replication factor, using different datasets. Each algorithm is run on the same datasets with different partitioning numbers (500 and 1000). Distributed Bifennel algorithm has partitioned the data 10 times faster than Bifennel using at most 28 threads. The algorithm was implemented in C++ and run on a 28 cores and 128 GB RAM machine running Linux. More experimental results with different settings are given in chapter 5.

DAĞITIK İKİ PARÇALI ÇİZGE DEMETLEME

ÖZET

Bu tez kapsamında iki parçalı büyük veriler için sunucu-istemci mimarisi kullanılarak çizge demetleme yapılmıştır. İki parçalı çizgede iki farklı düğümler kümesi (X ve Y) vardır. Aynı kümedeki düğümler arasında ayrıt bulunmazken, farklı kümelerdeki düğümler arasındaki ilişkileri ifade etmek üzere ayrıtlar vardır. Örneğin satın alınan kitap, film, müzik ya da başvurulan iş ilanları iki parçalı çizgenin bir düğümler kümesini oluştururken bunları satın alan veya başvuru yapan kişiler çizgenin diğer düğümler kümesini oluşturur. Günümüzde artan İnternet kullanımı ile İnternet ortamındaki veriler de bir hayli artmıştır. Bu verileri değişik amaçlar için tek bir makinede işlemek zor bir hal almıştır. Bu tez çalışmasındaki amaç da iki parçalı çizge demetleme problemini paralel yaklaşımlarla çözmektir.

İki parçalı çizge demetleme aralarında diğer gruplara göre daha yoğun ilişkiler bulunan farklı kümelerdeki düğümleri gruplamaktır. Böylece örneğin benzer türde müzikleri tercih eden kullanıcı grubunu bulmak ya da benzer iş ilanlarına başvuran adayları bulmak mümkün olabilmektedir. Düğümler kümesi çok büyüdüğünde literatürde var olan yöntemlerle iki parçalı çizge demetleme problemini standart donanımlar üzerinde çözmek mümkün olmamaktadır. Bu tez ile büyük veride iki parçalı çizge demetleme problemini çözmek için sunucu-istemci mimarisi ile paralel çalışan yaklaşımlar önerilmiş ve büyük iki parçalı çizge demetleme yapabilen bir yazılım geliştirilmiştir.

Öncelikle iki parçalı verinin bölütlenmesi için literatürde spektral demetleme adı verilen bir algoritmadan yararlanılmıştır. Algoritmanın birkaç varyasyonu olmakla birlikte, bu tezde simetrik olmayan komşuluk matrisi üzerinde Tekil Değer Ayrıştırması (SVD) uygulanarak bulunan demetlemeden yararlanılmıştır. Diğer varyasyonlar komşuluk matrisinin simetrik olduğu durumlarda çalışabildiğinden ve elimizdeki komşuluk matrisinin dikdörtgen olmasından dolayı bu tez kapsamında diğer varyasyonlar kullanılmamıştır. Algoritma temel olarak iki parçalı veriyi satırları bir küme sütünları ise diğer küme ve değerleri aralarındaki ilişki olan komşuluk matrisi olarak tanımlar. Komşuluk matrisinin değerleri basit olarak eğer bir kümedeki bir elemanın diğer kümedeki diğer elemanla ilişkisi varsa "1" yoksa "0" olarak belirlenir.

İki parçalı çizgeden A komşuluk matrisi oluşturulduktan sonra, algoritma aşağıdaki gibi devam eder.

1. A komşuluk matrisinden aşağıdaki gibi K matrisi elde edilir: $K = D_1^{-1/2} \mathrm{A} D_2^{-1/2}$
2. K matrisinin ikinci sağ (u_2) ve sol (v_2) tekil değerlerini hesaplanır ve aşağıdaki Z matrisi formuna getirilir: $\mathrm{Z} = \begin{vmatrix} D_1^{-1/2} u_2 \\ D_1^{-1/2} v_2 \end{vmatrix}$
3. Z matrisi üzerinde k-ortalama demetleme algoritması koşularak demetler bulunur.

Burada bahsedilen D_1 ve D_2 matrisleri iki parçalı çizgenin sırasıyla ilk ve ikinci kümelerinin derece matrisleridir. Eğer algoritma birden fazla boyutta demetlenmek istenirse, 2. adımda bulunan sağ ve sol tekil vektörler sadece ikinci vektörler değil, sırasıyla 3., 4., .. sağ ve sol tekil vektörlerden oluşmalıdır. Bu şekilde Z matrisi tek boyutlu değil birden fazla boyutlu olur ve k-ortalama demetleme algoritması bu çok boyutlu matris üzerinde koşturularak demetleme yapılır.

Bazı veri kümeleri için iki parçalı çizgede X'in eleman sayısı Y'nin eleman sayısından bir hayli fazladır, bu da elde edilen A komşuluk matrisinin dikdörtgen yapıda olmasına sebep olur. Literatürde bu tip dikdörtgen matrisler üzerinde çalışabilen dağıtık SVD algoritması önerilmiştir. Algoritma sunucu-istemci modeli ile çalışmaktadır. Verilen matriste satır sayısının kolon sayısından oldukça az olduğu düşünülürse, sunucu A komşuluk matrisini kolon-temelli bölerek herbiri aynı sayıda satıra sahip olan küçük matrisleri (A_1,A_2,..) istemci makinalara gönderir. İstemci makinalar bağımsız olarak SVD algoritmasını bu küçük matrisler üzerinde koşar ve elde ettiği tekil değerleri, sol ve sağ vektörleri sunucu makineye geri gönderir. Sunucu, istemcilerin elde ettikleri tekil değerleri ve tekil vektörleri toplayarak birleştirir ve A matrisinin tekil değer ve vektörlerini bulur. Daha sonra spektral algoritma k-ortalama demetleme algoritmasıyla birlikte veriyi bölütler. Fakat bu algoritma sadece yoğun karesel matrisler üzerinde doğru sonuçlar vermektedir. Çünkü bölünen matrisler ile A matrisinin kertelerinin (rank) eşit olması gerekmektedir, Fakat elde ettiğimiz iki parçalı çizge bir hayli seyrektir ve matris bölünürken küçük matrislerin kerteleri komşuluk matrisinin A kertelerinden düşük olabilmektedir.

Bu tez çalışmasında yukarıda bahsedilen problemi çözmeye yönelik yöntemler önerilmiştir. Önerilen *Ranky* isimli algoritma 3 farklı yaklaşımdan oluşmaktadır. Öncelikle A komşuluk matrisi bölünürken küçük matrislerde (A_1,A_2,..) kertenin daha düşük olmasına sebebiyet veren satırlar bulunur ve bu satırların kolonlarından birisine rastgele "1" eklenerek bu satırın düşük kerteye sebep olması önlenir. Bu işlemde satırların bulunmasının nedeni komşuluk matrisinin kertesini belirleyen, sayısı kolonların sayısından çok çok az olan satır sayısıdır. İkinci olarak iki parçalı çizge demetleme yaptığımız için, rastgele "1" ya da X'teki elemanlar ile Y'de elemanlar arasına rastgele ilişki eklemek yerine, bu satırların diğer küçük matrislerdeki komşuları (diğer satırlar) bulunur ve bu komşuların ilişkisi bulunan kolonlardan rastgele bir kolon seçilir. Fakat elimizdeki veride kertenin düşük olmasına sebep veren birden fazla satır olabilir, ve bu satırların komşuları benzer olabilir, bu şekilde rastgele atanan "1" ya da ilişki aynı kolona denk gelebilir ki yaptığımız deneylerde bu sonuçlarla karşılaştık. Bu yüzden 3. yöntem olarak öncelikle 2. yöntem olan komşuluk yöntemini uygulayıp daha sonra ilk yöntem olan rastgelelik yöntemini uygulayarak bu problemi çözüyoruz.

Önerilen bu yöntem kerte problemini çözüme kavuşturmuştur, fakat elde ettiğimiz demetleme sonuçlarında veri büyük ve seyrek olduğundan tutarsızlık gözlemlenmiştir. Bundan dolayı veriyi seyrek halden daha yoğun bir hale getirdikten sonra dağıtık SVD algoritması uygulanmıştır. Bu yöntem ile daha tutarlı iki parçalı demetler elde edilmiştir. Veri seyrek halden daha yoğun bir hale bölütleme algoritmaları kullanılarak getirilmiştir. Basit olarak aynı düğümler kümesindeki benzer olan bazı düğümler birleştirilerek tek bir düğüm olarak temsil edilmiştir. Daha yoğun hale getirilen bu veriye SVD algoritması uygulandıktan sonra birleşik olarak temsil edilen düğümler tekrardan ayrıştırılırak bölütleme tamamlanmaktadır. Örneğin elimizdeki iki parçalı çizgede örneğin X kümesinde 1.000.000 ve Y kümesinde 100.000 düğüm var ise, bu

yöntemden sonra elimizdeki matriste toplam X kümesinden 10.000 Y kümesinde ise 1000 düğüm olabilmektedir. Bu sayılar parametre olarak değişebilmektedir. Birkaç düğüm tek bir düğüm ile gösterildikten sonra A komşuluk matrisi yeni bağlantıları gösterecek şekilde yeniden oluşturulur. Örneğin X kümesindeki $i1$, $i2$, $i3$ düğümlerinin sırasıyla Y kümesindeki $a1$, $a2$, $a3$ düğümleri ile ilişkisi olsun. Eğer algoritma bu üç düğümü ($i1$, $i2$ ve $i3$) birleşik düğüm (yenidüğüm1) olarak temsil etmiş ise yeni matriste bu yenidüğüm1'in diğer üç ($a1$, $a2$, $a3$) düğüm ile bağlantısı olacaktır. Bu şekilde önceki matris seyrek olmaktan çıkıp daha yoğun bir hale gelebilmektedir. Literatürde birçok çizge bölütleme algoritması olmakla birlikte biz bu tez kapsamında düğüm-bölmeli (vertex-cut) iki parçalı çizge bölütleme algoritmaları olan Hash,Grid, Bicut, Aweto ve Bifennel algoritmalarını kullandık.

Bu algoritmaların başarımı tekrarlama faktörü (replication factor) denilen ve toplam tekrarlanan düğümlerle ile toplam düğüm sayısının, toplam düğüm sayısına oranı ile hesaplanmaktadır. Hash tabanlı algoritma iki parçalı demetin en fazla düğüm içeren kümesindeki düğümlerin numaralarını belirlenen bir hash fonksiyonundan geçirerek bölütleme sağlar. Örneğin X kümesinde 4 eleman ($i1$,$i2$,$i3$,$i4$) ve Y kümesinde 9 eleman ($a1$,$a2$,$a3$,...,$a9$) var ise, algoritma bu 9 elemanın numaralarını alır ve hash fonksiyonuna parametre olarak verir. Hash fonksiyonu bölütleme sayısına göre değişebilmektedir. Varsayalım ki burada 3 tane bölüt olsun. Hash fonksiyonu $H(x) = x\%3$ olur, fonksiyonda belirtilen x düğüm numarasıdır. Bu şekilde $i1$,$i4$,$i7$ nolu düğümler bir bölüte 2,5,8 nolu düğümler diğer bölüte ve son olarak $i3$,$i6$,$i9$ nolu düğümler ise diğer bölüte atanır. Daha sonra her küme sanki yeni bir düğümmüş gibi davranılır ve komşuluk matrisi yeniden oluşturulur. Fakat görüldüğü gibi bu yöntem hiçbir şekilde düğümlerin komşuluk ilişkilerini göz önünde bulundurmamaktadır. Hatta eğer düğüm numaraları çok farklı ise bölütler de dengesiz olabilmektedir. Diğer algoritmalar bu temel bölütleme algoritmasının göz önünde bulundurmadığı komşuluk ilişkilerini ve bölütlerin dengesini göz önünde bulundurarak bölütleme yapar. Bu algoritmalar arasında Bifennel algoritmasının, önerildiği makalede en düşük tekrarlama faktörüne sahip olduğu deneylerle ispat edilmiştir. Bu algoritma hem komşuluk ilişkilerini göz önünde bulundurmuş hemde bölütlerdeki dengenin (düğüm sayısının) yaklaşık eşit olmasını sağlamıştır. Bu tez kapsamında da Bifennel algoritması gerçeklenerek çift parçalı çizge demetleme sonuçları sunulmuştur.

1. INTRODUCTION

There are many types of graphs including simple, weighted, cyclic and acyclic, labeled etc. in literature. A special type of graph is called bipartite graph ($G = (X, Y)$) that is used in many applications from dating website to book selling e-commerce site. It consists of a set of vertices which decomposed into two disjoint sets (X and Y) such that no two vertices within the same set are adjacent. For example people and books are two different independent sets that only have links between them and they represent buying relationship between books and customers. Customers can only buy books or books can be bought by only customers. This type of relationship can be represented with a $(0, 1)$ adjacency matrix A of size $|X| \times |Y|$ where each row of the matrix is a vertex of the first set and each column is a vertex of the second set. In this matrix, the cell A_{ik} where i corresponds to a person and k corresponds to a book is 1 if there is an edge between vertex i and k, otherwise it is 0.

In this thesis, we focused on a distributed bipartite graph clustering method which copes with the big bipartite graph that can not be effectively clustered on a single machine. Though, we implemented a server-client architecture to cluster the given bipartite graph. There are spectral clustering methods based on distributed SVD algorithm to cluster big bipartite graphs. However, these methods are not successful to deal with sparse matrices since distributed SVD algorithm can not compute singular values and singular left and right vectors correctly for this kind of matrices.

We proposed three different methods to address the problem. Although we have solved this problem, the bipartite clustering result has become unbalanced due to sparsity of the data matrix. Then we have applied bipartite partitioning algorithms before applying distributed SVD on the matrix. We achieved compressed and dense matrix by applying the partitioning algorithms on the data matrix and applied distributed SVD algorithms. We get balanced clusters with this strategy. There are many partitioning algorithms such as Hash, Grid, Bicut, Aweto, Bifennel etc. Replication factor which is ratio of replicated vertices with all vertices to all vertices and execution time are evaluation

criteria of partitioning algorithms. Bifennel algorithm has the best performance according to the paper [1] in terms of partitioning processing time and replication factor. We applied distributed version of Bifennel bipartite partitioning algorithm and had balanced cluster as well as less replication factor.

The rest of the thesis is structured as follows: In Section 2 we present background and in Section 3 we give details about Ranky algorithm. Section 4 shows partitioning algorithms and 5 shows our analysis and results. Finally, Section 6 concludes the thesis.

2. BACKGROUND

In this chapter, we will give background of thesis which is based on distributed bipartite clustering. We will first explain bipartite clustering in detail, then give the methods used to cluster bipartite graphs such as spectral clustering. We will also discuss other partitioning algorithms such as Bifennel, Aweto, Random and Grid in chapter 4. We will use following definitions through this thesis :

- $G(V,E)$ represents a graph G where V is the vertex set and E is the edge set of the graph.
- X and Y represent independent sets of the bipartite graph.
- A represents adjacency matrix of the bipartite graph.
- $w_{i,j}$ denotes the weight of the edge between vertex in X and vertex in Y.
- C_1 and C_2 are subgraphs after bipartite partitioning

2.1 Bipartite Clustering

Data clustering is the method of identifying a group of samples in such a way that samples in the same group are more and more similar to each other rather than others in different groups. There are two different kinds of objects to be clustered simultaneously in bipartite clustering. Formally, assume that $G(V,E)$ is a graph where V is the vertex set and E is the edge set of the graph. A graph is bipartite graph if and only if whose vertices can be divided into two disjoint sets X and Y and each edge has one endpoint in X and one endpoint in Y. This type of graph can be seen in Figure 2.1. Bipartite graphs require different algorithms to be clustered because of their structure. Next we will discuss spectral clustering which is used to cluster bipartite graphs.

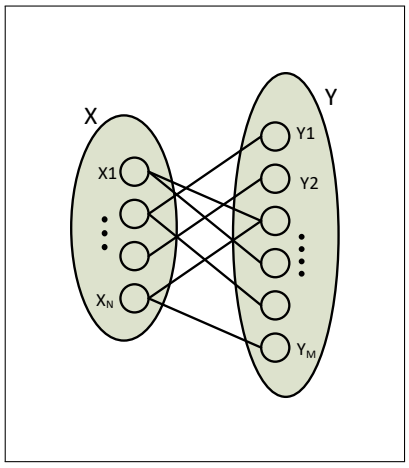

Figure 2.1 : Bipartite Graph Example.

2.1.1 Min-max cut and spectral clustering

Min-max cut algorithm partitions bipartite graphs into two subgraphs C_1 and C_2 such that sum of edge weights connecting vertices in C_1 to vertices in C_2 is minimum and the sum of edge weights within C_1 and C_2 is maximum. Thus, this algorithm tries to minimize similarity between clusters and maximize the similarity within a cluster [2]. The similarity between C_1 and C_2 is as follows:

$$cut(C_1,C_2) = W(C_1,C_2) \tag{2.1}$$

where $W(C_1,C_2)$ is the sum of all edges weights between subgraph C_1 and subgraph C_2. Min-max cut algorithm minimizes $W(C_1,C_2)$ while maximizing both $W(C_1)$ and $W(C_2)$ where $W(C_1)$ is the sum of all edge weights in $W(C_1)$ such that $W(C_1) = W(C_1,C_1)$. In the light of these, min-max cut objective function is defined as below:

$$Mcut(C_1,C_2) = \frac{cut(C_1,C_2)}{W(C_1)} + \frac{cut(C_1,C_2)}{W(C_2)} \tag{2.2}$$

The objective function in Equation 2.2 is called normalized objective function. Spectral clustering is a relaxation of these objective functions. Let DM be the diagonal degree matrix of symmetric adjacency matrix (W). If the graph is undirected, the adjacency matrix (A) is symmetric. Otherwise symmetric adjacency matrix (W) can be found as follows:

$$W = \begin{pmatrix} 0 & \mathrm{A} \\ A^T & 0 \end{pmatrix} \tag{2.3}$$

Laplacian (L) matrix is defined as $L = DM - W$ where DM is diagonal degree matrix and W symmetric adjacency matrix. Let $e = (1,1,1..1)$ be a vector, then $Le = 0$, hence

0 is an eigenvalue and e is the corresponding eigenvector of L. Min-max cut can be represented with DM and W. For any partitioning vector $x = (1,1,1,..-1,-1,-1)$ and $y = (-1,-1,-1,..1,1,1)$ partitioned with C_1 and C_2 respectively, the $cut(C_1,C_2)$ can be re-defined as follows:

$$cut(C_1,C_2) = x^T(DM-W)x = y^T(DM-W)y \tag{2.4}$$

$x^T(DM-W)x$ or $y^T(DM-W)y$ in Equation 2.4 leads to find eigenvalues and eigenvectors of Laplacian form of symmetric adjacency matrix. Thus, spectral clustering algorithm can be defined as follows:

Algorithm 1 Spectral Clustering Algorithm

Require: Symmetric adjacency matrix W

1. Compute Laplacian L (DM-W) form of adjacency matrix.
2. Compute eigenvalues and eigenvectors of laplacian matrix L.
3. Take the second smallest eigenvalue and its related eigenvector
4. Partition vertices into two partition by separating the values in the vector as negative and positive values

Spectral clustering divides the graphs into two subgraphs by minimizing the objective function. However, this algorithm can be used to recursively bisection the subgraphs. On the other hand, Dhillon [3] proposed another algorithm called multipartitioning that is based on spectral clustering which can partition graph into more than two subgraphs which is k. He also used singular value decomposition of Laplacian form of symmetric adjacency matrix instead of finding eigenvalues and eigenvectors. By doing so, there is no longer need to keep the symmetric matrix of adjacency matrix. In this thesis, we are using multipartitioning algorithm instead of using recursive spectral clustering algorithm. Multipartition algorithm is using k-means clustering algorithm when separating the values of second smallest eigenvector known as Fiedler vector. The algorithm is given in Algorithm 2.

In multipartitioning algorithm, SVD is used instead of eigenvalue decomposition to find the Fiedler vector that is related to second smallest singular value of laplacian form of adjacency matrix. Because of its importance next section we will explain Singular Value Decomposition (SVD) in detail.

Algorithm 2 Multipartitioning(k) Algorithm

Require: Adjacent matrix A

1. Compute $D_1^{-1/2}AD_2^{-1/2}$ where D_1 is the degree matrix of X and D_2 is degree matrix of Y
2. Compute $l = \log_2^k$ singular vectors of $D_1^{-1/2}WD_2^{-1/2}$ where k is the number of cluster
3. form the matrix $D_1^{-1/2}WD_2^{-1/2}$ to Z = $\begin{vmatrix} D_1^{-1/2}U \\ D_1^{-1/2}V \end{vmatrix}$ Where U and V are the left and right singular vectors of the matrix $D_1^{-1/2}WD_2^{-1/2}$
4. Run k-means algorithm on the l- dimensional data Z to get the desired clusters.

2.2 Singular Value Decomposition

The Singular Value Decomposition (SVD) of a matrix F can be defined as factorization or decomposition of the matrix into the product of three different matrices. Formally, the singular value decomposition of a matrix F with M rows and N columns can be represented as follows:

$$F = U\Sigma V^* \tag{2.5}$$

In Equation 2.5, U is a unitary matrix $(U^T = U^{-1})$ with dimensions MxM, V^* (conjugate transpose of V) is also a unitary matrix having NxN dimension and Σ is MxN diagonal matrix with non-negative real diagonal numbers where $\Sigma_{ii} = \sigma_i$ $fori = 1,...,min$(M,N). If the matrix F is real, then U and V are real and orthogonal. The vectors u_i $(i = 1,...,M)$ and v_j $(j = 1,...,N)$ are called the left and right singular vectors respectively and σ_k $(k = 1,...,min(M,N))$ are the singular vectors. In this thesis we assume that the matrix F is "short and fat" where the number of columns are much more than the number of rows. But the matrix can also be "tall and skinny" matrix where number of rows are much more than number of columns.

It is possible to get singular components of F by finding eigenvalues and eigenvectors of cross product matrices (F^*F and FF^*). The left and right singular vectors are the eigenvectors of the matrices respectively and singular values are the nonnegative square roots of the eigenvalues of one of the cross product matrices. Besides, singular components can be found by finding eigenvalues and eigenvectors of a symmetric matrix called cyclic matrix that is constructed as a matrix ($[F|F^*]$) from F and F^*.

But these two approaches are not recommended to compute singular components because of both high cost of the computation of cyclic matrix, especially in the case of sparse matrix. Besides, these methods cause loss of accuracy when computing FF^* [4]. Generally SVD algorithms focused on bidiagonalization step in order to get cross product matrix without computing it explicitly [5]. Householder method is one approach for computation of the bidiagonal form of a given matrix F and Golub-Kahan bidiagonalization or Lanczos bidiagonalization [6] is another approach for this step.

On top of these core algorithms to solve SVD, there are several distributed algorithms which can run simultaneously on both in different or same machines. The complexity of computing the SVD is $O(M^2N)$ or $O(MN^2)$ where $M < N$ or $M > N$ respectively for a matrix of size MxN. These algorithms try to solve the SVD problem with less complexity by using distributed incremental or hierarchical algorithms. Recently, Iwen and Ong [4] also proposed an algorithm to construct SVD of a matrix in a distributed and incremental way. They proved to recover singular components of large, dense and highly rectangular matrices, but not sparse matrices because of rank of the matrix. The algorithm can recover the singular components of the input matrix if its rank is known. When the dimension of a matrix with the size of MxN is considered, rank of the matrix will be at most $\min(M,N)$, which is number of rows in their assumption. Although, Vasudevan and Ramakrishna proposed a hierarchical SVD algorithm for low-rank matrices, matrices were large, dense and inherently low-rank they used [7].

SVD has many useful applications in many fields from data mining to signal processing including dimension reduction [8] , data clustering [3] , PCA [9], image restoration [10] [11]. Although the history of the SVD dates back to 1900, it was first established for general rectangular matrices by Eckart and Young [12] in 1939. Then it has become more and more popular after that year.

The SVD of a matrix can be formulated as an eigenvalue problem. Compared with an eigenvalue problem, it only works on some of square matrices, but SVD can be applied to all types of (square, rectangular) matrices. Input matrix must be transformed to square matrix before the SVD problem is considered as eigenvalue problem. There are two possible ways to achieve this:

- The cross product matrix, either F^*F or FF^*

- The cyclic matrix $H(F) = \begin{bmatrix} 0 & F \\ F^* & 0 \end{bmatrix}$

Roman et. al stated that these two approaches are not feasible to get singular components of non-square input matrix due to their drawbacks detailed in chapter 4 in [13]. Then Golub and Kahan [6] proposed bidiagonalization algorithm to solve the SVD problem. This algorithm produces the partial bidiagonal reduction of input matrix with increasing dimension in each iteration. There are several implementation of this algorithm in literature. Golub *et al.* [14] implemented block version of this method. A good low rank approximation algorithm, a version of lanczos bidiagonalization called one-sided SVD, was proposed by Simon and Zha [15]. Once bidiagonal form of input matrix is calculated, then the singular values of bidiagonal matrix form can be calculated using *QR* algorithm [16]. There is also stable divide and conquer algorithm proposed by Gu and Eisenstat [17] to compute the SVD of lower bidiagonal matrix.

In some big data applications, the input data is represented as a short and fat matrix with a small number of samples having a large set of features or vice-versa. For instance, there are only ten thousands of terms in Wikipedia, while the number of articles has more than 5,5 millions. There are several studies in literature attempted to solve distribute SVD for non square matrices [18–21, 21, 22]. Qu *et al.* proposed a distributed SVD algorithm for tall and skinny matrices and reported the results on synthetic data. Although they have a good accuracy, their algorithm works efficiently when the local matrices have low ranks [18]. Another algorithm proposed in [19] that is based on the algorithm proposed in [18] is using hierarchical *QR* algorithm to solve Principal Component Analysis (PCA) and inherently the SVD problem. This distributed algorithm uses tree-based merge technique to collect and merge R matrices. Iwen and Ong proposed an algorithm called a distributed and incremental algorithm for short and fat matrices. The algorithm is based on server-client architecture. First of all, server splits the adjacency matrix column-wise into sub matrices called as block matrices and distributes them to clients, then each client calculates SVD of its own block matrix separately and sends back the calculated singular values and singular left vector. After that, server merges all received singular values and singular left vectors in such a way to create a matrix called as proxy matrix (P) and recovers SVD of adjacency matrix by calculating SVD of the proxy matrix. Dimension of proxy matrix is much

more smaller than the original input matrix because of highly rectangular matrix. More recently, Vasudevan and Ramakrishna proposed a hierarchical SVD algorithm which can recover only singular values and left or right singular vectors like in the algorithm proposed in [7]. This algorithm is not working with only short and fat or tall and skinny matrices also all types of other matrices. Further, they split the matrix into block matrices, both row-wise and column-wise unlike the Iwen and Ong algorithm. But this algorithm is suitable for the dense and low rank matrices. We will give more details about distributed and incremental algorithm in the section 3.

3. RANKY ALGORITHMS

3.1 Incremental and Hierarchical SVD Algorithm

There are several algorithms such as householder reflection, Golub-Lanczos algorithm to solve the problem of singular value decomposition. With the help of the today's social media or giant Internet websites, the size of data produced is enormous. For instance, there are hundreds of millions tweets per day, more than this number is produced in facebook and instagram posts by the billions of users. SVD plays a critical role for this type of huge data produced by such sites. SVD can be used in many applications as mentioned before. However, the run time of the SVD can sometimes be really long depending upon the size of the matrix. Incremental and hierarchical algorithms compute huge matrices in a distributed and incremental way. It is a top layer above the SVD algorithms for highly rectangular matrices. The idea is relatively simple, it first splits the matrix into the independent blocks, column-wise if the number of rows is much more smaller than the number of columns in the matrix, row-wise if vice versa, then applies a SVD algorithm on each block, lastly combines the partial results from each block. All these steps can be seen in Figure 3.1. Firstly, the fat and short matrix (as known as highly rectangular) $F \in \mathbb{C}^{MxN}$ is divided into separate blocks where $F = [F^1|F^2|...|F^D]$ as shown in Figure 3.1. Since F^i has a rank at most $d \in \{1,...,M\}$, each block has a reduced SVD representation,

$$F^i = \sum_{j=1}^{d} u_j^i \sigma_j^i (v_j^i)^* = \hat{U}^i \hat{\Sigma}^i \hat{V}^{i*} \tag{3.1}$$

Let $P = [\hat{U}^1\hat{\Sigma}^1|\hat{U}^2\hat{\Sigma}^2|...|\hat{U}^D\hat{\Sigma}^D]$ be the proxy matrix of F. If F has the reduced SVD decomposition, $F = \hat{U}\hat{\Sigma}\hat{V}^*$ and P has the reduced SVD decomposition, $P = \hat{U}^\iota\hat{\Sigma}^\iota\hat{V}^{\iota*}$, then $\hat{\Sigma} = \hat{\Sigma}^\iota$, and $\hat{U} = \hat{U}^\iota WB$ where WB is a unitary block diagonal matrix. the singular

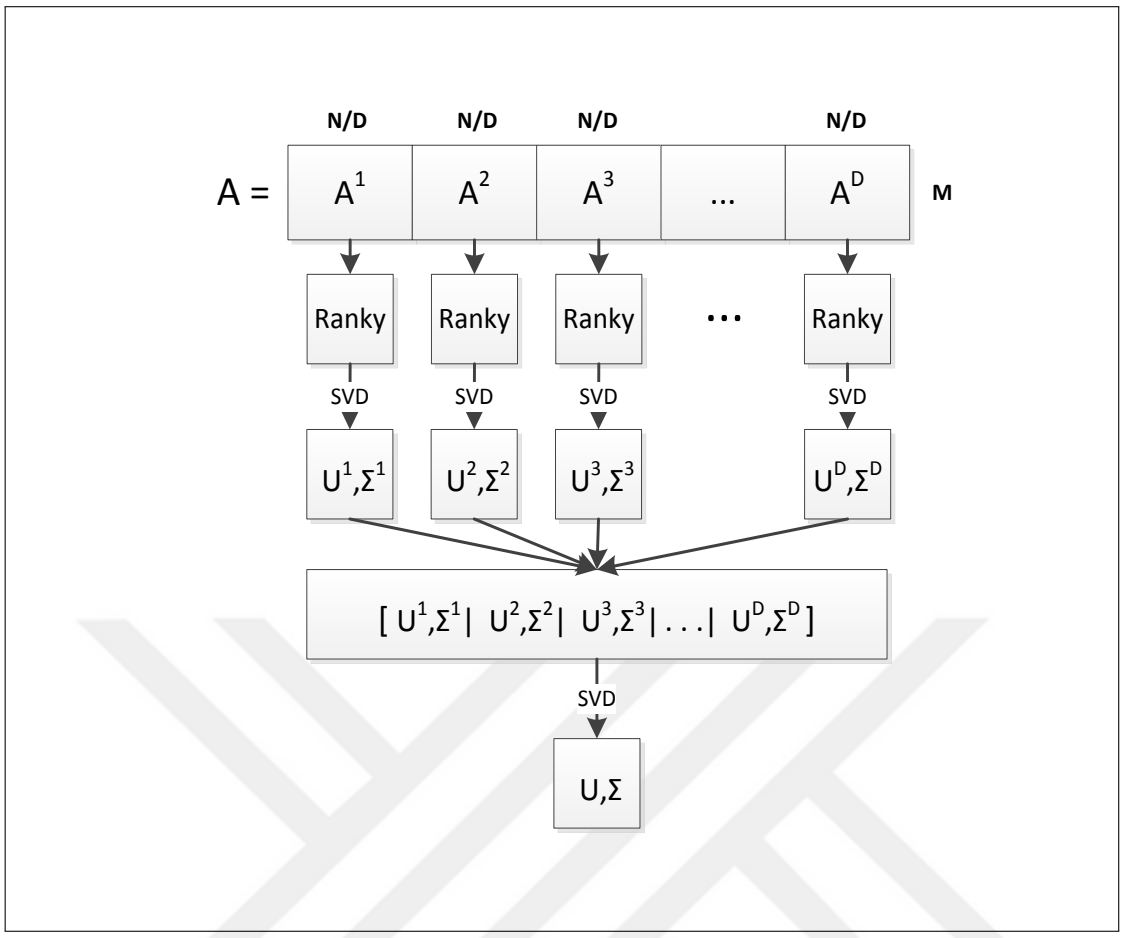

Figure 3.1 : Distributed SVD And Ranky Algorithm.

values of F are also the (non-negative) square root of the eigenvalues of FF^*. Then,

$$\begin{aligned} FF^* &= \sum_{i=1}^{D} U^i \Sigma^i (V^i)^* (V^i)((\Sigma^i)^*)(U^i)^* \\ &= \sum_{i=1}^{D} U^i \Sigma^i ((\Sigma^i)^*)(U^i)^* \end{aligned} \tag{3.2}$$

Similarly, the singular values of P are the (non-negative) square root of the eigenvalues of PP^*.

$$PP^* = \sum_{i=1}^{D} U^i \Sigma^i (U^i \Sigma^i)^* = \sum_{i=1}^{D} U^i \Sigma^i ((\Sigma^i)^*)(U^i)^* \tag{3.3}$$

Iwen and Ong proposed the formulas in eq. 3.2 and 3.3 respectively [4]. It was proved that SVD of proxy matrix P must be the same as SVD of the matrix F if and only if each block (F^i) of the matrix F has rank d. The distributed and incremental SVD algorithm proposed by Iwen and Ong [4] was proven based on the equations (3.2) and (3.3) to compute singular values and left singular vectors of the matrix F by computing the singular values and singular left vectors of the proxy matrix P. But

there is a problem such that when the rank (d) of the block matrices of matrix F is smaller than the rank of the matrix F itself, the distributed and incremental algorithm can not compute singular values and singular left vectors with high accuracy (actually computes incorrectly). There might be some rows of some blocks that is completely zero because of the sparsity of F and this leads to the rank of block matrices smaller than d in that case.

3.2 Proposed Methods For Large and Sparse Matrices

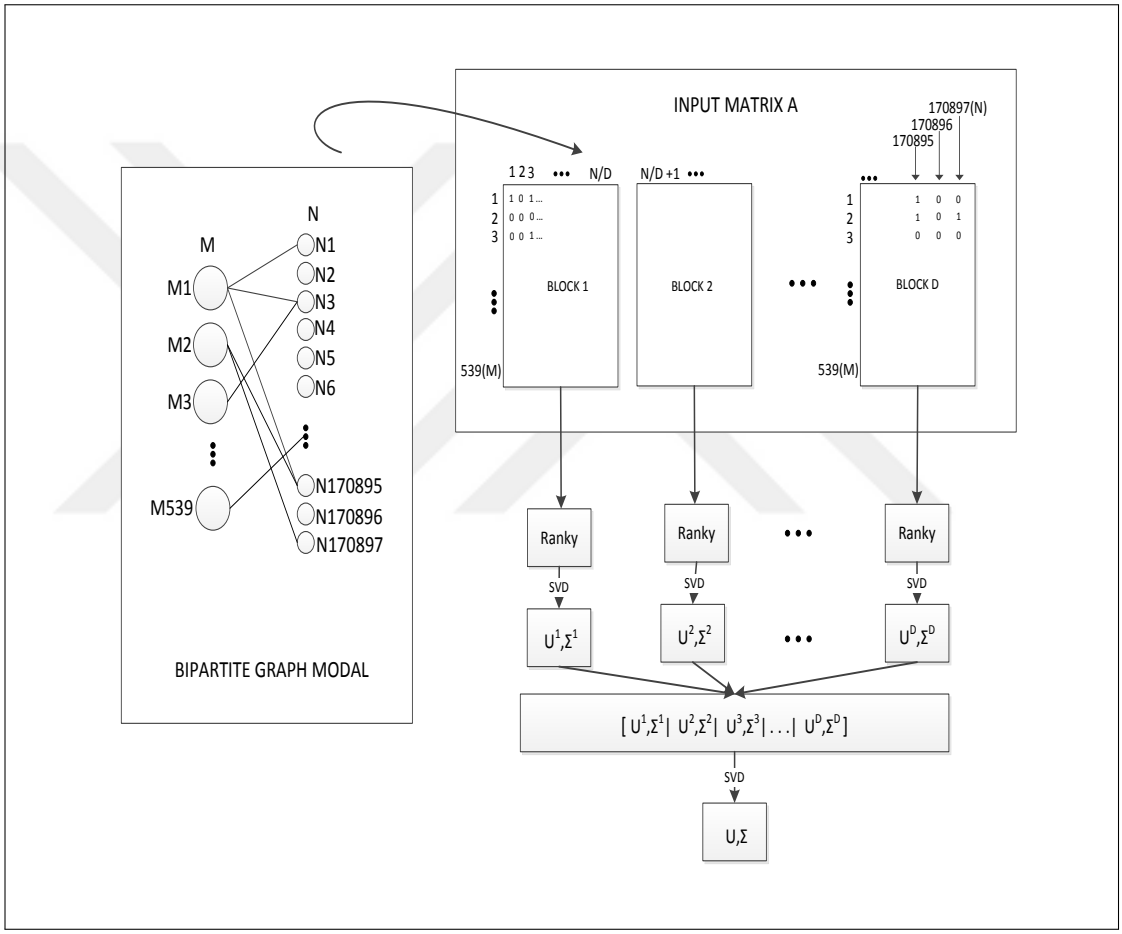

Figure 3.2 : General Schema, One Level Distributed and Incremental SVD with Ranky algorithm.

As aforementioned in the previous section distributed and incremental algorithm can not compute singular left values and singular left vectors correctly even if one of the block of the matrix has smaller rank than the matrix itself. This problem causes inconsistency when computing SVD of the matrix. It is worth to keep in mind that we encounter this problem owing to large sparse matrix. Firstly the bipartite data

is converted to large sparse rectangular matrix F whose rows correspond to X and columns to Y. Then, the matrix is split up to blocks column-wise and a general SVD algorithm is run on each block. After calculating SVD on each block matrix, partial results are combined together to get the SVD of whole matrix F. But there is a problem arising from block matrices. After the representation of bipartite graph as the matrix, the matrix becomes sparse and some nodes will not have any link in some blocks. This problem leads to have less rank of corresponding block than the matrix F. Incremental and distributed SVD algorithm is not able to calculate SVD of the matrix F if the rank is unknown or less for block matrices. We solve this problem using a set algorithms called Ranky. All of the process can be seen in general schema in Figure 3.2.

3.2.1 Ranky algorithm

A set of algorithms, Ranky, namely Random checker, Neighbor checker and Neighbor Random checker. These three algorithms are applied before calculating SVD of each block matrix of the matrix F. Sometimes we will refer to the rows and columns of the matrix F as nodes to be more descriptive. Each row that has no entry or contains zero in a block matrix will be called lonely node, for instance second node is a lonely node as shown in block matrix 1 in Figure 3.2. We give the steps of the Ranky algorithm, then we will explain how other algorithms work with this algorithm. Because other algorithms are just one step of Ranky algorithm. **Ranky**: The steps of Ranky are described as follows:

1. Load matrix F
2. Call one of the Random checker, Neighbor checker or Neighbor Random checker methods.
3. Compute singular values and singular left vectors in parallel for each block matrix.
4. Generate proxy matrix P by getting singular values and singular left vectors from all block matrices.
5. Compute singular values and singular left vectors of proxy matrix P.

Other methods such as Random checker and Neighbor checker can be integrated easily in the second step of the algorithm to solve rank problem of the matrix. As mentioned

earlier, bipartite graph is represented a matrix F having dimension of MxN where M and N represents node in X and Y respectively. After that, the matrix is divided in to D blocks and each block is checked in order to equalize the rank. If there is a lonely node in any block, then other algorithms are being called.

Algorithm 3 Ranky Algorithm

Require: matrix F having dimension MxN
  Split the matrix F into D blocks based on column-wise
  for d = 0 **to** D **do**
    for m = 0 **to** M **do**
      $Checker = true$
      for $n = (N/D) * d$ **to** $(N/D) * (d+1)$ **do**
        if $F_{m,n} \neq 0$ **then**
          $Checker = false$
          break
        end if
      end for
      if $Checker$ **then**
        call one of the Random checker,Neighbor checker or Neighbor Random checker methods
      end if
    end for
  end for
  Compute singular values and singular left vectors in parallel for each block matrix.
  Generate proxy matrix P by getting singular values and singular left vectors from all block matrices.
  Compute singular values and singular left vectors of proxy matrix P.

3.2.2 Random checker method

This method is the simplest method among others. We assume that the number of rows are smaller than the number of columns in each block matrix. It can be inferred that the rank of each block matrix is equal to the rank of matrix F with the approximate probability formula as below:

$$Pr \cong \left(1 - \frac{1}{NC} * NO\right) \tag{3.4}$$

In eq. 3.4, NC represents the number of columns in the block matrix and NO represents the number of rows which have only one column filled and others are zero. Assume that we have the following block matrix F^i having dimension of 5 x 500 and only the

last row has no entry in any column.

$$F^i = \begin{bmatrix} 0 & 1 & 0 & 1 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{3.5}$$

The second, third and forth rows have only one entry in the second, first and forth column respectively in the block matrix shown in equation 3.5. Hence number of columns for this block matrix is $NC = 500$ and number of rows which have only one entry is $NO = 3$. If the last row is filled randomly with Random checker method, the approximate probability of having same rank with matrix F in terms of row-wise will be as follows:

$$Pr \cong \left(1 - \frac{1}{500} * 3\right) = 0.994 = \%99.4 \tag{3.6}$$

As shown in Equation 3.6, the approximate probability that the block matrix has the same rank as the matrix F is %99.4 when using Random checker method in the previous example. If the number of columns gets bigger, then the approximate probability will be higher. Although the Random checker method has a high approximate probability, it does not consider the neighborhood information of nodes in the graph. The method is as follows: We use the term "approximate" with the

Algorithm 4 Random checker method

col = find a random column in block d
$F_{m,col} = 1$

probability, because there is no way to calculate certain probability of matrix having rank d as far as we know. We propose another method called Neighbor checker that is checking neighbors of each node, then adding an edge.

3.2.3 Neighbor checker method

The second method is the neighbor checker method. This method is adding an edge to the lonely node by checking the neighbors of the lonely nodes in other blocks rather than adding randomly. The steps of this method are as follows :

- Check each node of each block matrix, if there is a lonely node, then continue, otherwise go to the next step

- Check other block matrices to determine neighbors of the corresponding node.
- Add the candidate neighbors found in other block matrices to NeighborCandidateList
- Find all nonzero columns of the nodes in the NeighborCandidateList and add these columns to NeighborList
- Select a column from the NeighborList list and add an edge between the lonely node and the column in the corresponding block matrix.

Algorithm 5 Neighbor checker method

```
create an empty list NeighborCandidateList
for d1 = 0 to D do
  if d1 == d then
    continue
  end if
  for n1 = (N/D) * d1 to (N/D) * (d1 + 1) do
    if F_{m,n1} ≠ 0 then
      for m1 = 0 to in M do
        if F_{m1,n1} ≠ 0 then
          add m1 to NeighborCandidateList
        end if
      end for
    end if
  end for
end for
create an empty list NeighborList
for m2 = 0 to size(NeighborCandidateList) do
  for n2 = 0 to in (N/D) * (d + 1) do
    if F_{m2,n2} ≠ 0 then
      add n2 to NeighborList
    end if
  end for
end for
col = choose a random column from NeighborList
F_{i,col} = 1
```

Each block matrix is controlled by Neighbor checker method as shown in Figure 3.2. For instance, there are edges between M1 and N1, N3 and N170895. However, there is no edge between M2 and others in the first block matrix. But there are edges between M2 and N170895,N170897 in the last block matrix. Neighbor checker is checking the first block matrix and identifies that the second row (M2) is fully zero, meaning that

there is no edge between M2 and any other nodes in the first block matrix. Then other blocks rather that the first one are being checked one by one to determine neighbors of M2 and if there is a neighbor, it is added to the NeighborList. In our example, M1 is one of the neighbors of M2 because of the neighborhood of N170895. Then a common edge between M1 and M2 is put to the second row of the first block matrix to get the rank of the this block matrix as same as the rank of the matrix F. But this method has some disadvantages when adding an edge to lonely node. For instance, if there is only one neighbor which has only one column filled (entry) of lonely node, choosing that column causes the rank less than d. Even if there are more than one neighbor, choosing a node which has one column filled randomly causes same problem. Therefore, we use another method, Neighbor Random checker method.

3.2.4 Neighbor random checker method

The rank of the block matrix might be less than the matrix F because of two rows having the same entries in same column. Neighbor checker method does not only consider this problem when adding a new edge to a lonely node also takes the advantage of Random checker method. Thus, we use Random checker and Neighbor checker algorithms together to overcome this issue.

Neighbor Random checker: We use this method to increase the probability that the block matrix and matrix F have the same rank by taking advantage of Neighbor checker and Random checker methods in step 3 instead of Random checker.

- Check each row of each block matrix, if there is a row which is completely zero in a block matrix, then continue, otherwise go to the next step
- Check other block matrices to determine Neighbors of the corresponding node.
- Add the candidate Neighbors found in other block matrices to NeighborCandidateList
- Find all nonzero columns of the nodes in the NeighborCandidateList and add these columns to NeighborList
- if NeighborList is empty, then add an edge to a column of corresponding lonely node randomly.

- if NeighborList contains 1 element, find a common edge between the lonely node and the the column in the NeighborList and add this edge to the corresponding block matrix.
- if NeighborList contains more than one element, then select a node from the NeighborList list and find a common edge between the lonely node and the selected column and add this edge to the corresponding block matrix.
- Add an edge to a column of corresponding lonely node randomly.

Algorithm 6 Neighbor Random checker Method

Firstly call Neighbor checker method
Then call Random checker method

It is relatively simple that it call firstly Neighbor checker then Random checker algorithm.

4. PARTITIONING ALGORITHMS

Graph dataset has very large size with more than billions of vertices and trillions of edges nowadays. Data scientists need a powerful tool, framework or a programming paradigm to analyze these huge datasets. There are many directly graph related frameworks in literature such as Pregel [23] or its open source alternative Giraph [24], GraphLab [25], GRAPE [26] etc. All of these frameworks use either Bulk Synchronous Parallel model or Map Reduce parallel model. In these frameworks, the first step before analyzing the data is graph partitioning. Because graph algorithms that work on huge graph dataset require efficient partitioning strategy to minimize communication among machines and maintaining the load balance. Graph partitioning is an NP-hard problem and it has been studied for decades. There two types of partitioning: online and offline graph partitioning in literature. Offline graph partitioning algorithms such as METIS and spectral clustering needs to know whole graph information to perform offline partitioning. But high computation and memory consumption occur due to the size of the graph. On the other hand, online graph partitioning does not require full graph information. There are two main types of online partitioning either edge-cut or vertex-cut. Edge-cut divides the graph into subgraphs by cutting edges among them, whereas vertex-cut partitions vertices among subgraphs. There is a criteria called as replication factor that measures vertex-cut algorithm in terms of number of replicated vertices. It can be measured as follows:

$$Replicationfactor = \frac{\text{\#replicated vertices + \# of all vertices}}{\text{\# of all vertices}} \tag{4.1}$$

Graph partitioning achieves the best score when the replication factor is 1, because it shows there is no replicated vertices and graph has been partitioned without replication. In this chapter, we will discuss different vertex-cut online graph partitioning algorithms starting from the simple one, hash based or random vertex-cut graph partitioning to complex algorithms such as Bicut, Aweto etc.

4.1 Hash Based

Hash based graph partitioning is the simplest partitioning algorithm among others. Firstly, it defines a hash function and distribute vertices by using this function. For instance, assume that there are 3 workers to be distributed the graph, hash function can be as follows:

$$Hash(x) = x\%3 \tag{4.2}$$

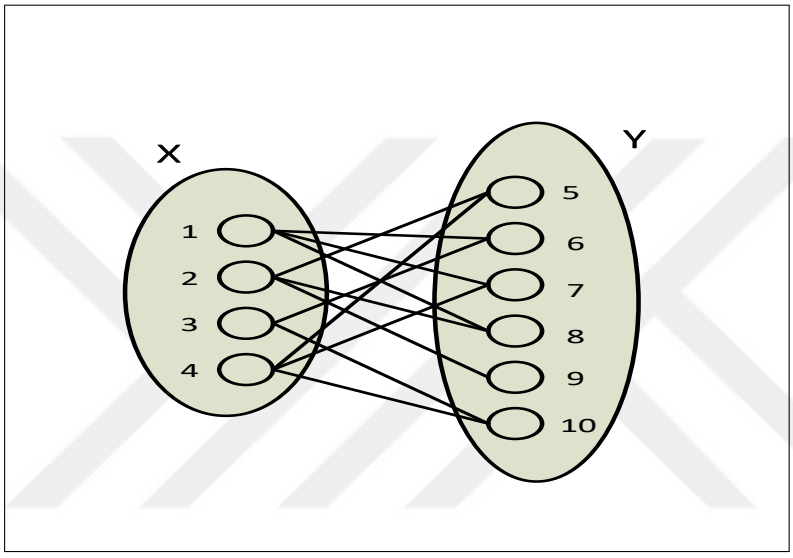

Figure 4.1 : Graph Partitioning General Example.

In Equation 4.2 x is the id of the vertex. There is no need to use all vertices for hash function, only one side of the bipartite graph either X or Y is sufficient. Assume that we have a bipartite graph shown in Figure 4.1. This partitioning strategy is using hash function as in Equation 4.2. For example, we use only the right side (Y) of the bipartite graph. The algorithm starts with the first vertex of the Y side which is 5. Hash function is calculated as $F(5) = 5\%3$ and $F(5) = 2$, then the vertex number 5 will be assigned to 2.worker with its all corresponding edges and vertices which are 1 and 4. This result can be seen in Figure 4.2. Replication factor for this small graph is 1.5 . Number of replicated vertices are 1, 2, 3 and 4 and vertex 1 is replicated twice. Thus replication factor can be calculated as follows:

$$RF = \frac{(1*2+1+1+1)+10}{10} = \frac{15}{10} = 1.5$$

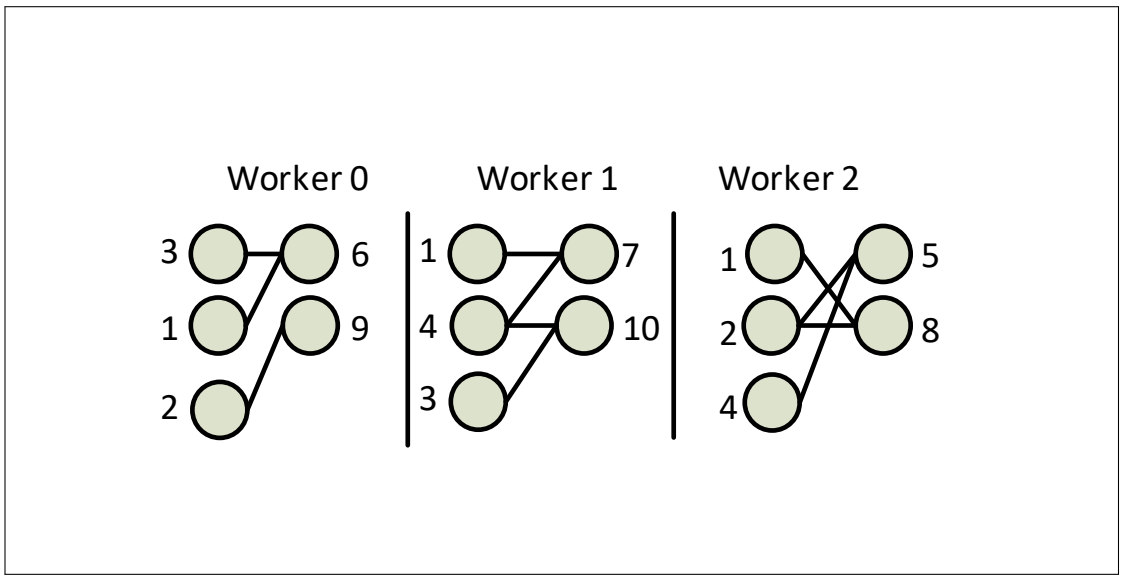

Figure 4.2 : Hash Based Partitioning.

In this equation total number of vertices is 10, but after partitioning we have 15 vertices in total, we can calculate the replication factor by dividing total number of vertices after partitioning by total number of vertices before partitioning directly. Hash-based partitioning does not consider neighbors and distribute randomly vertices among workers. This may cause high replication factor. However, other algorithms such as Grid, Bifennel, Aweto do.

4.2 Grid

4.2.1 Grid-based random vertex-cut

Grid-based partitioning has two different approaches such as grid-based random or grid-based greedy vertex-cut. We will first discuss the grid-based random vertex cut. This approach places the workers into a grid-shaped matrix, then assigns vertices to workers with some constraints. Let u and v be two vertices which have an edge between them and S_i and S_j be the constrained set of u and v respectively. In order to achieve a good partitioning, following conditions must be held.

1. Constrained sets must intersect with $S_i \cap S_j \neq \emptyset$
2. Two constrained sets must have same size $|S_i| = |S_j|$
3. One constrained set does not be a superset of another. $S_i \not\subset S_j$

The question is how to find the constrained sets that hold the requirements above. Firstly, vertex v and u having a common edge (u,v) are mapped using a hash function into workers i, j in the grid-shaped matrix G. Then the intersection worker set of S_i and S_j which are of all horizontal and vertical neighbors of vertex u and v are chosen. Lastly, a worker is chosen randomly from the intersection worker set. By doing so, the replication factor is $2\sqrt{|n|}-1$ at most in this approach, where n is the number of workers in the cluster [27]. Assume that we have 9 workers and we create a matrix

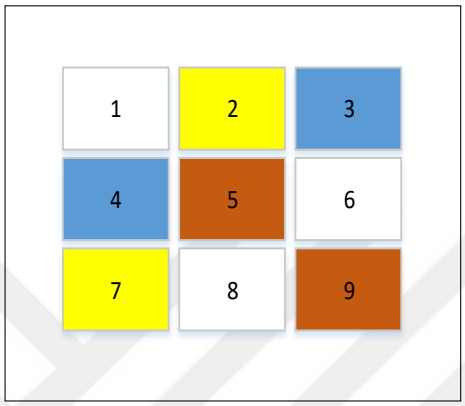

Figure 4.3 : Grid Partitioning Example.

having size 3x3 as shown in Figure 4.3. If a vertex v is mapped to worker 1 according to the hash function, the constrained set of v will be $S_i = \{2,3,4,7\}$ where 2 and 3 are horizontal neighbors of v and 4 and 7 are the vertical neighbors of v. In the same way, if the another vertex u from edge (u,v) is mapped to 6th worker, then the constrained set of u will be $S_j = \{3,4,5,9\}$. The intersection worker set will be the common workers which are 3 and 7 between vertex u and v. Lastly a random worker is chosen from this set to assign vertex u and v.

4.2.2 Grid-based greedy vertex-cut

This approach is used to assign vertices from the intersection worker set in the step of machine selection. Greedy approach is used instead of random approach. For instance, choosing a machine which has less vertices than others is an option. Load balance problem is solved as well as minimizing the replication factor by using this approach.

4.3 BiCut

Bicut bipartite partitioning algorithm is the first algorithm that can be used specially on bipartite graphs. When the bipartite graph has skewed distribution which is one side (X or Y) has more nodes than opposite side (Y or X) of bipartite graph, it is useful to use. The partitioning is done using the side that has more vertices [28]. This side is called as favorite subset. This algorithm has also greedy and random approaches.

4.3.1 Randomized bipartite-cut

By default, favorite subset is used to assign vertices to machines. This approach is quite straightforward, it takes the vertices from the favorite subset and assigns vertices with all edges to machines using a hash function. For example, in Figure 4.1 there are only 4 vertices in X and 6 vertices in Y. Y is chosen as a favorite subset because of its has more number of vertex. Assume that our hash function is $h(x) = x\%3$, then vertices 6 and 9 will be assigned to worker 0, vertices numbered 7 and 10 will be assigned to worker 1 and others to worker 2.

4.3.2 Greedy bipartite-cut (Aweto)

Randomized Bipartite-cut algorithm does not provide a good balance because of hash function. There is no any mechanism to adjustment the balance of the workers, or neighborhood of the vertices. Aweto takes the advantage of checking similarity of neighbors between vertices in the favorite subset as well as balancing load of machines. Aweto takes one more additional round of edge exchange to use the similarity of neighbors among vertices in the favorite subset and the balance of edges in partitions. The first step is always same that is using a hash function and randomly assignment. After this step, Aweto re-assign all vertices in the favorite subset using the formulas below:

- re-assign a vertex to partition i if $\delta_g(v,S_i) \geq \delta_g(v,S_j)\ for j \in (1,2,..p)$
- S_i is the current vertex set of the vertex

- $\delta_g(v,S_i) = |N(v) \cap S_i| - B(|S_i|)$ where $B(x) = |S_i|^{1/2}B(x)$ and $N(v)$ is the set of neighbors of vertex v

In this thesis we also use this algorithm in addition to Bifennel graph partitioning algorithm which is based on this algorithm. Because of the similarity of the algorithm Aweto and Bifennel, we will give an example of Bifennel algorithm next.

4.4 Bifennel

Bifennel is based on the Fennel [29] algorithm which is an vertex cut graph partitioning algorithm. Unlike Bicut and Aweto algorithms, This algorithm is trying to balance of size of partitions and checking the similarities between vertices. Steps of this algorithm are similar to the steps of the Aweto algorithm. This algorithm is also working with the favorite subset. The steps can be defined as follows:

1. Create adjacency matrix of favorite subset of bipartite graph.
2. Use the formula $\delta_g(v,S_i) = |N(v) \cap S_i| - \sqrt[2]{|S_i| + |N(v)|}$ to assign vertex v in the favorite subset to partition i where $\delta_g(v,S_i) \geq \delta_g(v,S_j)$ for all $j \in j = \{1,2,3,..k\}$

The first part of the formula is same with Aweto algorithm, however, second part which is handling edge balance of partitions is different. Results that are reported in [29] show that Bifennel algorithm has better performance than Aweto and Bicut in terms of the replication factor and computation cost. Also runtime of Bifennel has almost doubled the runtime of Aweto and Bicut. The algorithm is as follows:

4.5 Distributed Bifennel

Bifennel partitioning is a sequential bipartite partitioning algorithm which means there is no any parallel execution. However this algorithm can be implemented as a parallel algorithm by applying some differences on it. This algorithm is working by taking a vertex from favorite set , assigns it to a partition and skips to another vertex in the set. Partitioning results are changing when the order of vertices are changed. If we start with a random vertex from the favorite set, the partitioning result would change, but replication factor would become fairly constant.

Algorithm 7 Bifennel Algorithm

Require: Create a k–bit bitmap to save vertices' location in U, named as *neighbor_map*. Save metadata into adjacent list, named as *adj_map*

```
if the graph is finalized then
    return
end if
proc_num_edge[k] ⇐ 0
for each v in V M do
    create a k-element vector proc_degree to keep |N(v) ∩ S_i|
    best_proc ← 1
    best_score ← INT_MIN
    for each v' ∈ |N(v)| do
        for i = 1 to k do
            proc_degree[i] ← neighbor_map[v'][i] //compute |N(v) ∩ S_i|
        end for
    end for
    for i = 1 to k do
        proc_score ← proc_degree[i] - ∛(proc_num_edge[i] + |N(v)|)
        if proc_score > best_score then
            best_score ← proc_score
            best_proc ⇐ i
        end if
    end for
    for each v' ∈ |N(v)| do
        neighbor_map[v'][best_proc] ← 1
        send(best_proc, edge(v, v')) //send all edges to v's edges to appointed machine
    end for
    proc_num_edge[k] ← proc_num_edge[k] + |N(v)|
end for
delete neighbor_map, adj_map. //save memory for local graph
```

The formula that decides which vertex must be assigned to which partition can be calculated separately. S_i represents in this equation $\delta_g(v,S_i) = |N(v) \cap S_i| - \sqrt[2]{|S_i| + |N(v)|}$ the edge number of partition i. The number of neighbors of vertex v $|N(v)|$ does not change, thus we can calculate it independently for each node in the favorite set. Keeping the number of edge of partitions global helps us to assign vertices balanced. In this thesis, we used a threaded version of Bifennel algorithm named Distributed Bifennel in order to get equally likely balanced partitions in addition to less replication factor. Experimental results show that the replication factor of Distributed Bifennel algorithm is almost the replication factor of Bifennel. The advantage of Distributed Bifennel algorithm does not only have approximate replication factor with Bifennel algorithm also has fast execution time.

5. EXPERIMENTAL RESULTS

In this chapter, we will give results of Ranky algorithm in detail and discuss results of Bifennel and Distributed Bifennel algorithm. Lasly, clustering results will be given in terms of edge cut ratio.

5.1 Ranky Algorithm Results

Data used in the experiments is obtained from kariyer.net, one of the most popular job-site companies in Turkey. Data can be defined with a bipartite graph that has 539 jobs and 170,897 candidates with 269,650 edges representing job applications. This bipartite graph is converted into a job-candidate adjacency matrix whose rows represent jobs and columns represent candidates.

LAPACK SVD algorithm, dgesvd function located in threaded Intel MKL library is used to find SVD of each block matrix. The code was written in C++ and run on 8 cores and 10 GB RAM machine running Linux operating system. This algorithm currently runs on one machine but can run on distributed machines in a cluster and transfer data between the machines via sockets. Execution time of these results is not reported here as these are ultimately dependent on the number of processors and number of machines used in a distributed situation. Criteria is sum of total error which is an evaluation metric between the true singular values (σ_i) and obtained singular values (($\hat{\sigma}_i$)). Similarly, we used the same evaluation metric for the singular left vectors between true (e_i) and obtained ($\hat{e_{ui}}$) as shown below. We compare there different algorithms the Random checker, Neighbor checker and Neighbor Random checker.

$$e_\sigma = \sum_{i=1}^{N} |\hat{\sigma}_i - \sigma_i| \quad \text{and} \quad e_u = \sum_{i=1}^{N} |\hat{e}_{ui} - e_i|$$

The sum of total error for singular values and singular left vectors is shown in Table 5.1 and Table 5.2 with Random Checker and Neighbor Checker methods respectively as rank controller. Errors are negligible as shown in Table 5.1 and Table 5.2. But, it is an undeniable fact that there is no relationship between the errors and number of blocks

due to randomness. Although using more number of blocks is an advantage as speed of execution time, it might cause a problem in terms of rank of the block matrices. Because having more block matrices leads to have less rank of the block matrices. Hence the Random Checker method might change the structure of adjacency matrix in case of using more blocks because of its nature. Other method, Neighbor checker, is using the similarity of neighborhood of nodes when solving the rank problem. Even distributing more block matrices does not change the structure of adjacency matrix. On the other hand, this method can not find singular values and singular left vectors with negligible error as much as Random Checker method does. Because lonely nodes that the same of the other nodes cause the less rank of the block matrices than the rank of adjacency matrix.

Table 5.1 : Random Checker.

# Blocks	Block Size	e_σ	e_u
2	539 x 85448	$2.502443e-13$	$4.052329e-10$
3	539 x 56965	$2.067235e-13$	$3.030222e-10$
4	539 x 42724	$3.258505e-14$	$6.044171e-10$
8	539 x 21362	$4.130030e-14$	$1.867252e-10$
10	539 x 17089	$4.263256e-13$	$4.604847e-10$
16	539 x 10681	$4.501954e-14$	$6.100364e-10$
32	539 x 5340	$2.554623e-13$	$9.281878e-10$
64	539 x 2670	$8.620882e-14$	$3.095248e-10$
128	539 x 1335	$3.600453e-13$	$1.665984e-10$

For all these reason mentioned, we use Random Checker and Neighbor Checker methods together as neighbor Random Checker to solve this problem.

Table 5.3 shows the results of neighbor Random Checker method. Similar results we obtained with the Random Checker method. Calculation of singular values and left singular vectors in terms of sum of total error when using three different methods are same as shown in Table 5.1 and 5.3. However, it might be useful to use neighbor Random Checker method especially in solving clustering problems. Because the purpose of graph partitioning approaches is finding similar and coherent groups of nodes.

Table 5.2 : Neighbor Checker.

# Blocks	Block Size	e_σ	e_u
2	539 x 85448	$2.522729e-14$	$1.502954e-01$
3	539 x 56965	$4.903300e-13$	$1.069363e-02$
4	539 x 42724	$2.416956e-13$	$4.489185e-01$
8	539 x 21362	$3.885781e-14$	$4.402455e-01$
10	539 x 17089	$3.480549e-13$	$5.048745e-01$
16	539 x 10681	$2.601808e-13$	$2.820104e-02$
32	539 x 5340	$3.574918e-14$	$6.011384e-01$
64	539 x 2670	$2.621237e-13$	$4.517198e-01$
128	539 x 1335	$1.404987e-13$	$7.113150e-10$

Table 5.3 : Neighbor Random Checker.

# Blocks	Block Size	e_σ	e_u
2	539 x 85448	$2.298162e-14$	$6.175930e-10$
3	539 x 56965	$1.432188e-13$	$7.913495e-10$
4	539 x 42724	$2.468581e-13$	$6.211098e-10$
8	539 x 21362	$2.033373e-13$	$8.652412e-11$
10	539 x 17089	$1.565414e-14$	$1.504255e-10$
16	539 x 10681	$9.953149e-14$	$1.138005e-10$
32	539 x 5340	$2.702838e-13$	$4.859414e-10$
64	539 x 2670	$1.625922e-13$	$1.827257e-10$
128	539 x 1335	$1.404987e-13$	$7.113150e-10$

5.1.1 Partitioning algorithms results

Bifennel bipartite partitioning algorithm has the minimum replication factor compared with other bipartite partitioning algorithms according to the result reported in [1]. In this section we only compare the results of Distributed Bifennel algorithm and Bifennel algorithm in terms of replication factor and execution times. All experiments for bipartite graph partitioning algorithms were performed on two different datasets. The first one contains 48,138 jobs in X and 195,845 candidates in Y with 1,934,005 edges and the second one has 95,973 jobs in X and 941,604 candidates in Y with 34,781,273 edges. First dataset, called DS1, is run on a machine that has 8 cores and 10 GB RAM

running Linux operating system due to its small size and the latter larger one, called DS2, is run on 28 cores and 128 GB RAM machine running Linux operating system.

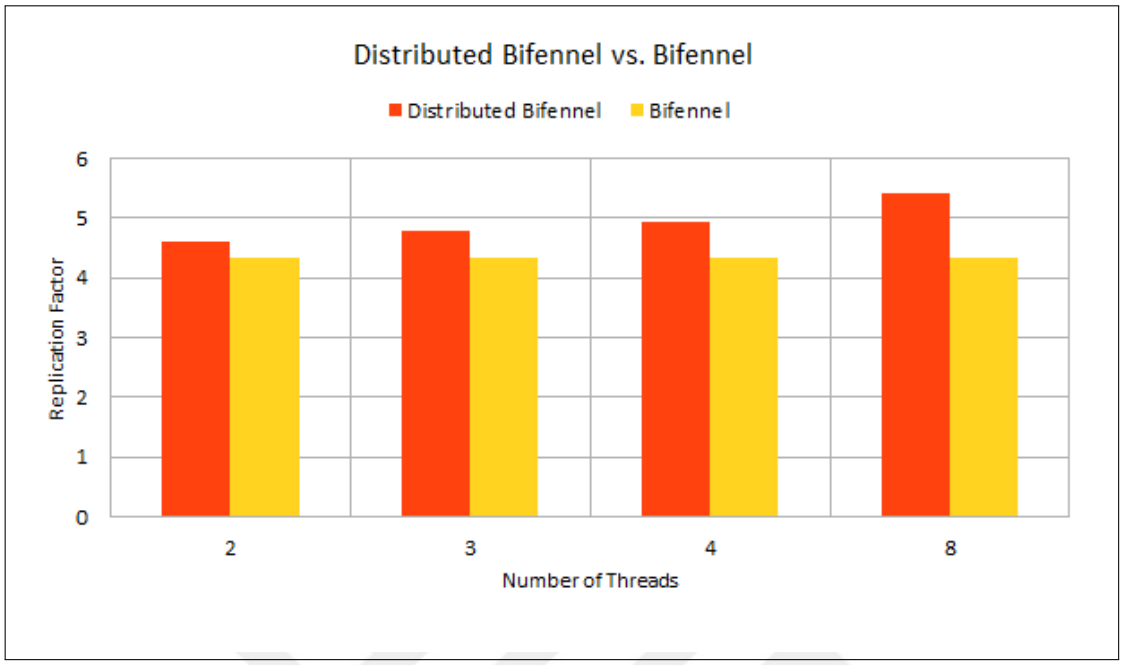

Figure 5.1 : Replication Factor vs. Number of threads (1000 partitions for DS1).

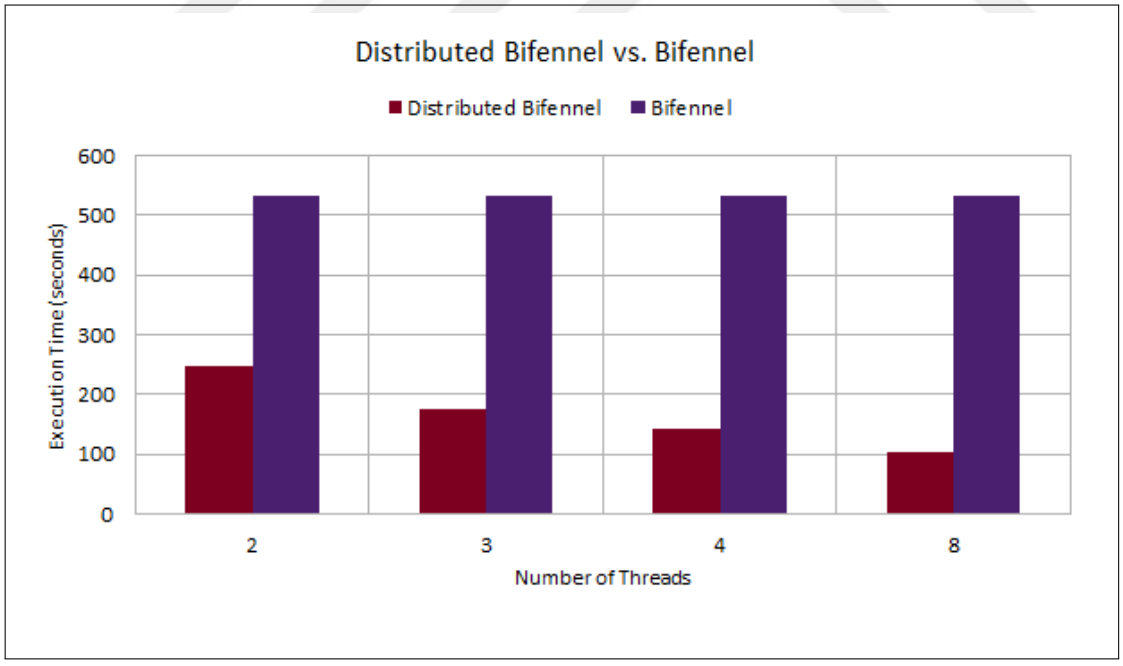

Figure 5.2 : Execution Time vs. Number of threads (1000 partitions for DS1).

The Figure 5.1 and Figure 5.3 clearly show that the replication factor is approximately equal for Distributed Bifennel and Bifennel algorithm. The replication factor is at most

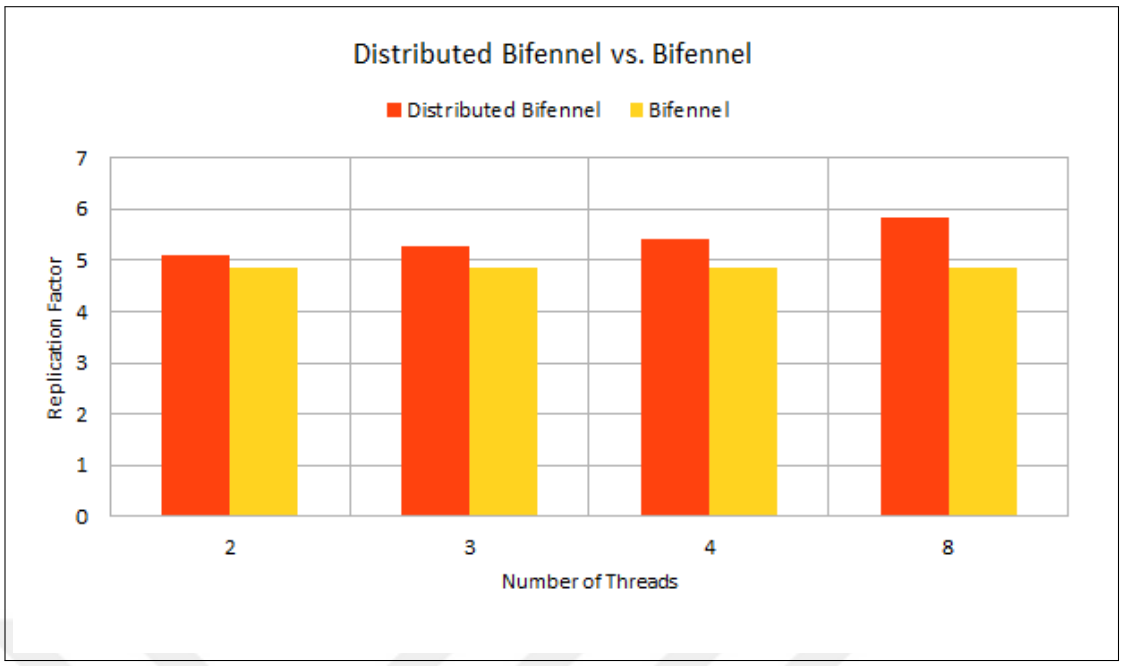

Figure 5.3 : Replication Factor vs. Number of threads (500 partitions for DS1).

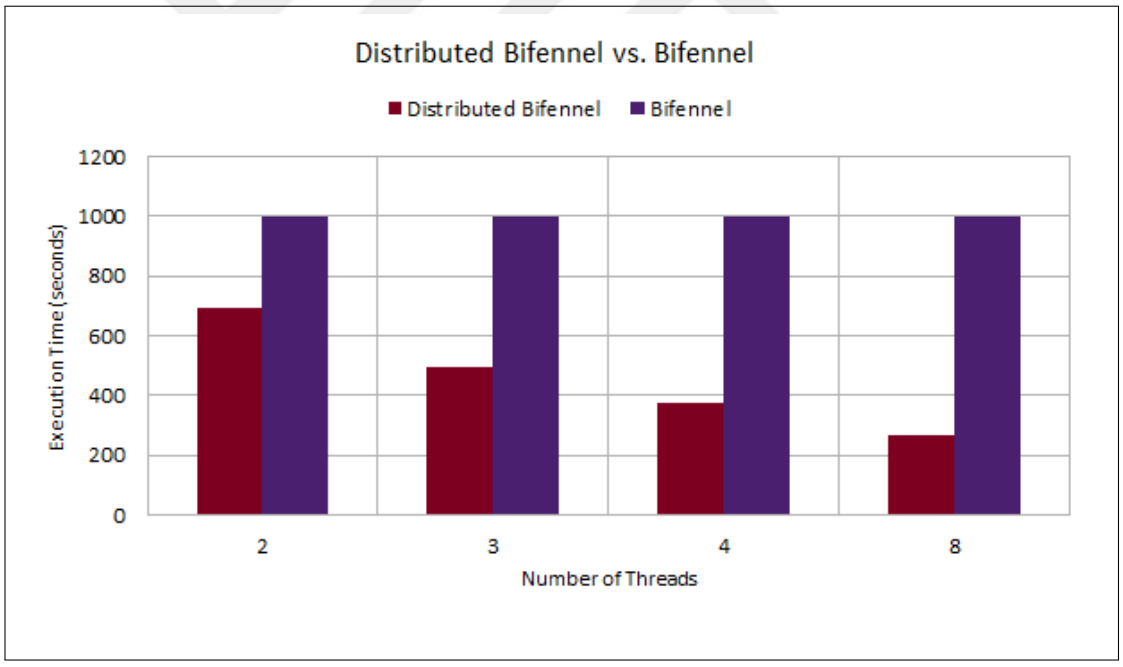

Figure 5.4 : Execution Time vs. Number of threads (500 partitions for DS1).

5.84 with 8 threads and 5.1 with only 2 threads for partitioning 195,845 jobs in X and 48,135 candidates in Y into 1000 partition using Distributed Bifennel. This number is 4.85 for Bifennel algorithm. On the other hand, the execution time of Distributed Bifennel algorithm with 8 threads is almost 4 times faster than Bifennel algorithm as

shown in Figure 5.2 and 5.4. Even Distributed Bifennel algoritm is 1.44 times faster than Bifennel when using only 2 threads.

Nodes in X set have been partitioned into 1000 partitions using distributed Bifennel algorithm only in 608 seconds and 10.48 replication factor with 28 threads. This results have declined to 442 seconds and 8.42 replication factor for only 500 partitions under same conditions. But Bifennel algorithm has partitioned the same data in 6463 seconds with 7.3 replication factor and 3157 seconds with 5.82 replication factor for 1000 and 500 partitions respectively. Results show that Distributed Bifennel algorithm has partitioned the data 10 times faster than Bifennel using at most 28 threads as shown in Figure 5.7 and 5.8. On the other hand, replication factor has increased by 1.4 times when using Distributed Bifennel algorithm as shown in Figure 5.5 and 5.6.

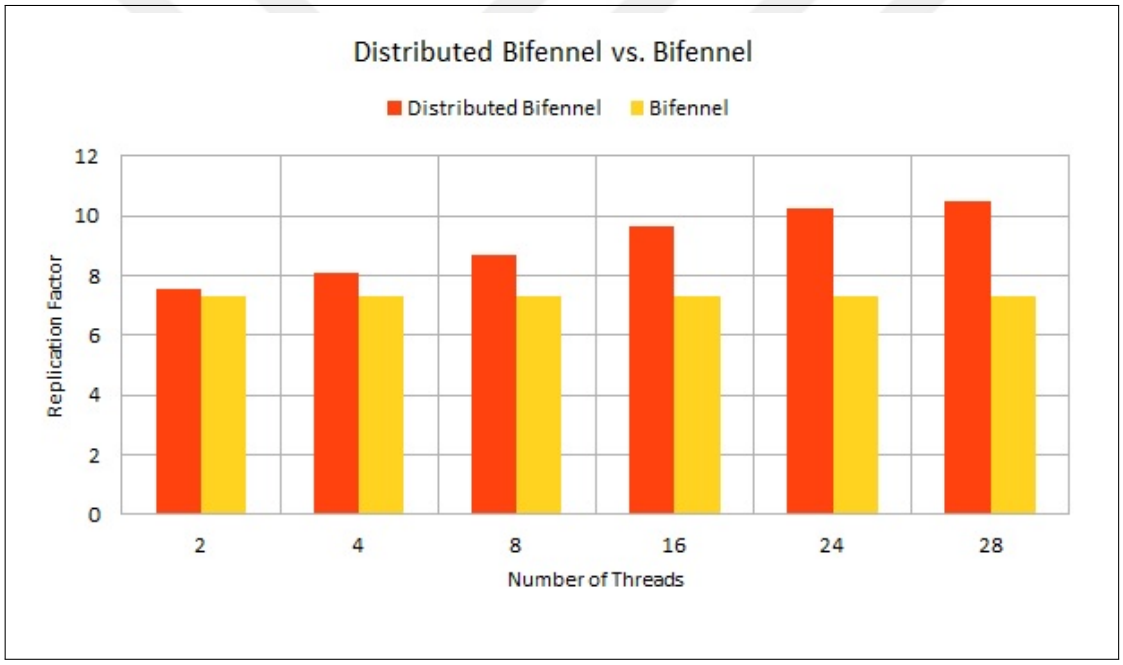

Figure 5.5 : Replication Factor vs. Number of threads (1000 partitions for DS2).

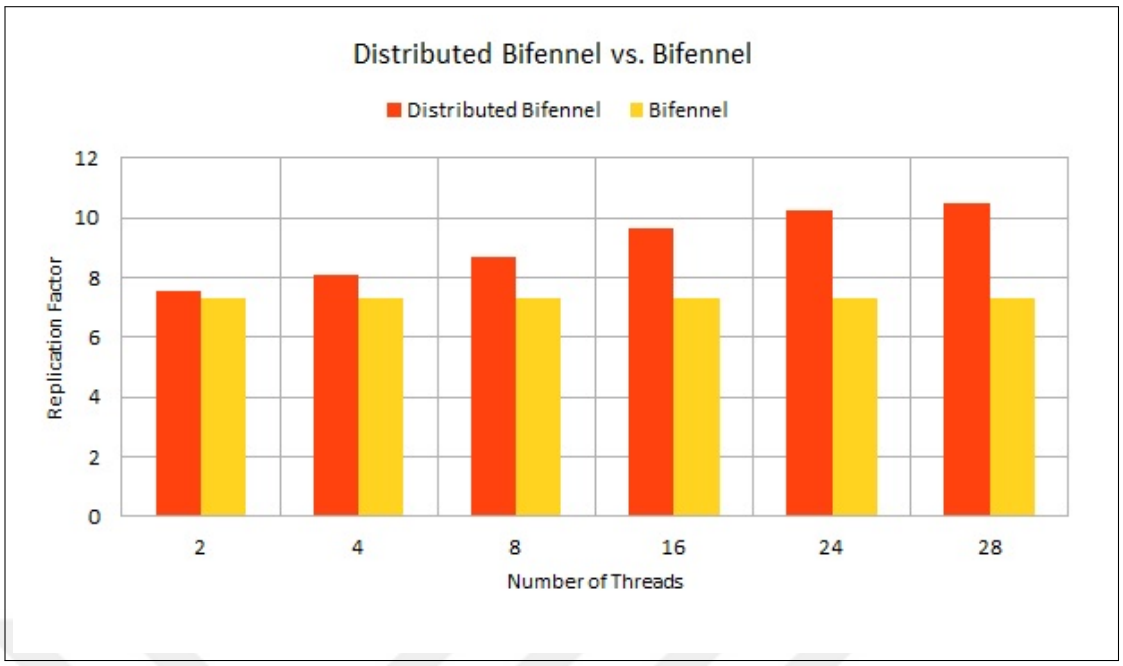

Figure 5.6 : Replication Factor vs. Number of threads (500 partitions for DS2).

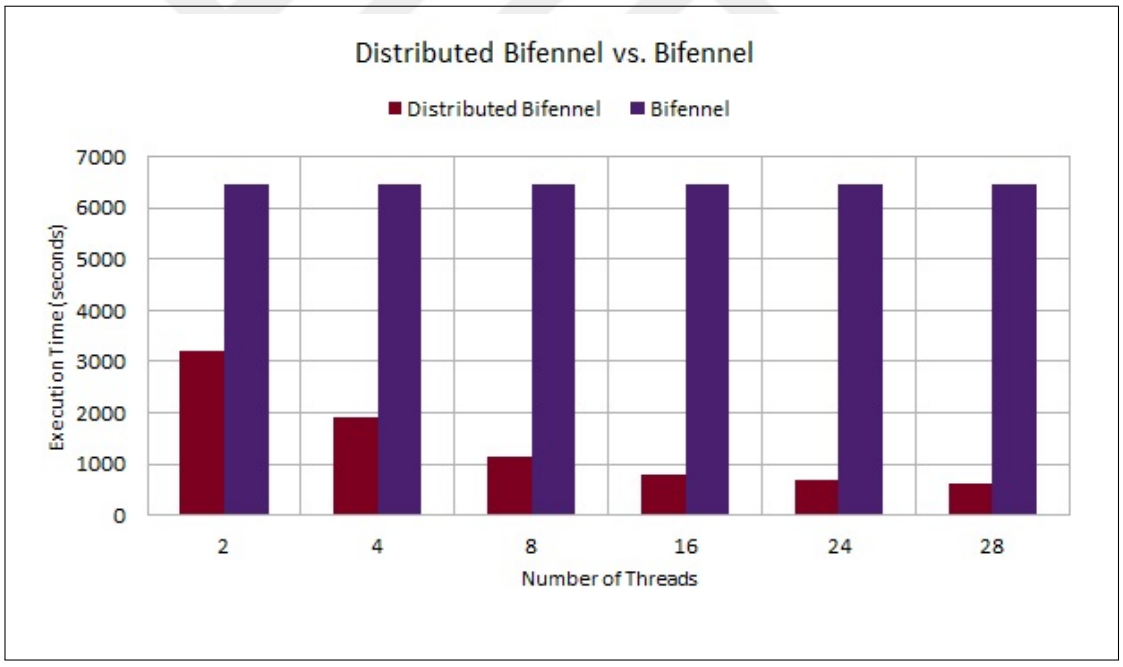

Figure 5.7 : Execution Time vs. Number of threads (1000 partitions for DS2).

5.1.2 Clustering results

Although the most important evaluation metric is execution times, there are different metrics which can measure the quality of clustering / partitioning. Edge cut ratio can

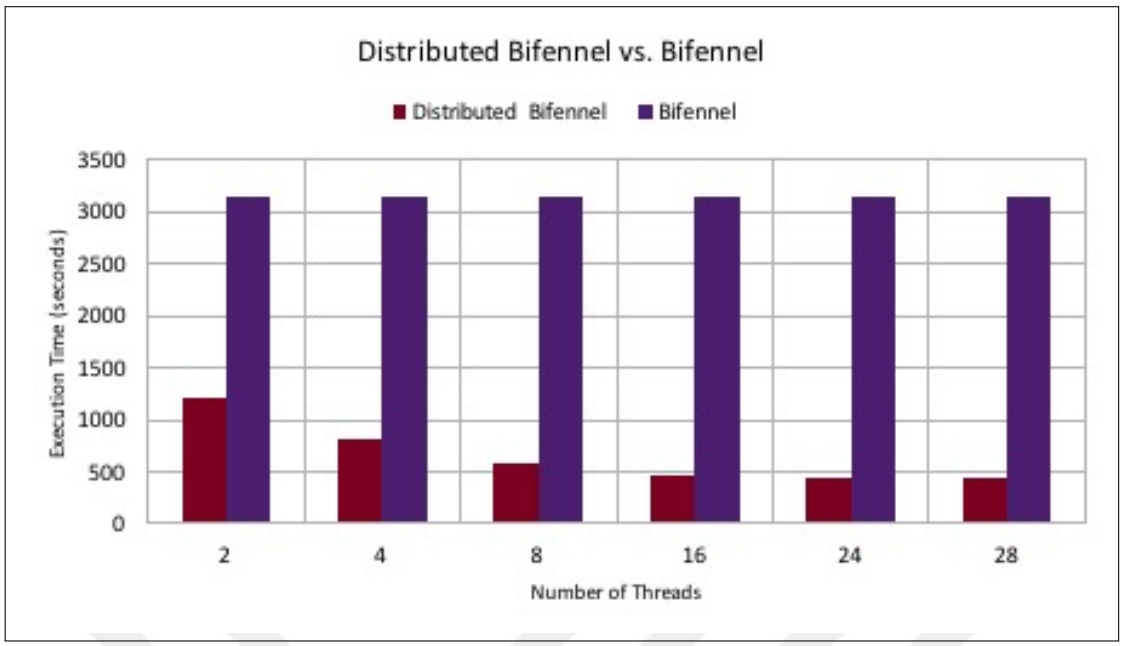

Figure 5.8 : Execution Time vs. Number of threads (500 partitions for DS2).

be used to measure the quality of clustering and it is calculated using formula shown in Equation 5.1.

$$\text{Edge Cut Ratio} = \frac{\text{Number of Edges Cut}}{\text{Number of Edges}} \tag{5.1}$$

For example, edge cut ratio for the graph shown in Figure 5.9 is $4/8 = 0.5$ where 4 is the number of edge cuts and 8 is the all number of edges.

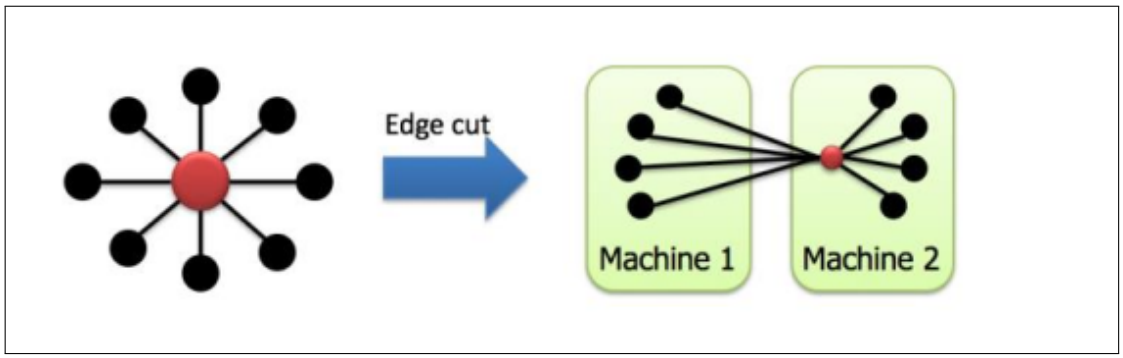

Figure 5.9 : Edge Cut Example.

It is an undeniable fact that the edge cut ratio is small if the number of cluster is small. On the other hand, it is going up with the increasing number of cluster. Edge cut ratio is only around 0.2 with 2 clusters and around 0.8 with only 10 clusters for the dataset

DS1. While it has changed sharply between the cluster number 3 and 6, it has not changed that much between cluster 7 and 10.

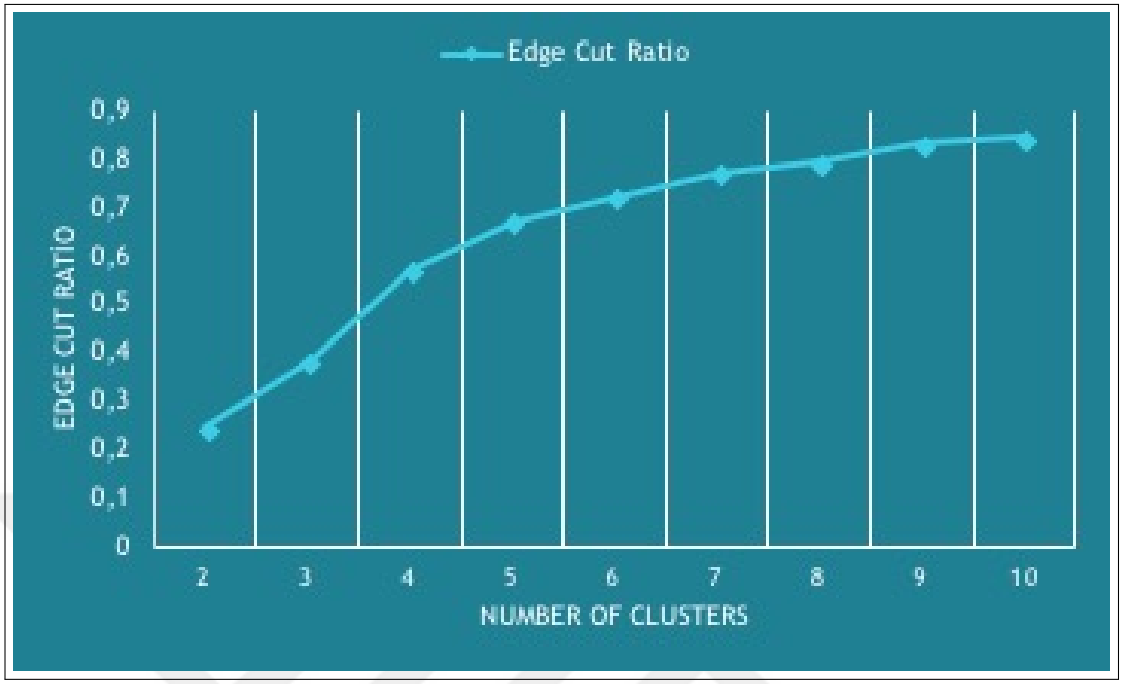

Figure 5.10 : Edge Cut.

6. CONCLUSIONS AND DISCUSSIONS

In this thesis, we proposed new methods to cluster large sparse bipartite data. Firstly, we converted the data into adjacency matrix and then attempted to cluster the matrix by using spectral clustering along with incremental and hierarchical distributed SVD algorithm. But we encountered a rank problem of the matrix. The rank problem has arising from dividing the matrix into sub matrices with column-wise manner. The incremental and hierarchical distributed SVD algorithm works only if the rank of the sub matrices(a.k.a block matrices) is equal to the rank of the adjacency matrix. The rank of the matrix becomes greater than the rank of sub matrices due to sparsity of the matrix.

We propose a set of methods, namely Random checker, Neighbor checker and Neighbor Random checker, called Ranky algorithms, to solve SVD of a large and sparse rectangular matrices in a distributed manner. Ranky is inspired by incremental and hierarchical distributed SVD algorithm. The experimental results proved that Ranky algorithms recover singular values and left singular vectors of large-sparse adjacency matrices with negligible error. Random checker applies random strategy to handle rank problem of sparse adjacency matrix and Neighbor checker uses neighbors of the nodes. Neighbor Random checker solves this problem by taking advantage of both methods.

Although this methods solve the rank problem, clustering could not be solved due to sparsity. After that we applied another algorithms called Bifennel bipartite partitioning before the incremental and hierarchical SVD algorithm and compressed data. By doing so, the clustering results became sensible and balanced. Then we propose a Distributed Bifennel algorithm to make the application faster. The results showed that the replication factor of Distributed Bifennel algorithm is almost same as the replication factor of Bifennel itself and Execution time is much more smaller than Bifennel. As for future work, some bipartite graph partitioning algorithms can be developed to completely guarantee that replication factor is minimum. What is more,

a framework containing both edge-cut and vertex-cut partitioning strategies can be found to partition sparse bipartite data.

REFERENCES

[1] **Wang, L.W.**, **Chen, S.C.**, **Chen, W.**, **Hsiao, H.C. and Chung, Y.C.** (2015). BiFennel: Fast Bipartite Graph Partitioning Algorithm for Big Data, *Smart City/SocialCom/SustainCom (SmartCity), 2015 IEEE International Conference on*, IEEE, pp.715–720.

[2] **Ding, C.H.**, **He, X.**, **Zha, H.**, **Gu, M. and Simon, H.D.** (2001). A min-max cut algorithm for graph partitioning and data clustering, *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, IEEE, pp.107–114.

[3] **Dhillon, I.S.** (2001). Co-clustering documents and words using bipartite spectral graph partitioning, *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, pp.269–274.

[4] **Iwen, M. and Ong, B.** (2016). A distributed and incremental svd algorithm for agglomerative data analysis on large networks, *SIAM Journal on Matrix Analysis and Applications*, *37*(4), 1699–1718.

[5] **Cline, A.K. and Dhillon, I.S.** (2006). Computation of the singular value decomposition.

[6] **Golub, G. and Kahan, W.** (1965). Calculating the singular values and pseudo-inverse of a matrix, *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, *2*(2), 205–224.

[7] **Vasudevan, V. and Ramakrishna, M.** (2017). A Hierarchical Singular Value Decomposition Algorithm for Low Rank Matrices, *arXiv preprint arXiv:1710.02812*.

[8] **Ravi Kanth, K.**, **Agrawal, D. and Singh, A.** (1998). Dimensionality reduction for similarity searching in dynamic databases, *ACM SIGMOD Record*, volume 27, ACM, pp.166–176.

[9] **Hannachi, A.**, **Jolliffe, I. and Stephenson, D.** (2007). Empirical orthogonal functions and related techniques in atmospheric science: A review, *International journal of climatology*, *27*(9), 1119–1152.

[10] **Wall, M.E.**, **Rechtsteiner, A. and Rocha, L.M.**, (2003). Singular value decomposition and principal component analysis, A practical approach to microarray data analysis, Springer, pp.91–109.

[11] **Moonen, M. and De Moor, B.** (1995). *SVD and Signal Processing, III: Algorithms, Architectures and Applications*, Elsevier.

[12] **Eckart, C. and Young, G.** (1939). A principal axis transformation for non-Hermitian matrices, *Bulletin of the American Mathematical Society*, *45*(2), 118–121.

[13] **Roman, J.E.**, **Campos, C.**, **Romero, E. and Tomás, A.** (2015). SLEPc users manual, *D. Sistemes Informatics i Computació, Universitat Politecnica de Valencia, Tech. Rep. DSIC-II/24/02-Revision*, *3*.

[14] **Golub, G.H.**, **Luk, F.T. and Overton, M.L.** (1981). A block Lanczos method for computing the singular values and corresponding singular vectors of a matrix, *ACM Transactions on Mathematical Software (TOMS)*, *7*(2), 149–169.

[15] **Simon, H.D. and Zha, H.** (2000). Low-rank matrix approximation using the Lanczos bidiagonalization process with applications, *SIAM Journal on Scientific Computing*, *21*(6), 2257–2274.

[16] **Golub, G.H. and Reinsch, C.** (1970). Singular value decomposition and least squares solutions, *Numerische mathematik*, *14*(5), 403–420.

[17] **Gu, M. and Eisenstat, S.C.** (1995). A divide-and-conquer algorithm for the bidiagonal SVD, *SIAM Journal on Matrix Analysis and Applications*, *16*(1), 79–92.

[18] **Qu, Y.**, **Ostrouchov, G.**, **Samatova, N. and Geist, A.** (2002). Principal component analysis for dimension reduction in massive distributed data sets, *Proceedings of IEEE International Conference on Data Mining (ICDM)*.

[19] **Bai, Z.J.**, **Chan, R.H. and Luk, F.T.** (2005). Principal component analysis for distributed data sets with updating, *International Workshop on Advanced Parallel Processing Technologies*, Springer, pp.471–483.

[20] **Baker, C.G.**, **Gallivan, K.A. and Van Dooren, P.** (2012). Low-rank incremental methods for computing dominant singular subspaces, *Linear Algebra and its Applications*, *436*(8), 2866–2888.

[21] **Menon, A.K. and Elkan, C.** (2011). Fast algorithms for approximating the singular value decomposition, *ACM Transactions on Knowledge Discovery from Data (TKDD)*, *5*(2), 13.

[22] **Brand, M.** (2006). Fast low-rank modifications of the thin singular value decomposition, *Linear algebra and its applications*, *415*(1), 20–30.

[23] **Malewicz, G.**, **Austern, M.H.**, **Bik, A.J.**, **Dehnert, J.C.**, **Horn, I.**, **Leiser, N. and Czajkowski, G.** (2010). Pregel: a system for large-scale graph processing, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ACM, pp.135–146.

[24] **Avery, C.** (2011). Giraph: Large-scale graph processing infrastructure on hadoop, *Proceedings of the Hadoop Summit. Santa Clara*, volume 11, pp.5–9.

[25] **Low, Y.**, **Bickson, D.**, **Gonzalez, J.**, **Guestrin, C.**, **Kyrola, A. and Hellerstein, J.M.** (2012). Distributed GraphLab: a framework for machine learning and data mining in the cloud, *Proceedings of the VLDB Endowment*, *5*(8), 716–727.

[26] **Fan, W.**, **Xu, J.**, **Wu, Y.**, **Yu, W. and Jiang, J.** (2017). GRAPE: parallelizing sequential graph computations, *Proceedings of the VLDB Endowment*, *10*(12), 1889–1892.

[27] **Jain, N.**, **Liao, G. and Willke, T.L.** (2013). Graphbuilder: scalable graph etl framework, *First International Workshop on Graph Data Management Experiences and Systems*, ACM, p. 4.

[28] **Chen, R.**, **Shi, J.**, **Zang, B. and Guan, H.** (2014). Bipartite-oriented distributed graph partitioning for big learning, *Proceedings of 5th Asia-Pacific Workshop on Systems*, ACM, p. 14.

[29] **Tsourakakis, C.**, **Gkantsidis, C.**, **Radunovic, B. and Vojnovic, M.** (2014). Fennel: Streaming graph partitioning for massive scale graphs, *Proceedings of the 7th ACM international conference on Web search and data mining*, ACM, pp.333–342.

CURRICULUM VITAE

Resul Tugay:

Place and Date of Birth: Erzurum 1992

E-Mail: resultugay@hotmail.com, tugayr@itu.edu.tr

EDUCATION:

- **B.Sc.:** 2015, Karadeniz Technical University, Electrical and Electronic Engineering, Computer Engineering Department
- **M.Sc.:** 2018, Istanbul Technical University, Faculty of Computer and Informatics, Computer Engineering Department

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2015-2016 Software Developer / Agito Bilgisayar Yazılım ve Danışmanlık Hizmetleri A.Ş.
- 2016-Present Research Assistant / I.T.U Computer Science Department

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

- Tugay R. and Gündüz Öğüdücü Ş. (2017). Demand Prediction using Machine Learning Methods and Stacked Generalization *In Proceedings of the 6th International Conference on Data Science, Technology and Applications - Volume 1: DATA, ISBN 978-989-758-255-4, pages 216-222. DOI: 10.5220/0006431602160222*, July 24-26, 2017 Madrid, Spain.