Using Map-Reduce for Large Scale Analysis of Graph-Based Data

NAN GONG

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2011

TRITA-ICT-EX-2011:218

Using Map-Reduce for Large Scale Analysis of Graph-Based Data

Master of Science Thesis

***Author*:**

Nan Gong

Examiner:

Prof. Vladimir Vlassov

KTH, Stockholm, Sweden

Supervisor:

Dr. Jari Koister

Salesforce.com, San Francisco, USA

Prof. Vladimir Vlassov

KTH, Stockholm, Sweden

Royal Institute of Technology

Kungliga Tekniska högskolan

Stockholm, Sweden.

August, 2011

Abstract

As social networks have gained in popularity, maintaining and processing the social network graph information using graph algorithms has become an essential source for discovering potential features of the graph. The escalating size of the social networks has made it impossible to process the huge graphs on a single machine in a “real-time” level of execution. This thesis is looking into representing and distributing graph-based algorithms using Map-Reduce model.

Graph-based algorithms are discussed in the beginning. Then, several distributed graph computing infrastructures are reviewed, followed by Map-Reduce introduction and some graph computation toolkits based on Map-Reduce model. By reviewing the background and related work, graph-based algorithms are categorized, and adaptation of graph-based algorithms to Map-Reduce model is discussed. Two particular algorithms, MCL and DBSCAN are chosen to be designed using Map-Reduce model, and implemented using Hadoop. New matrix multiplication method is proposed while designing MCL. The DBSCAN is reformulated into connectivity problem using filter method, and Kingdom Expansion Game is proposed to do fast expansion. Scalability and performance of these new designs are evaluated. Conclusion is made according to the literature study, practical design experience and evaluation data. Some suggestions of graph-based algorithms design using Map-Reduce model are also given in the end.

Acknowledgement

Firstly, I'd like to thank my supervisor, Jari Koister of Salesforce.com, for giving me this opportunity to conduct such an interesting research. And also thank him and all of my colleagues in Salesforce for their help and advices on my thesis work.

I would also like to thank my parents and all of my friends who gave unconditional support during my study in Sweden.

Finally, I would like to thank my examiner, Vladimir Vlassov of Royal Institute of Technology, for giving me academic suggestions and reviewing my report to improve the quality of my research.

Table of Contents

Table of Figures

List of Abbreviations

BSP	Bulk Synchronous Parallel
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
EC2	Elastic Compute Cloud
GIM-V	Generalized Iterated Matrix-Vector
HDFS	Hadoop Distributed File System
I/O	Input/Output
KEG	Kingdom Expansion Game
MCL	Markov Clustering Algorithm
PLN	Partial Largest Node
SaaS	Software as a Service

Chapter 1

Introduction

Nowadays, Cloud Computing becomes more and more popular and growing rapidly since Eric Schmidt, CEO of Google, came up with the term "Cloud Computing". "I think it's the most powerful term in the industry," said Marc Benioff, CEO of Salesforce.com [1]. Almost all of the well-known enterprises in IT field were involved in this cloud computing carnival, including Google, IBM, Amazon, Sun, Oracle, and even Microsoft.

According to IBM's definition [2], "Cloud computing describes both a platform and a type of application." One of the key features of cloud computing is on-demand. No matter from software or hardware view, cloud computing will always involves over-the-Internet provision of dynamically scalable and often virtualized resources [3].

Industry focused on cloud computing not only because of its growing but also for its outstanding flexibility, scalability, mobility, automaticity, and the most important point is that it helps organizations to reduce cost.

Lots of companies have published their products which were claimed using cloud technology. The product line covered from low level abstraction such as Amazon's EC2 to higher level like Google's App Engine etc.

Social networks are getting popular and growing rapidly during recent years. They became consumers of cloud computation, because when the size of social network growing larger, it is impossible to process a huge graph on a single machine in a "real time" level execution time. Graph-based algorithms are becoming important not only on social networks, but also on IP networks, semantic mining and etc.

Map-Reduce [4] is a distributed computing model proposed by Google. The main purpose of Map-Reduce is to process large data sets paralleled and distributed. It provides a programming model which users can 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. Map-Reduce has become a popular model for developments in clouds computing.

Salesforce.com is a software as a service (SaaS) company that distributes business software on a subscription basis. It is best known for its Customer Relationship Management (CRM) products. Discovery team of Salesforce is working on Platform Intelligence with several distinct types of technologies and features including recommendations, trending topics and etc. The research on graph-based algorithms and data will play an essential role of the future development of the features. The Discovery team expected the Map-Reduce platform could help them on scaling problems when number of users was growing rapidly.

However, Map-Reduce also has limitations. Although lots of real world problems can be modeled by using Map-Reduce, there are still large quantity of them can't be presented very well. This thesis will focus on graph related algorithms, looking into how to present them using Map-Reduce and how Map-Reduce can model them well, in order to evaluate whether Map-Reduce, as a model, is good for graph processing or not.

1.1 Problem Statement

Firstly, the term "graph based algorithms" or "graph related algorithms" covered a large range of algorithms. In Chapter 2, how to categorize these algorithms and try to find pattern amongst them will be discussed. This problem also potentially indicates what class of algorithms can be implemented using Map-Reduce model.

Since Map-Reduce model has its own weakness (i.e. can't share any information among different slave machines while running map or reduce functions), not all of graph based algorithms can be mapped onto it. And, even though some graph related problems can be solved by Map-Reduce, they may not be the best solutions in cloud environment. Finding out which kinds of graph based algorithms are the most suitable for the Map-Reduce Model is another problem of the thesis. This will also be studied in Chapter 2.

The third problem to be solved is how to scale-out computation of graph based algorithms using a Map-Reduce programming model and computing infrastructure. The algorithms must scale to support algorithms on graphs with potentially millions of vertices and edges. This problem will be looked into with practical algorithm design in Chapter 3 to 4.

In order to scale, the algorithm implemented on Map-Reduce must be:

- Parallelizable.
- Support data distribution schemes viable on a Map-Reduce infrastructure.

1.2 Approach

A literature survey of how Map-Reduce have been applied to massive computation will be carried out. Related works will be reviewed and analyzed. This survey will mainly focus on graph related area and give an overview of graph based algorithm. A conclusion will be made about how Map-Reduce model is suitable to graph based algorithms.

After literature study, 2 particular algorithms will be chosen to be converted using Map-Reduce model. These choices will be made based on literature study. The chosen algorithms should be considered "hard to map-reduce", and should not have been previously applied on Map-Reduce model by anyone before.

These algorithms will be implemented on Hadoop [5] platform, an open source Map-Reduce implementation. Systematical and objective evaluations will be carried out after implementation is complete. The result of evaluation will be used to confirm the final conclusion.

1.3 Thesis Outline

Further background and related work are discussed and analyzed in Chapter 2 to find out the characters of graph-based algorithms and approaches which have been used to parallelize the algorithms. In Chapter 3, two algorithms are chosen to redesign using Map-Reduce model, several design strategies are proposed. Implementation details of these designs are explained in Chapter 4, and the best performed implementations will be taken into evaluation phase in Chapter 5. In the end, a conclusion will be made in Chapter 6.

Chapter 2

Background and Related Work

Lots of work about graph related computation have been carried out, not only algorithms but also infrastructures. These researches together provide a solid knowledge base and highly valuable guidance to this thesis.

2.1 Graph Based Algorithms

The term "graph" is a kind of mathematical structures. According to Claude Berge's definition [6], a graph is an ordered pair G = (V, E). V is the set of vertices or nodes which indicate the objects people need to study, and the elements in set E are edges or lines which are abstraction of relations between different objects. If each pair of vertices is connected by edges which do not have incoming or out coming states, the edges are undirected. Otherwise, the edges are directed from one vertex to another.

Vertices and edges could have values or states on them. Some graph theory problems are focused on those states (e.g. Minimal Spanning Tree Problem [7] and Traveling Salesman Problem [8]), but still some of them are not (e.g. Seven Bridges of Königsberg Problem [9]).

During advancement of computer science, mathematicians and computer scientists got a powerful tool which can be used to solve those graph related problems which were considered hard or impossible to solve before high performance computer was produced. Appearance of high performance computers significantly widened people's thoughts on algorithms design. Graph theory was begun to be used to solve the problems in computer science field.

Large number and varied kinds of graph based algorithms were proposed, covering graph construction [10, 11, 12, 13], ranking [14, 15], clustering [16, 17, 18], path problem [39] and etc. Take the shortest path problem as an example, Dijkstra algorithm [39] can be used to find the shortest path between two vertices in a directional weighted graph. This algorithm keeps the current shortest distance values from the starting vertex S to all of other vertices, once a shorter path is found by extension of directly connected edges (e.g. distance from A to B to C is shorter than current A to C value), the current value will be replaced with the shorter one. In the end of the algorithm, all of distance values can be guaranteed to be the shortest paths from S to all of other vertices. Most of graph-based algorithms can be categorized into two classes, vertex-oriented and edge-oriented. Take vertex filter [11, 12] as an example, which will focus on the value of vertex, data of each vertex will be processed separately, no message passing from one vertex to another, this is a typical vertex-oriented algorithm. But if the main part of an algorithm is to compute the states of edges or message transmitting, (e.g. PageRank [14]), the algorithm will be considered edge-oriented.

As shown in [19, 20, 21], most of edge-oriented algorithm can be solved in a vertex-centric way. However, in distributed computing, high volume of network traffic became a serious problem when perform edge-oriented algorithms on a vertex-oriented infrastructure. The reason which caused this problem is that, data of graph is stored in a vertex manner. If the state of an edge is modified by one

of its associated vertices, which will happen quite often in an edge-oriented algorithm, the other vertex must be notified to share the new state of the edge, in other words, one vertex must send message to the other one. If these two vertices are not located on same machine, much network traffic will be generated.

2.2 Infrastructures

Lots of researches have been done on distributed infrastructures of large scale graph processing. Some of them (such as [14], [16], and [18]) have already been in use to solve practical problems concern graphs, e.g. social networks, web graphs and document similarity graphs. The scale of graph could be from millions up to billions of vertices and trillions of edges. How to perform graph based computation efficiently in a distributed environment is the main purpose of these researches.

CGMgraph [23] is a library which providing parallel graph algorithms functionalities. In strictly definition, CGMgraph is not infrastructure since it tried to give out well implemented parallel graph algorithms rather than providing a framework to let programmer design the algorithm themselves. Another drawback is CGMgraph didn't fully consider fault tolerance which is crucial for large scale graph processing.

Pregel [20], which is also the name of river from Seven Bridges of Königsberg Problem, is a computational model proposed by Google to handle graph relevant large scale distributed computation. The model of Pregel is inspired by Valiant's Bulk Synchronous Parallel (BSP) model [22], which has been widely used in distributed systems. BSP model synchronizes the distributed computation by Superstep. In one Superstep, each machine performs independent computation using values which are stored in its local memory. After computation, it exchanges information with other machines and waits until all of machines have finished the communication. Taking ideas from BSP model, algorithms on Pregel are always presented as iterations of Supersteps. Pregel is using a vertex-centric approach, which means the data structure is constructed by vertex and its associated edges, weighted or non-weighted. Similar to Map-Reduce, Pregel is also hiding and managing the detail of distribution from users. Combiner mechanism is used to reduce the volume of network traffic. In contrast to Map-Reduce, the state of graph which stays in slave machines are stable. This provides a high locality for data reading and writing, which makes processing more efficient. In each iteration, vertices vote to halt if they do not have any further work. This mechanism also increases efficiency of computation. However, BSP model is not omnipotent and is still being extended and improved [24, 25].

Microsoft also involved in researches of large scale graph mining and processing. Surfer [19] is a large graph processing engine designed to execute in the cloud. It borrowed ideas from Map-Reduce, and made a remarkable improvement to perform graph processing using propagation. Map-Reduce is suitable for vertex-oriented algorithms which have flat data model (the elements of data don't have hierarchy or any other special relation), it has its inherent weakness on edge-oriented algorithms. To reduce the adverse impact of Map-Reduce, graph propagation is introduced. There are two functions to be defined by users to use propagation, transfer and combine. Transfer is used to send out messages or information from one vertex to its neighbors, while combine is used to receive and aggregate the messages at a vertex. The efficiency of graph computing on Surfer is coming from the tradeoff between Map-Reduce and propagation.

2.3 Map-Reduce and Hadoop

As described in the beginning of Chapter 1, Map-Reduce [4] provides a programming model which users need to defined map and reduce functions to specify what kind of calculation should be perform on partition of input.

One iteration of map and reduce functions is called Map-Reduce Job. Job will be submitted to the master node of a machine cluster. According to the definition of Map-Reduce Job, master machine will divide input data into several parts and arrange a number of slave machines to process these input data partitions in map functions. The output of map function will be intermediate result in form of key-value pairs. The result will be sorted and shuffled, then routed to the proper reducer according to the rule defined in the partitioner. The intermediate result will be processed again in the reducers, and turned into final result. Because of the programs are written in function style and all of scheduling works and fault tolerance are automatically done by Map-Reduce system itself, those programmers who don't have any parallel and distributed programming experiences can study easily and use Map-Reduce to model their own problem and process data using the resources on a cloud.

Taking PageRank as an example [21], graph is split into blocks and taken as input of map function. In map function, the value of each vertex is divided by the edge number of that vertex, and the result is stored as key/value pair {neighbor ID, result}. Before reduce function, each machine will fetch a certain range of key/value pairs onto its local storage, and perform reduce function on each key value. In this example, reduce function reads all of values under the same key (vertex ID), sum them up, and write the result back as the new value of this vertex.

Hadoop [5] which project by Apache, is an open source implementation of Map-Reduce model. Besides Map-Reduce function, it also provides a Hadoop Distributed File System (HDFS) [26], which is also inspired by Google's work on distributed file systems [27]. Because of its capability and simplicity, Hadoop have become a popular infrastructure for cloud computing. However, the Hadoop project is still young and immature. The weaknesses of Hadoop have manifested themselves mainly in the areas of Resource Scheduling [30], Single Point Failure [41] and etc.

Not only the team in Apache, but also the Hadoop developers from all over the world are continuously making their best effort to improve and perfect the Hadoop system and relevant projects of Apache. As an excellent large scale data mining platform, Hadoop is regarded as a good framework for graph related processing.

The effort in Carnegie Mellon University has carried out PEGASUS [28], a Peta-Scale Graph Mining System. It is described as a library which performs typical graph mining tasks such as computing the diameter of the graph, computing the radius of each node and finding the connected components. In [28], the authors claimed that, PEGASUS is the first such library implemented on the top of the Hadoop platform. Graph mining operations, such as PageRank, spectral clustering, diameter estimation, connected components and etc., can be seen as a series of multiplication of matrix. PEGASUS provides a primitive called GIM-V (Generalized Iterated Matrix-Vector multiplication) to achieve scalability of matrix multiplication and linear speed up on the number of edges. But not all of graph based algorithm can be presented as matrix computation, thus, PEGASUS is also not a universal solution.

With the joint effort of Beijing University of Posts and Telecommunications and IBM China Research Lab, X-RIME [29] is designed upon Hadoop to support social network analysis (SNA). X-RIME provides a library to do Map-Reduce and graph visualization. It builds a few value-added layers on top of Hadoop, and can be integrated with Hadoop based data warehouses, such as Hive [30].

There are some other researches focused on building an upper layer upon Hadoop, providing graph and matrix processing capability [31, 32].

Besides toolkits on Hadoop, plenty of researches have been done on how to convert graph based algorithms on Map-Reduce model. In [21], authors defined a pattern for applying graph based algorithms in Map-Reduce. Pretty much like Pregel, this pattern uses vertex-centric way to represent graphs, uses a BSP-like model to simulate the message passing procedure, and uses optimized combiners and partitioners to reduce the amount of message passed, thus, to reduce the network traffic. But again, BSP is a parallel model, thus, it is not perfect for all kinds of algorithms (i.e. sequential algorithms are still hard to be adapted on BSP model, e.g. go to the nearest vertex which have not been reached until no place to go).

2.4 Summary

The bloom of social and web networks stimulates demand for large scale graph mining. Lots of researchers now focus on how to make the graph processing scalable. Even some useful toolkits and powerful infrastructures have been in use, no single solution is perfect for every graph based algorithm. Matrix and BSP model are two of the most popular methods to solve graph-based problems in distributed environment. For the matrix method, adjacency matrix of graph is usually taken to be analyzed. However, when the vertices in graph have complex states, the adjacency matrix doesn't really help in analysis of these states (i.e. adjacency matrix can be used in representing relations between vertices, but not to represent the state of a vertex). BSP model defined a message exchange pattern in parallel environment using superstep. It makes some of graph-based algorithm easy to be parallelized, such as PageRank and etc. Hadoop became a popular platform and object of researches. Vertex-oriented algorithms which have a flat data model are excellently fitted on Map-Reduce, but edge-oriented ones do not fit as well. This is due to that, the edge-oriented algorithms usually need to share the states of edges among multiple vertices, and Map-Reduce is a "share nothing" model, it is inherently weak on edge-oriented algorithms. The algorithms, which can be applied on BSP model or can be presented as matrices problems, are possible to be implemented on Hadoop. However, because of the locality problem and overhead caused by Hadoop itself, the performance is not guaranteed to be as high as running on a BSP model infrastructure. Hadoop does't guarantee data locality, in other words, it will try to process the file block locally, but when local processing slots are occupied, then local file blocks may be processed by other machines. BSP infrastructures like Pregel would support data locality. However, even using a BSP-like or matrices method, edge-oriented algorithms can't be said as perfectly fitted on those models (i.e. BSP and matrices models).

Chapter 3

Design of Graph-Based Algorithms Using Map-Reduce Model

In this chapter, two algorithms are chosen to be analyzed and redesigned using Map-Reduce model. For each algorithm, several different strategies are proposed both in data sharing and computation. Theoretical analysis is made on these strategies to estimate the performance of them.

3.1 Chosen Algorithms

Graph based algorithms covered a wide range of problems. To make the research significant, the algorithms chosen here must be representative. These algorithms should fulfill the following factors:

1. Chosen algorithms should have been widely used.
2. They would not be considered easy to map-reduce.
3. They should be comparable, by which it means, they should have certain level of similarity on data inputs, outputs and purposes of algorithms themselves.
4. Those algorithms which have not been applied on Map-Reduce model would be considered first.

Clustering algorithms are widely used by designers and developers to solve a lot of realistic problems such as categorizing, recommendation and etc. The cluster algorithms on graph highly represented graph-relate algorithms (all of characters of graph should be considered, such as state of vertex, state of edge and direction of edge). Numerous cluster algorithms are reviewed including Optics [44], Clarans [43] and Agenes [45], some of these algorithms have been implemented using Map-Reduce model, e.g. K-mean [42]. After filtration, 2 particular algorithms came into scope. They are Markov Clustering Algorithm (MCL) [16, 17] and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [18].

Clustering algorithms are used to find clusters in data set. The data in the same cluster have similar pattern. In our case, clustering on graph, the cluster is defined as a subset of vertices in the graph, and the elements in one cluster are strongly connected to each other. Both MCL and DBSCAN achieved this goal using different methods.

Both of these two algorithms are well-known and plenty of research has been done on distributing and paralleling them to achieve an object of speed up. However, no material has shown that these algorithms had been implemented on any Map-Reduce platform. It is easy to know from the names of these two algorithms that they are both focused on clustering vertices in a graph. Since these two algorithms are usually taken to solve similar problems, choosing them will provide additional comparability to find out the performance and clustering result quality difference between each other in the following evaluations. The result can be interesting, especially for making decision on which algorithm to use in development of clustering feature based on Map-Reduce model.

3.1.1 Markov Clustering Algorithm (MCL)

MCL, firstly proposed by Stijn van Dongen [16], is a graph clustering algorithm which finds out the clusters of vertices by simulating random walks using matrices computation.

MCL is based on the property that if there exits clusters in a graph, then the number of links within a cluster will be more than the number of links between different clusters, it is easy to tell that, if you start travelling random on the graph from one vertex, the end point you arrived is more likely in the same cluster. This is the basic principle of MCL using for clustering. By using random walk, it is possible to find out the clusters on the graph according to where the flow concentrated.

Random walk is simulated and calculated in MCL by using Markov Chain [33] which is proposed by A. A. Markov in 1906. Markov Chain is presented as a sequence of states X_1, X_2, X_3, etc. Given the present state, the past and future states are independent, which means that the next state of system depends on and only depends on the current state of system. This property can be formally described as below:

$$\mathbf{Pr}\ (\ X_{n+1} = x \mid X_1 = x_1, X_2 = x_2, X_3 = x_3, \ldots, X_n = x_n\ ) = \mathbf{Pr}\ (\ X_{n+1} = x \mid X_n = x_n\ )$$

The function **Pr** is used to compute the state (probability value) in a Markov Chain. Given state sequence X_1 to X_n, the state X_{n+1} can be calculated by **Pr**. However, **Pr** will get the same state value for X_{n+1} by only using X_n as input. It means that, the value of state X_{n+1} only depends on state X_n.

In MCL, the states are represented as probability matrix. The vertex in graph can be considered as a vector. Edge of the vertex can be seen as element of vector. Putting all of vectors of nodes in the graph together by column, a matrix which shows the connectivity of graph will be formed. In this matrix, if the value of each element indicates the possibility to "walk" through a certain edge, then it is called probability matrix.

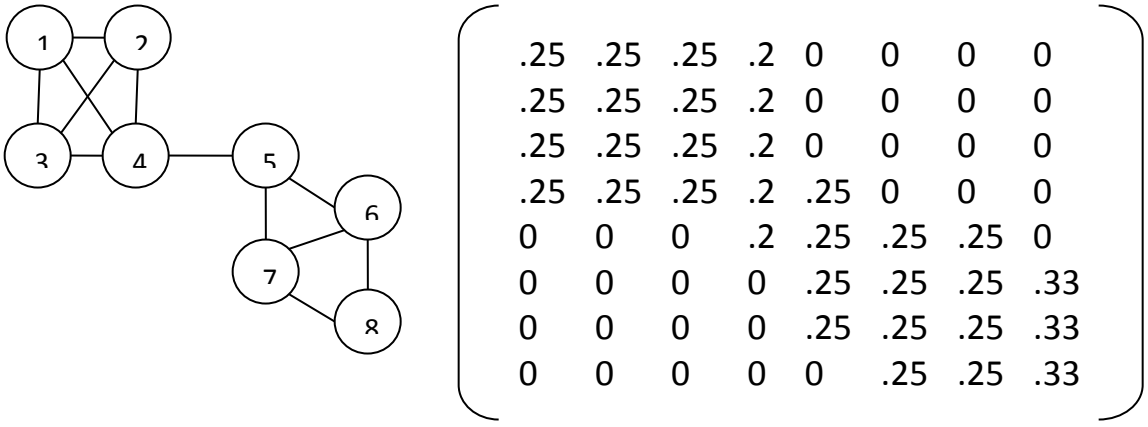


Figure 3.1: Construction of Possibility Matrix (1)

As shown in Figure 3.1, the graph is constructed by 8 vertices. Each vertex has several edges connected to other vertices. Assuming uniform probabilities on all edges while travelling from one

vertex (including self loop), the probability matrix should be looked like the one listed on the right side of figure, the sum value of each column is exactly 1.

By taking power of the probability matrix, MCL will act a random walk to allow the flow going between vertices which are not directly connected. Take the graph in Figure 3.1 as an example, after matrix multiplication, the value in column 1 row 5 would be 0.005, this shows that there is a flow from vertex 1 to vertex 5 after random walk. This operation is called *expansion* in MCL.

As discussed before, random walk is much more likely to stay in densely linked part of graph than cross the boundary which will have sparse edges. However, after an extended period of random walk, this character will be weakened.

The purpose of MCL is to find clusters in the graph, in other words, to make sure random walks stay in cluster. Thus, only using expansion obviously can't achieve the goal.

After taking powers of the possibility matrix, the values will be higher for those that are within clusters, and lower between the clusters. MCL provides another operation called *inflation* to strengthen the connections between vertices which are already been strong and further weaken weak connections. Raising power (exponent larger than 1) of each value in the matrix will make this gap even greater. Taking power of each element in a matrix is also known as Hadamard power. In the end of inflation process, probability (percentage) is recalculated by column, and then the sum value of each column is 1 again. The exponent parameters here will influence the result of clustering, larger exponent value will result in more and smaller clusters.

Repeating execution of expansion and inflation will separate the graph into several parts without any links between them. Those connected vertices stay in a partition represent a cluster according to the MCL algorithm computation.

3.1.2 Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

Similar to MCL, DBSCAN is also an algorithm which is used to perform clustering on graphs. However, DBSCAN is based on a different clustering theory.

DBSCAN is a density-based clustering algorithm which defines clusters on a graph based on the number of neighbors of a vertex within a certain distance. Vertices in one cluster are *density-reachable*. The term *density-reachable* is defined as below:

1. If a vertex q is *directly density-reachable* from a vertex p, then q is not farther away than a given distance ε from p, and vertex p must have at least k number of neighbors within distance ε.
2. If q is called *density-reachable* from p, then there is a sequence of vertices $v_1, v_2, \ldots, v_n$ which $v_1 = p$ and $v_n = q$, and each $v_{i+1}$is *directly density-reachable* from v_i.

However, the *density-reachable* is not a symmetric relationship. For example, the distance between vertex p and q is no larger than ε, but q doesn't have as sufficient neighbors as p has, thus, we only can say that, q is density-reachable from p, but not vice versa.

Another term is used to describe the connectivity based on *density-reachable*: there are two vertices *p* and *q*. If there exists a vertex *m* which can *density-reach p* and *q*, then *p* and *q* are *density-connected*.

By defining the notions above, a cluster in DBSCAN algorithm can be described as below:

1. The nodes and only these nodes which are mutually *density-connected* belong to the same cluster.
2. If *q* does not have enough neighbors and is *density-reachable* from one or several vertices, then *q* belongs to either cluster which these vertices belong to.
3. If *q* does not have enough neighbors and isn't *density-reachable* from any vertices, then *q* doesn't belong to any cluster. The vertex *q* is called Noise in the graph.

To perform the DBSCAN algorithm on graph, two parameters are required: ε and *k*. The parameter ε is a value of distance, which within the distance, vertices calculates the number of neighbors. The other number *k* is a value used as threshold which represents the number of directly linked neighbors within the distance ε.

The algorithm randomly chooses a vertex which has not been visited yet as the starting point. The number of neighbors within ε range is counted, and if it has sufficient neighbors (at least no less than *k*), then a new cluster will be created and mark this vertex belonging to the cluster. Otherwise, the vertex is labeled as noise.

If a vertex is part of a cluster with at least *k* neighbors within ε distance, all of these neighbors belong to the same cluster. Thus, all of these neighbors are added into a waiting list, and then the algorithm will choose a vertex from this list to do the same computation described above until there is no vertex in the list anymore, which means that the cluster is fully discovered. Then the algorithm will randomly choose another vertex which have not been visited and start a new turn of finding a cluster.

It should be noticed that, even if a vertex is marked as noise, it may be added to a cluster later. The reason is that, this vertex may be *density-reachable* from some other vertices which have not been processed yet, but sooner or later these unprocessed vertices will be visited, and the "noise" vertices which are *density-reachable* from newly processed vertices will be added to the cluster at that time.

3.2 Design of MCL on Map-Reduce

3.2.1 Problem Analysis and Decomposition

As explained in section 3.1.1, the MCL algorithm can be decomposed into matrix computation processes. Whole problem can be represented as iterations of N*N matrix square and Hadamard power of matrix.

Considering the problem in reality like social or web networks, almost all of vertices in the graph have edges linked to other vertices, which means that, in a probability matrix or connectivity matrix, almost all of rows and columns have values.

Assume there are 1 million vertices in the graph, and each vertex has 100 edges connected to other vertices in average, it is easy to see that, the connectivity matrix is huge and extremely sparse.

Thus, the core problem of designing MCL on Map-Reduce model is how to compute a sparse matrix using Map-Reduce. More specifically, the problems are:

1. How to express a matrix for MCL problem
2. How to compute power of a matrix in a manner that scales
3. How to compute Hadamard power of matrix in a scalable way
4. How to compose the solutions of the problems mentioned above into an algorithm implementing MCL using the Map-Reduce model

3.2.2 Matrix Representation

In reality, the problem of how to build a probability matrix is always the first step of the entire MCL computation procedure.

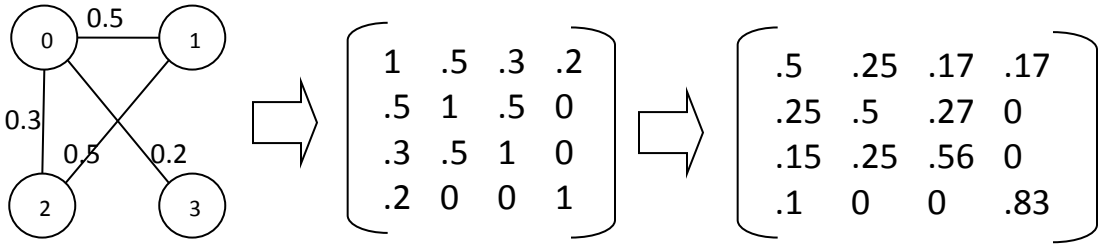


Figure 3.2: Construction of Possibility Matrix (2)

Taking similarity graph as an example, converting the weighted graph into a matrix style, a similarity matrix is shown in the middle of Figure 3.2. By calculating the probability percentage on each column, the probability matrix is given in the end. It is easy to be noticed that, the similarity matrix is symmetric, but no longer for probability matrix.

To present a sparse matrix, the common way widely used is that, only the value of element which is not 0 and the index of this element are stored. Skipping all of 0 values in a sparse matrix could keep the physical storage size of the matrix smaller comparing to a naïve way which stores all of values in matrix including 0. Using this method to describe the probability matrix in Figure 3.2, the matrix can be explained in 2 different ways:

1. *ColumnID : RowID : Value*

    - {0:0:0.5, 0:1:0.25, 0:2:0.15, 0:3:0.1, 1:0:0.25, 1:1:0.5, 1:2:0.25, 2:0:0.17, 2:1:0.28, 2:2:0.56, 3:0:0.17, 3:3:0.83}

2. *ColumnID* => {*RowID: value* *}

    - Column 0=> {0:0.5, 1:0.25, 2:0.15, 3:0.1}
    - Column 1 => {0:0.25, 1:0.5, 2:0.25}

- Column 2=> {0:0.17, 1:0.28, 2:0.56}
- Column 3=> {0:0.17, 3:0.83}

The first way, as shown above, will provide full accessibility to individual elements in a matrix using column and row index. While using the second way, the whole column of data will be retrieved first, then using row index to find the corresponding value. The second way will take less storage space comparing to the first one because it records each column index only once. In case of MCL, data is loaded by row and column. Using the second way will most benefit the performance both in time and space.

Beside the two methods of representing the matrix above, there is another way that can be used by splitting matrix into several sub-matrices [28, 34].

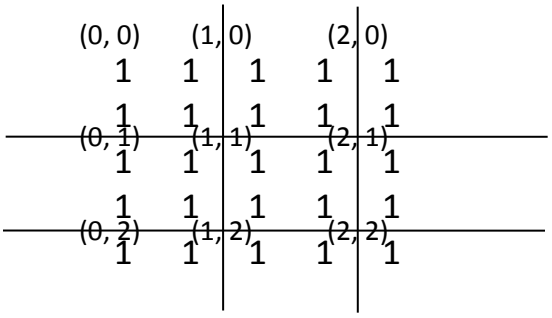


Figure 3.3: Sub-Matrices

As shown in Figure 3.3, a 5*5 size matrix is split into several sub-matrices, which the size is 2*2 at most. The matrices which stay at the end of columns and rows are not full size sub-matrices. Each sub-matrix gets its own ID in format of (columnNumberOfSubMatrix, rowNumberOfSubMatrix). Using the sub-matrices way to describe the matrix in Figure 3.2 with sub-matrix size 2*2, the result would be:

3. Sub-Matrix (columnNumberOfSubMatrix, rowNumberOfSubMatrix) => {localColumnID: localRowID: value *}

    - Sub-Matrix (0, 0) => {0:0:0.5, 0:1:0.25, 1:0:0.25, 1:1:0.5}
    - Sub-Matrix (0, 1) => {0:0:0.15, 0:1:0.1, 1:0:0.25}
    - Sub-Matrix (1, 0) => {0:0:0.17, 0:1:0.28, 1:0:0.17}
    - Sub-Matrix (1, 1) => {0:0:0.56, 1:1:0.83}

Need to notice that, each sub-matrix is using its own index to the element, not the global one. However, the indices used by sub-matrices and global indices can be transformed via the equations:

- $C = c + n * columnNumberOfSubMatrix$
- $R = r + n * rowNumberOfSubMatrix;$

C and *R* here indicate the global indices of rows and columns of elements in the matrix, and *c, r* together are local indices which show the positions of elements in sub-matrices. The number *n* is the length of vector in a sub-matrix which means that the size of a sub-matrix is *n*n*.

Given global indices, local indices in sub-matrices can be easily calculated using the following method:

- *columnNumberOfSubMatrix = floor (C / n);*
- *rowNumberOfSubMatrix = floor (R / n);*
- *c = C – n * columnNumberOfSubMatrix = C – n * floor (C / n);*
- *r = R – n * rowNumberOfSubMatrix = R – n * floor (R / n);*

The advantages while using this method to represent a matrix will be shown in the following section.

3.2.3 Power of N*N Sparse Matrix (Expansion)

Numerous matrix multiplication methods have been proposed for the Map-Reduce model [28, 34]. However, the expansion operation of the MCL algorithm is essentially computing the power of matrix. Therefore, performance of algorithms can be improved (i.e. less memory consumption and different mapping logic).

To expand the random walk flow, the power of probability matrix is calculated. According to mathematical theory, the power of matrix is computed as following:

- If a_{ij} is an element of matrix M and b_{ij} is an element of M^2, then $b_{ij} = \sum_{k=0}^{n} a_{ik}\, a_{kj}$ $(i\ is\ row, and\ j\ is\ colume)$

It is easy to find out that, to compute a value in matrix M^2, one row and one column of values in matrix M are needed.

In a parallel or distributed environment, sharing data is an essential problem. Especially in Map-Reduce, it is a "share nothing" model when performing the map tasks or performing reduce tasks. Thus, minimizing the amount of data sharing is a critical issue in Map-Reduce algorithm designing. Since two lines of data in matrix are needed to compute a single value in taking power of matrix, reusing the data which have been used to compute another value is the core strategy which is needed to discuss.

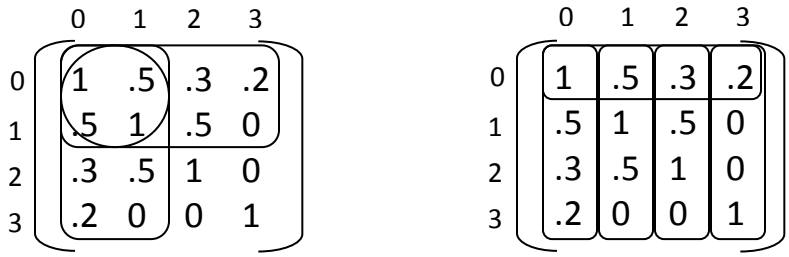


Figure 3.4: Two Ways of Element Calculation

Assume 4 values are needed to be calculated in M^2, according to Figure 3.4, the least number of data needed is 4 vectors in matrix M when these 4 values are given in format as a 2*2 matrix (left). If these values stays in line (right), then 5 vectors of data in M will be needed to calculate them. Thus, splitting the computation target matrix into several square sub-matrices is the best way to scale up the matrix computation.

While computing power of matrix, to calculate one value needs all of values from its column and its row. If the matrix M^2 in divided into 4 sub-matrices, half of columns and rows will be read to compute one quarter of M^2. In the view of a single element, it will be read 4 times during the whole procedure of calculating matrix M^2. If the matrix is split into *s***s* parts, then a single element in matrix M will be duplicately read 2**s* times comparing to before division.

Some values are defined here before going into the detail of algorithm design:

- *N* is the number of vertices in the graph, thus, the size of matrix is *N* * *N*.
- *n* is the number of rows and columns in a sub-matrix, then the size of sub-matrix is *n* * *n*
- If *N* mod *n* = 0, *p* = *N* / *n*, otherwise, *p* = *ceiling (N / n)*, which means that the matrix will be divided into *p***p* sub-matrices.

Based on sub-matrices, the following strategies of matrix square are proposed:

Strategy 1

A simple rule is used to decompose the work into map function and reduce function during all of the algorithm designs:

- No matter which kind of data was taken as input of map function, perform analysis on the data as much as possible until no further result can be achieved
- Combine the results using particular keys, in order to make reducer be able to perform further analysis. The definition of keys should also consider what kind of result reducer could get
- If the work is not able to be done in one job, another job is needed. No matter work is done in one job or not, the output format of reduce function should also be considered, in order to benefit the following analysis of data.

Before expansion, the data of matrix M will be organized by column, and converted into probability matrix using Algorithm 1.1:

Algorithm 1.1: MarkovMatrix()

```
Map-Reduce Input: connectivity (similarity)sparse matrix
Map-Reduce output: markov (probability) matrix
class MarkovMapper
        method map(column)
                sum=sum(column)
                for all entry ∈ column do
                        entry.value=entry.value/sum
                        collect {column.id, {out, entry.id, entry. value}}
```

```
                collect {entry.id, {in, column.id, entry.value}}
class MarkovReducer
    method reduce(key, list {sub-key, id, value}})
        newcolumn=Ø
        newrow=Ø
        for all element ∈ list do
            if sub-key is "out" then newcolumn.add({key, id, value})
            else newrow.add({id, key, value})
        collect newcolumn
        collect newrow
```

To convert the matrix into probability matrix, map function will read the matrix by column, and calculate the probability value of each element of the matrix, and then, collect the column ID or row ID as key, the probability as value. Reduce function fetches all of the values under a same key, and put it back to file system, in our case, the new rows and columns with probability values will be put back to file system.

After that, the data probability matrix will look like the following:

- Column 0-> {0 0.5, 1 0.25, ….}
- Column 1-> {0 0.25, 1 0.5, ….}
- ….
- Row 0-> {0 0.5, 1 0.25, ….}
- ….

Because these data are output from the Algorithm 1.1, they are stored on hard-drives. It means that 2 full aspect of probability matrix are stored on disk, one by column and another by row. Each map task is taking charge in calculation of one sub-matrix of M^2, so if the size of sub-matrix is *n*, the input of map task will be *2*n* lines of data, in other words, if the average number of elements in each column or row in matrix M is *X*, the number of input data (elements) will be *2*n*X*. For instance, as shown in Figure 3.5, for sub-matrix block (0, 0) which in 2*2 size, it will get Column 0, 1 and Row 0, 1 as input of map-reduce.

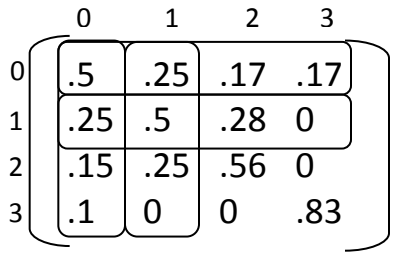


Figure 3.5: Sub-Matrix Calculation (Information Stored by Column and Row)

Expansion can be done in one iteration of map and reduce tasks. In mapper, all of elements contained in this sub-matrix can be simply calculated, and in reducer, the results are re-organized by column to make them easy to use during the next stage (inflation).

Algorithm 2.1: expansion()

```
Map-Reduce Input: markov matrix
Map-Reduce output: expanded matrix
class ExpansionMapper
        method map(column, row)
                sum=0
                for all entry ∈ column do
                        if entry.id is contained in row.idSet then
                                sum=sum+entry.value*row.get(id).value
                if sum!=0 then collect {column.id, row.id, sum}
class ExpansionReducer
        method reduce(key, list {row, value}})
                newcolumn=Ø
                for all element ∈ list do
                        newcolumn.add({key, row, value})
                collect newcolumn
```

In map function, all of columns and rows which are used to calculate a sub-matrix are read. Calculation is done by classic matrix multiplication method. The result will be collected using both column ID as key, row ID and new value of element as value. Reduce function gets all of values by column ID, and put them back by columns.

This method of computing power of matrix is quite straight forward and low cost. However, there are still some obvious drawbacks. The matrix data stored on disk are duplicated, totally depending on number of sub-matrices. More sub-matrices are split, more data are needed to store on disk. Furthermore, to compute fast, all of columns and rows data should stay in memory, however, when the scale of matrix is growing, amount of these data will also be increased. Then it will be hard to put all of data into memory.

Strategy 2

Compared to Strategy 1, Strategy 2 uses a more complicated algorithm, but will have better memory performance.

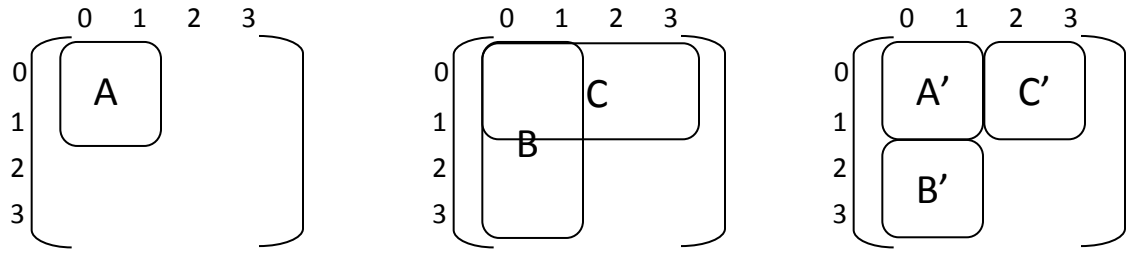


Figure 3.6: Sub-Matrix Calculation (Information Stored by Sub-Matrix)

Strategy 2 is based on the fact that, calculating a sub-matrix can be further decomposed into several parts. For example, as shown in Figure 3.6, sub-matrix A in M^2 can be calculated by A=CB (B and C are left and top parts of matrix M), but this process can also be expressed as A= A'A'+C'B', where A', B' and C' are sub-matrices of M. According to this property, calculation can be decomposed and it will be much friendlier in memory consumption comparing to Strategy 1. In Strategy 1, the data of A' will stay in memory twice (in both horizontal and vertical vectors), but in Strategy 2, it is not necessary to put A' twice in memory, once is enough, this part of data can be reused in upcoming calculations.

To use Strategy 2, the probability matrix is no longer needed to be represented as both column and row format, only column style is enough. Thus, Algorithm 1.1 can be simplified into Algorithm 1.2:

Algorithm 1.2: MarkovMatrix()

```
Map-Reduce Input: connectivity (similarity)sparse matrix
Map-Reduce output: markov (probability) matrix
class MarkovMapper
        method map(column)
                sum=sum(column)
                for all entry ∈ column do
                        entry.value=entry.value/sum
                        collect {column.id, {entry.id, entry. value}}
class MarkovReducer
        method reduce(key, list {id, value}})
                newcolumn=Ø
                for all element ∈ list do
                        newcolumn.add({key, id, value})
                collect newcolumn
```

Calculation of sub-matrix can be described formally as below:

- $\mathrm{SubMatrix}_{n+1,i,j} = \sum_{k=0}^{n} \mathrm{SubMatrix}_{n,i,k}\, \mathrm{SubMatrix}_{n,k,j}$ (i *is row, and* j *is colume*)

If $\mathrm{SubMatrix}_{n,i,k}\mathrm{SubMatrix}_{n,k,j}$ is taken as unit of calculation, then each element in matrix M will be involved in 2**p*-1 units of calculations. The result is called a unit of $\mathrm{SubMatrix}_{n+1,i,j}$. In a sub-matrices' view, a sub-matrix S_{ij} will be involved in computations of $S_{ij}S_{jk}\ (k = 0 \ldots . p-1)$ and $S_{ki}S_{ij}\ (k = 0 \ldots . p-1)$, also 2**p*-1 times. Thus, each units of calculation can be simply done and the sub-matrix can be computed by sum of certain units.

Algorithm 2.2: expansion()

```
Map-Reduce Input: Markov matrix
Map-Reduce output: unit matrix
class ExpansionStepOneMapper
        method map(column)
                blockColID = floor (column.id / subMatrixSize)
```

```
            for all entry ∈ column do
                blockRowID = floor (entry.id / subMatrixSize)
                x=0
                do
                    collect {{blockColID, x, blockRowID}, {column.id, entry.id, entry.value}}
                    x=x+1
                while (x<p)
                x=0
                do
                    if (x!= blockColID) then
                        collect {{x, blockRowID, blockColID}, {column.id, entry.id, entry.value}}
                    x=x+1
                while (x<p)
class ExpansionStepOneReducer
    method reduce (key {blockColID, blockRowID, subBlockID}, list {col, row, value}})
        matrix1, matrix2, matrix3=Ø
        for all element ∈ list do
            if blockRowID * n <= row< blockRowID * n + n and subBlockID * n <= col < subBlockID * n + n then matrix1.put({{col, row}, value})
            if blockColID * n <= col < blockColID * n + n and subBlockID * n <= row < subBlockID * n + n then matrix2.put({{col, row}, value})
        matrix3 = matrix1 * matrix2
        collect matrix3
Map-Reduce Input: unit matrix
Map-Reduce output: expanded matrix
class ExpansionStepTwoMapper
    method map(matrix)
        for all element ∈ matrix do
            collect {{matrix.columnID, matrix.rowID}, {element.col, element.row, element.value}}
class ExpansionStepTwoReducer
    method reduce (key {matrixColumnID, matrixRowID}, list {col, row, value}})
        matrix = Ø
        for all element ∈ list do
            matrix (list.col, list.row) = matrix (list.col, list.row) + list.value
        collect matrix
```

In algorithm 2.2, the task is divided into 2 steps, one Map-Reduce job per step.

The first job is used to calculate all of the units of sub-matrices. During the first step, data in M are read and sent to reducer using key {blockColID, blockRowID, subBlockID}, where blockColID and blockRowID together is the index of sub-matrix in M^2 which is wanted to calculate, and subBlockID indicates which unit of sub-matrix it is referred to. In reducer, the data under one key will belong to the sub-matrices (columnID = subBlockID, rowID = blockRowID) or (columnID = blockColID, rowID = subBlockID) in M. So it is easy to separate them into two parts, then

multiplication of these two sub-matrices will be done. The result after first step will be different units of sub-matrices in M^2, stored on disk by key given to reducer.

The second job is used to sum up the units which belong to the same sub-matrix, therefore, getting the result of expansion by sub-matrix. To compute the sub-matrices, in step 2, all of units will be read and sent to reducer according to the first two number of the key. Then reducer will simply sum up the received units of matrices.

According to practical experience, if an algorithm can be done in one job of Map-Reduce, then decomposing it into several jobs usually will cost more time on execution. This extra execution time mainly comes from the overhead of Map-Reduce infrastructure itself, including task creating and network transmitting latency. Thus, a strategy 3 is purposed to integrate the two steps above.

Strategy 3

Another mechanism provided by Map-Reduce called Partitioner is used in this strategy. Partitioner is defined to arrange the certain range of key-value pairs from map task output into specified reducer. By using it, all units of a sub-matrix will be guaranteed to be calculated in one reducer. Then the addition work can be done in the reducer, instead of doing it in another iteration of map-reduce job.

Similar job has been done by John Norstad [34]. Different to John's work, the method proposed in this thesis is optimized for power of matrix (only one multiplier matrix stayed in memory instead of both multiplier and multiplicand matrices) and supports calculating multiple sub-matrices in one reducer.

The algorithm will be rewritten as following:

Algorithm 2.3 expansion()

```
Map-Reduce Input: Markov matrix
Map-Reduce output: expanded matrix
class ExpansionMapper
        //same as class ExpansionStepOneMapper  in Algorithm 2.2
class ExpansionPartitioner
        method getPartition (key {blockColID, blockRowID, subBlockID}, list {col, row, value}})
                result = blockColID % numPartitions
                return result;
class ExpansionReducer
        matrix1, matrix2, matrix3 = Ø
        blockColIDBefore, blockRowIDBefore = Ø
        method reduce (key {blockColID, blockRowID, subBlockID}, list {col, row, value}})
                if (blockColIDBefore !=  blockColID & blockRowIDBefore != blockRowID) then
                        blockColIDBefore = blockColID
                        blockRowIDBefore = blockRowID
                        if (blockColIDBefore != Ø) then
                                collect matrix3
                for all element ∈ list do
```

```
            if (blockRowID * n <= row< blockRowID * n + n and subBlockID * n <= col <
            subBlockID * n + n) then matrix1.put({{col, row}, value})
            if (blockColID * n <= col < blockColID * n + n and subBlockID * n <= row <
            subBlockID * n + n) then matrix2.put({{col, row}, value})
        matrix3 = matrix3 + matrix1 * matrix2
    method cleanup()
        if (blockColIDBefore != Ø) then
            collect matrix3
```

Here the definition of map function is the same to ExpansionStepOneMapper in Algorithm 2.2. However, partitioner is used to route result of map functions to proper reducer, in order to calculate the sub-matrices in one step (reducer). The data will be sent to reducers using the rule which defined in partitioner. The keys will be processed in a sorted order in reducers. Using these properties, one or more sub-matrices can be calculated in a single reducer, which means that the number of reduce tasks can be controlled flexibly.

To compute multiple sub-matrix in single reducer, two parameters, blockColIDBefore and blockRowIDBefore, are needed. They are used to keep the sub-matrix ID which is processing. When the ID changed, it means that the calculation of former sub-matrix have finished, then the result of the former sub-matrix will be write back to file system.

3.2.4 Hadamard Power of Matrix (Inflation)

Compare to computing square of a matrix, Hadamard power is much easier to compute. Given a matrix, the Hadamard Power involves calculating the power of each element in the matrix. But the Inflation step of MCL doesn't only contain computation of Hadamard Power, but also need to convert the powered matrix into a new probability matrix, in other words, to compute the percentage of each element by column.

Regardless if we use strategy 1, 2 or 3 in an expansion step, inflation can be done during one map-reduce iteration. For strategy 1, the input data of inflation is given by format of column, so the inflation procedure can be done in mapper, and use reducer to generate the data for the next round of map-reduce (Algorithm 3.1).

Algorithm 3.1 inflation()

```
Map-Reduce Input: expanded matrix
Map-Reduce output: markov (probability) matrix
class InflationMapper
    method map(column,r)
        sum=sum(all column.entry.value^r)
        for all entry ∈ column do
            entry.value=entry.value^r/sum
            collect {column.id, {out, entry.id, entry. value}}
            collect {entry.id, {in, column.id, entry.value}}
class InflationReducer
        // same as the MarkovReducer in Algorithm 1.1
```

As shown in the pseudo code above, the data is read by column in the map function. Each element of the column raised to the power of a coefficient number 'r'. Then, the new values in this column are normalized i.e. Normalization means all of new values are divided by the sum of them, after that, the sum of these values is 1.

The output of strategy 2 and 3 are in format of sub-matrices, thus, Inflation cannot be done in mapper directly, however, the reducers can perform the inflation work while mappers only read data from file system and transfer them to proper reducers by taking column ID as intermediate keys (Algorithm 3.2).

Algorithm 3.2 inflation()

```
Map-Reduce Input: expanded matrix
Map-Reduce output: markov (probability) matrix
class InflationMapper
        method map(submatrix)
                for all entry ∈ submatrix do
                        collect {entry.columnID, {entry.rowID, entry. value}}
class InflationReducer
        method reduce(columnID, list {rowID, value} )
                sum=sum(all list.value^r)
                for all {rowID, value} ∈ list do
                        value=value^r/sum
                        collect {columnID, {rowID, value}}
```

3.2.5 Integration

When designing the algorithms for probability matrix calculation, expansion and inflation, the input and output data format is strictly considered. The output of probability matrix calculation and inflation is in the same format as input of expansion. The output format of expansion is the same comparing to input of inflation. Thus, the designed module of algorithms can be combined into a Job Chain to achieve the clustering goal of MCL.

The composition of jobs should look like:

[MarkovMatrix Job] ([Expansion Job][Inflation Job])+

MarkovMatrix Job (i.e. probability calculation) will be run in the beginning, then, expansion and inflation will be performed in order. Expansion and inflation could be performed one or more times, until algorithm terminates through convergence (discussed in Section 4.2.2).

3.3 Design of DBSCAN on Map-Reduce

3.3.1 Problem Analysis

DBSCAN is a typical sequential algorithm. In each stage of this algorithm, global state is the only determinant of vertex's state which is not clustered yet. In other words, if we want to find which cluster a vertex belongs to, we need to know all of other vertices and their state. This character

makes it hard to parallelize DBSCAN algorithm. Some research has been done to parallelize DBSCAN algorithm and use it on data clustering in database [35, 36], but those methods proposed are not fit to graph clustering.

The essence of DBSCAN is expansion. The number of clusters in DBSCAN is not pre-defined, and the size of each cluster is also not known. To achieve fast converge and good load balance of the algorithm, parallelization of DBSCAN requires:

1. Multiple clusters do expansion in parallel
2. In one cluster, multiple vertices can do expansion in parallel

Our goal is to parallelize DBSCAN for graph computation by achieving both of these goals.

3.3.2 Problem Reformulation

In the original DBSCAN algorithm, there are 3 kinds of nodes (vertices) in the graph:

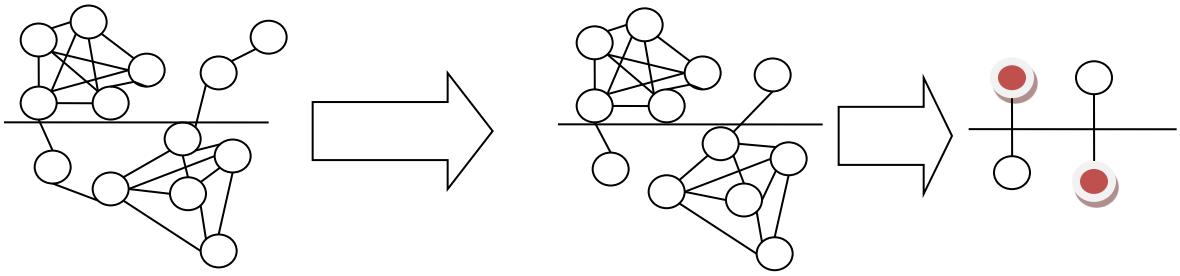
1. **CORE NODE**: a core node has at least *K* neighbor nodes in distance ε.
2. **EDGE NODE**: edges nodes have less than *K* neighbors in distance ε, but at least one of these neighbors is core node.

3. **NOISE**: noise nodes are similar to edge nodes. The only difference is that, noise nodes are not connected to any core node in distance ε.

A cluster is constituted from core nodes and edge nodes. Edge node may be *density-connected* to core nodes which belong to more than one cluster, then, this edge node belongs to one of these clusters..

A naïve thought of parallelizing DBSCAN is that, the graph is divided into several sub-graphs, and sub-graphs are assigned to various slave machines. Each sub-graph is processed on slave machine separately, and the results of them are merged later.

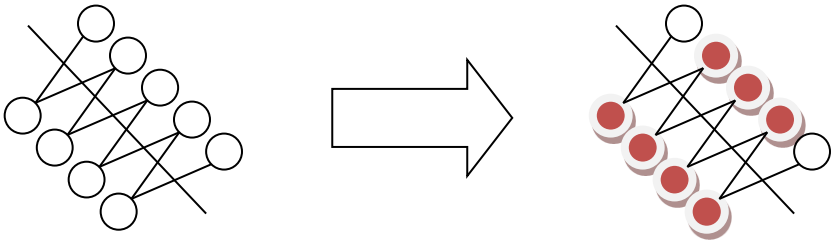
However, this naïve strategy is hard to implement. The first problem faced is how to divide the graph, in order to generate least sub-clusters which are needed to be merged. As explained before, original DBSCAN algorithm doesn't need to merge results, after all nodes in the graph have expanded, clustering is done. Another problem is how to merge these results. After abstracting of the problem (Figure 3.7), we can see this naïve strategy is essentially decreasing the size of problem and changing it to a connectivity problem, but not solving the merge problem. The graph is split into two parts, each partition of graph will be processed using DBSCAN, but keep density-connections to other partitions. The right part of Figure 3.7 shows the result clusters on both graph partitions (red node) and the connections to other partition.



K = 3, distance = infinite

Figure 3.7: Naïve Method of Parallel DBSCAN (Better Case)

The result above shows a fairly well decreasing of problem size. But, not all of graph can be scaled down using this method. Figure 3.8 shows a worst case of this strategy, it is easy to see that, the size of problem doesn't drop at all.

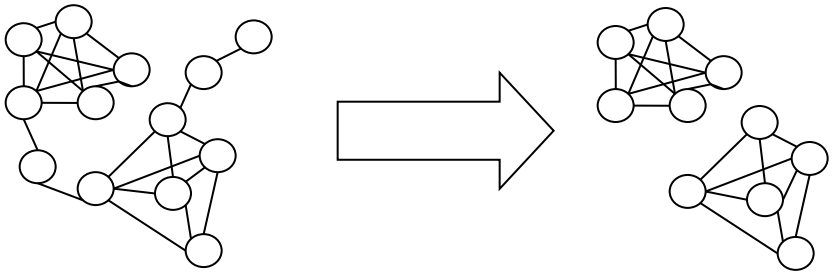


K = 2, distance = infinite

Figure 3.8: Method of Parallel DBSCAN (Worst Case)

Of course, this is only an extreme situation when graph partitioning is not considered. Nevertheless, it is enough to show that, this strategy is not realistic as a general solution. Thus, reformulation of DBSCAN algorithm is necessary when moving it to Map-Reduce infrastructure.

Since merging of results is not avoidable when running DBSCAN parallel, it is better to keep the consumption of results merging the least. To avoid the situation above (Figure 3.8), another strategy is proposed.



Filtering K = 3, distance = infinite.

Figure 3.9: Graph Filter

A slight change in the problem formulation will show an interesting aspect. We will focus on filtering all of the noise nodes and the edge nodes out of the graph. Figure 3.9 illustrates that after filtering, the nodes in the graph are separated into 2 parts which do not have any connections between them. We can call each part an **ISLAND**. Nodes in an island will follow the definition below:

1. If there is a path between 2 nodes, then these 2 nodes are connected.
2. All nodes in an island will be connected to each other.
3. If 2 nodes belong to different islands, then they will not be connected to each other.

From the original definition of DBSCAN algorithm, it is easy to tell that, each island is a cluster, and it is easy to prove:

1. After filtering, the core node which directly-connected to another core node will have an edge between them.
2. Because in an island each pair of nodes will have a path between them, thus, all of nodes in the island are *density-connected* (defined in Section 3.1.2). Therefore the nodes in one island belong to the same cluster.
3. If two core nodes *p* and *q* are not *density-connected*, there does not exist a node *v* which *p* and *q* can be *density-reachable* (defined in Section 3.1.2) from. Thus, there is no node that can connect to both *p* and *q* in the filtered graph. Therefore, nodes staying in different islands belong to different clusters.

Thus, the DBSCAN problem has been reformulated into a connectivity problem. The target is to find out all of the islands which are the clusters on the original graph.

3.3.3 Solutions - Kingdom Expansion Game

Although DBSCAN is reformulated as a simple connectivity problem, an efficient way of finding out islands is still necessary. An interesting strategy proposed for this problem is called Kingdom Expansion Game.

After filtering, all of the remaining nodes on the graph are core nodes. The fact that they are all core nodes provides another benefit. Since no noise and edge nodes remain in the filtered graph, every node will belong to some cluster. The implication is that, choosing any node as the starting point of expansion will directly build a cluster, not exclude noise. This property is meaningful when implementing DBSCAN algorithm on Map-Reduce platform. If taking one time of expansion as a Map-Reduce job, processing noise node will not give out any new nodes which can be taken as the next job's starting point. Because there exists an overhead for each Map-Reduce job, more jobs means more execution time, no matter whether processing is performed in the job or not. To run the algorithm efficiently on Map-Reduce, a smart way is to ensure that we always choose core nodes as start points. Thus, after filtering, any node can be taken as a starting point since they are all core nodes.

Kingdom Expansion Game (KEG) is a strategy proposed in this thesis to parallelize DBSCAN algorithm. The principles for KEG are not complicated and can be described as follows:

There were some islands separated by sea. In some places on these islands, several powerful people became kings. They built their own kingdom and began to expand them. Year after year, dominions of these kingdoms are expanded larger and larger. Once two kingdoms are adjacent to each other, war will be fought between them. The kingdom with larger territory will absorb the smaller one. In the end, there was only one kingdom left on each island.

From the story, it is easy to figure out how the strategy will be like.

Assume we had a graph, after filtering, there were some islands on the graph. These islands do not have any connections between each other.

Firstly, some nodes on the graph are randomly chosen as starting point of expansion. Thus, each of these nodes represents a cluster in the beginning.

Expansion was done round by round. In each round, existing clusters do a unit length of expanding, which means that, adding all its connected neighbors into its own cluster if these neighbors haven't belonged to any cluster yet. If one of its neighbor already belonged to another cluster, it means that these two clusters are met each other, and they will be merged to one cluster.

In the end, on each island, there will be only one cluster.

One problem in this story is that, when starting points are chosen, it is not guaranteed all of islands have at least one starting point, so that some islands may not be expanded in the end. The solution is simple. In each round of expansion, system randomly chooses a group of new starting points from vertices which have not been expanded yet. Therefore, sooner or later, all of islands will get starting points. If number of vertices is X and y vertices are chosen as new starting points per round, by applying this solution, it is guaranteed that all of vertices will be expanded in no more than X/y iterations.

3.3.4 Algorithm Design

In the following description, the words "**Kingdom**" and "**Cluster**" have the same meaning.

Violating the original intention of Map-Reduce model, KEG needs to share data among slave machines. Map-Reduce model itself doesn't encourage sharing information while running map tasks or reduce tasks, but it doesn't mean that's impossible. HDFS and HBase [37] both provide certain ability of data sharing. Here we suppose each node in the graph is independently accessible. Data of each node includes node's ID, node's type, which cluster it belongs to, and its neighbors. In the beginning, nodes' types and nodes' belonging are empty.

Generally said, running a filter is to decide the type of nodes. Type 0 indicates the node is an edge node or a noise node, type 1 means this is a core node. Given a graph, filter can be done by processing nodes separately. Thus, it is good for parallelizing it on Map-Reduce.

Algorithm 4: filter()
Map-Reduce Input: given graph

Map-Reduce Output: graph without noise and edge nodes

```
class FilterMapper
    method map (nodeID, K, ε)
        node = readNode(nodeID)
        neighbors = node.neighbors
        for all neighbor ∈ neighbors do
            if distance (neighbor, node) > ε then delete neighbor
        if neighbors.size >= K then
            freeLand.append (node.ID)
            updateNode (nodeID, type = 1)
            updateNode (nodeID, neighbors)
        else clear neighbors
            updateNode (nodeID, type = 0)
            updateNode (nodeID, neighbors)
```

Algorithm 4 shows how to do filter on graph using Map-Reduce model. Mapper will read data of nodes one by one. For each node, mapper simply counts the number of its neighbors within distance ε, if the number is not smaller than a given threshold *K*, then this node will be included in the filtered graph and the ID of this node will be stored into a file called *freeLand*, otherwise, the node will be marked with type 0.

After filtering, all of nodes will have a type. Nodes with type 1 will have their neighbors within distance ε, and the ones with type 0 will loss all of their neighbors. Separated islands are built on the graph by nodes which are marked with type 1. It is the time to choose starting points, make clusters, and let the cluster expanding.

The file *freeLand* can be seen as a cluster which contains all of core nodes (type 1). What needed to do then, is selecting some nodes as starting points (kings) of the expansion from *freeLand.*

Algorithm 5: newKingdoms (numberOfKingdom)

```
newKingdoms (numberOfKingdom)
        borders = Ø
    do numberOfKingdom times
        capital = randomNode (freeLand)
        freeLand.remove (capital)
        borders.add (capital)
        createKingdom (capital)
        kingdom(capital).append (capital)
        updateNode(capital, belongTo = capital)
    return borders
```

Algorithm 5 is used to initiate expansion (This is not a Map-Reduce Job). The algorithm random picks numberOfKingdom nodes as the initial nodes of expansion out of file *freeLand*. For each of these nodes, a temporary cluster (kingdom) will be created. Each temporary cluster will have a corresponding file in file system. Each newly created cluster has only one element. The belonging of nodes will be updated. The rest part of *freeLand* is a file which records all IDs of the core nodes that haven't decided which kingdom they belong to.

Now initial kingdoms are built, the next step is to let the kingdoms do expansion.

As explained before, after filtering, the nodes in the graph now are all core nodes. Here, the definition of core node is refined to prepare for the next stage of expansion.

1. **BORDER:** In each Kingdom (Cluster), the newly expanded dominion (nodes) from last round is called border
2. **INLAND:** each border node of a kingdom will be processed once. After processing, this node is called inland, and no longer be a border node. Of course, all of the first node in a kingdom will be considered as border in the beginning.

The reason of distinguishing border and inland nodes is that, when doing expansion, the nodes which are already expanded will not be processed again. These nodes become inland. During each iterating, algorithm only does expansion on border nodes.

To make a full parallel algorithm, besides making multiple kingdoms expanding at same time, it also needs to make a single kingdom expanding parallel. The border node set is a very good input for a parallel expansion. Because sooner or later, each core node will be border node one time and only one time, every border node will be processed, and not rely on the processing result of other border nodes. It makes expansion easy to apply on Map-Reduce model.

Algorithm 6: Expansion()

```
Map-Reduce Input: all border nodes of all kingdoms
Map-Reduce Output: new border and cluster merging list
class ExpansionMapper
    method map (borderNodeID)
        borderNode = readNode (borderNodeID)
        neighbors = borderNode.neighbors
        for all neighborID ∈ neighbors do
            neighbor = read (neighborID)
            if neighbor.isFree then
                freeLand.remove (neighborID)
                updateNode (neighborID, belongTo = borderNode.belongTo)
                updateKingdom (borderNode.belongTo, append, neighborID)
                if neighbor.type = 1 then
                    collect (borderNode.belongTo, neighborID)
```

```
            else if neighbors.isOtherCluster & neighbor.type = 1 then
                collect (conflict, {borderNode.belongTo, neighbor.belongTo})
class ExpansionReducer
    method reduce(key, values)
        newBorderList= Ø
        conflictList= Ø
        if (key is not "conflict") then
            for all nodeID∈ values
                newBorderList.add (nodeID)
            removeDuplicate(newBorderList)
            collect newBorderList
        else conflictList. addAll (elements)
            removeDuplicate(conflictList)
            kingdomSets = group (conflictList)
            for all kingdomSet ∈ kingdomSets do
                kingdomIDLargest = max (kingdomSet)
                for all kingdomID ∈ kingdomSet except kingdomIDLargest do
                    kingdom = readKingdom(kingdomID)
                    for all nodeID ∈ kingdom do
                        updateNode (nodeID, belongTo = kingdomIDLargest)
                    updateKingdom(kingdomIDLargest, merge, kingdomID)
```

Algorithm 6 shows how to do expansion using given border nodes. In mapper, each border node will be processed. If its neighbor doesn't belong to any cluster, this neighbor will be included into the cluster which the border node belongs to, and will be a border node if it is a core node. If the neighbor belongs to another cluster, conflicting information will be passed to reducer. Besides recording all of border nodes for next round of expansion, the reducer gather all of conflicting information, group them to figure out which clusters should be merged, and then merge the clusters by rewriting *belongTo* value of each merged node.

The algorithm is done by iterations of newKingdoms() and expansion(). The reason of doing newKingdoms() multiple time is that, each island must have at least one kingdom on it to guarantee all of core nodes have been expanded in the end. If newKingdom() is run only one time, it is possible that there's no starting points on some islands. Thus, always selecting new starting points from freeLand after expansion() is necessary. The algorithm will terminate when no more border node and freeLand is empty.

The most important result we got from Algorithm 6 is merging list. The length of this won't be long. Assume we have 1000 clusters, half of them generate merging information, each of them gives 2 merging, and then the actual merging event numbers is 1000. So it won't take too much time process the information even in a much bigger graph.

3.3.5 Improvements

The algorithm proposed above have its own shortcomings.

1. In each round of expansion, a certain number of new kingdoms are created. But in a worst case, the number of kingdoms will grow larger and larger.
2. In mapper and reducer, there are lots of read and write operations to file system. This is not encouraged in designing of Map-Reduce algorithm.
3. Merge is done sequentially. Although deciding which kingdoms to merge is an easy job, but merging them on graph structure takes lots of time.

The concept "Partial Largest Node" (PLN) is introduced to limit the number of kingdoms. Assuming all of nodes are using numeric ID, then the PLN is defined as below:

1. It is a core node.
2. The ID of this node is the largest comparing to its neighbors.

Because all of IDs on one island are not duplicated, therefore, there will be only one largest ID in one island. Because nodes in one island are not connected to nodes in other islands, so the node with largest ID will only connected to nodes in the same island, and therefore, all of its neighbors' IDs are smaller than it's ID. Thus, on each island, there will be **at least one PLN**.

Finding out all of partial largest nodes on the graph and taking them as starting points will guarantee all of core nodes involved in expansion when algorithm have terminated.

Algorithm 7: PLN()

```
Map-Reduce Input: graph without noise and skirt nodes
Map-Reduce Output: partial largest nodes
class PLNMapper
    method map (nodeID)
        node = readNode(nodeID)
        neighbors = node.neighbors
        max = max (neighbors)
        while (neighbors is not empty) do
            if nodeID > max then
                collect nodeID
                break
            else if freeLand.notConetain (max) then neighbors.remove (max)
            else break
```

Algorithm 7 can be used after filter to generate PLN. It compares the node's ID and largest ID of its neighbors. If the node's ID is larger, then this node must be a PLN. Otherwise, it will check whether its neighbor with largest ID is a core node or not, if not, it will remove this largest neighbor and do the comparing again, until the largest neighbor is a core node or it is smaller than node's ID.

Algorithm 5 is no longer needed when using Algorithm 7. However, there's no conflict if using them together. When the quantity of PLN is small, Algorithm 5 can be used to help speeding up processing.

Being an alternative way of merging kingdoms on graph structure, Kingdom League is proposed instead of original kingdom merge mechanism. The basic idea of kingdom league is to make a list of alliance which contains all of kingdoms connected together, which means that, if A connects to B and B connects to C, then A, B and C will be in one league. While a kingdom is created, a corresponding file will be created to store the league information of this kingdom. Using kingdom league mechanism will increase number of files on file system, but it will significantly cut the amount of file operations. To use this mechanism, the algorithm 6.1 should be rewritten as below:

Algorithm 6.2 Expansion()

```
Map-Reduce Input: all border nodes of all kingdoms
Map-Reduce Output: new border and cluster merging list
class ExpansionMapper
    method map (borderNodeID)
                    ……
            else if neighbor.isOtherCluster & neighbor.type = 1 then
                league = readLeague (borderNode.belongTo)
                if ! league.contain (neighbor.belongTo) then
                    collect (conflict, {borderNode.belongTo, neighbor.belongTo})
class ExpansionReducer
    checkedKingdoms = Ø
    notCheckedKingdoms = Ø
    resultLeague = Ø
    conflictList = Ø
    method reduce(key, values)
            ……
        else conflictList. addAll (elements)
            removeDuplicate(conflictList)
            notCheckedKingdoms = conflictList.getAllkingdoms()
            returnKingdoms = Ø
            for all kingdomID ∈ notCheckedKingdoms do
                resultLeague.clear
                if kingdomID !∈ checkedKingdoms then
                    relatedKingdoms = check (kingdomID, returnKingdoms)
                for all kingdomID∈  resultLeague do
                    writeLeague(kingdomID, resultLeague)
    method check (kingdomID, returnKingdoms)
        if kingdomID !∈ checkedKingdoms then
            league = readLeague(kingdomID)
```

```
league = league ∩ notCheckedKingdoms
resultLeague = league ∪ resultLeague
checkedKingdoms.add(league)
toCheck = Ø
for all conflict ∈ conflictList do
    if conflict.kingdom1 ∈ league then toCheck.add (conflict.kingdom2)
    if conflict.kingdom2 ∈ league then toCheck.add (conflict.kingdom1)
for all toCheckKingdomID ∈ toCheck do
    returnKingdoms.add(check(toCheckKingdomID, returnKingdoms ))
return returnKingdoms
```

Mapper will read the league file to see whether the conflicting kingdoms belong to the same league or not, if yes, no conflict information will be passed to reducers.

3.3.6 Potential Simplification

According to Jean-Baptiste Chaubet's work [40], filter of the graph can be done without too much extra cost while computing similarity graph. What needed to do is adding a counter to each node, when similarity value is calculated between 2 nodes, the counter will plus 1 if the value is not smaller than a given threshold. When the value of counter achieved a given value K, the corresponding node will be marked as core node. Following his thoughts, PLN can also be found out while building similarity graph. In the beginning, all of nodes will be marked as PLN. While a node is decided to be a core node, this node will push this information together with its nodeID to its neighbors. All of nodes which received this information will check the nodeID in this information, if the nodeID in the information is larger than its own node ID, this node will cancel its PLN mark. When a node have computed all of the similarity values and is not a core node, this node will cancel its PLN mark. Therefore, after the similarity graph is built, all of the PLN nodes are known.

Chapter 4

Implementation Details

All of MCL proposals are implemented. For DBSCAN, only PLN strategy is applied. Some details of implementations are explained in this chapter. By comparing the different implementations of MCL, a best solution is found and taken into evaluation phase. Since the DBSCAN proposal must share data on file system, the correctness of data in the implementation is also discussed is this chapter.

4.1 Graph Data Structure

Similarity graphs are used as input of both MCL and DBSCAN. In our case, edges in graphs are bidirectional and weighted. The weight of each edge is no smaller than 0 and no more than 1 which can represent the similarity of different vertices in the graph.

Two kinds of structure of graph files are taken into consideration:

1. Single File: use one file to contain all of the information of the graph.
2. Multiple Files: one file only contains information of one vertex, thus, the number of file is equal to the number of vertices in the graph.

In single file, the graph is represented as:

<Vertex ID> <Edge 1 ID> <Edge 1 Weight> <Edge 2 ID> <Edge 2 Weight>… <Edge n ID> <Edge n Weight>.

But in multiple file format, the ID of vertex will be used as the file name, and in the file, the graph will be shown as:

<Edge 1 ID> <Edge 1 Weight> <Edge 2 ID> <Edge 2 Weight>… <Edge n ID> <Edge n Weight>.

The benefit of using a single file is that, it is the best way of Hadoop to process, because Hadoop is optimized to process small number of large files, but not large number of small files. Using one file per vertex is good for the individual accessibility of vertices, but HDFS will have difficulty to solve too many files. As input where individual accessibility is not that important, using single file is a better choice, although it may increase the complexity of file processing.

4.2 MCL

4.2.1 Intermediate Result

A customized splitter is made for all of MCL implementations to store the files by block. For example, a sub-block is described as following:

<Column From, Column To>

<Column ID> <Row ID 1> <Value 1> <Row ID 2> <Value 2>… <Row ID n> <Value n> *
<Row From, Row To>
<Row ID> <Column ID 1> <Value 1> <Column ID 2> <Value 2>… <Column ID n> <Value n>*
<End>

or

<Sub-Block ID>
<Column> <Row> <Value> *
<End>

When the splitter read the word "End", it means a block have finished. Therefore, multiple blocks can be stored into one file.

The drawback of this format is that, all of values are stored in plain text. For a Double type value which is 64 bits, it should only take 8 bytes in Sequential File Format, but in a Text File Format, it may be shown as 4.94065645841246544*E*-324 which is obviously much larger than 8 bytes. Thus the network volume is far higher than the ideal amount.

The algorithm is implemented as a job chain of Expansions and Inflations. One job will be run in the beginning to produce the probability matrix. After this step has been done, Expansion and Inflation will be applied alternately until termination. Intermediate result will be stored using block format on HDFS and be taken as the input for the next job.

4.2.2 Termination

There are 2 ways to decide the termination of algorithm:

1. When all of elements in the matrix are not changed.
2. When the number of output records in reduce task is the number of vertices.

The first way is the most accuracy method to decide when MCL should terminate. However, comparing two huge matrices will take a lot of time. In the beginning, the elements in the intermediate results will change quite often. It is not hard to compare 2 matrices, because once a changed element is found, the 2 matrices will be considered different. But when the algorithm is going to terminate, most elements in the matrix is fixed. To decide the matrix is changed or not, almost the whole matrices will be read and compared.

The second way is a pretty smart way to decide when to terminate. It borrows the build-in counter feature of Hadoop, and doesn't cause any extra work load. However, this method can only be used in the situation that all of elements will definitely converge to value 1. Because, if the algorithm converge with two or more non-1 value, the number of output value will be larger than the number of vertices.

4.2.3 Comparison

By testing the 3 strategies using the same graph and same parameters, it is found that the 3rd strategy is obviously superior comparing to former 2 in term of execution time.

No matter how the MCL is designed, if the graph and parameters are keeping unchanged, the iterations to terminate will be fixed. Therefore, the execution time of single job will decide the total time.

In Strategy 1, to compute a sub-matrix from column 0 to 10 and from row 0 to 10, all of these columns and rows are needed. The solution is to store all of them into a file block, and take the block as the input of expansion. This costs too much time on data transmitting and writing back to file system. According to our experiments, in a 10000 nodes graph, if matrix is divided into 100 sub-matrices, the total size of data could be higher than 200GB. Thus, the performance is not ideal.

In Strategy 2, the total data amount decreased significantly. However, the number of map tasks and reduce tasks in expansion is equal to the number of sub-matrices to the power of 1.5. If one column is divided into N vectors, the number of sub-matrices is N*N, and N*N*N tasks will be executed both in map and reduce phase. This number is growing exponentially, when number of vertices has a linear growing. The overhead caused by the Hadoop itself is much longer comparing to calculation time.

In Strategy 3, the data amount is same to Strategy 2, but number of tasks is reduced to N, thus, it is significantly faster comparing to Strategy 2.

After the brief comparison, the implementation of Strategy 3 is chosen to be evaluated.

4.3 DBSCAN

4.3.1 Intermediate Result

After filtering, a new graph is stored using the multiple files format. All of the IDs of PLNs are put into a file as the input of expansion. The output of each expansion job is IDs of vertices which will be taken into next iteration of expansion.

4.3.2 Correctness of Data Sharing

Because data sharing is involved into the algorithm design, the correctness is a critical part of the algorithm.

Different machines may read and write the same file at different time during one job. This may cause an inconsistency of data which those machines got. It needs to be guaranteed that, although the inconsistency does exist, the correctness of results will not be affected.

HDFS is simpler comparing to GFS [27] on file writing. It doesn't support any parallel writing to the same file, and doesn't support appending to an exist file. HDFS will only lock the chunks involved into write operations, it is sufficient for usual I/O operations of Hadoop.

One vertex may be read and written multiple times during one job. For example, if a vertex is neighbors of 2 PLNs, then both of these 2 PLNs will read the file of this vertex. The operation sequence could be:

1. R1R2W1W2 or R1R2W2W1: both PLNs read the file, found it is not belong to any cluster and decide to change the belonging of this vertex. The final result is given by the last write operation.
2. R1W1R2: one PLN read the file first and changed the belonging of vertex, then, another PLN read the file and found the file have been changed, and it will not change the file again.

In both cases above, the vertex belongs to either a PLN which it density-connected to, thus, the result is still correct if multiple machines want to do read and write operations on same file.

4.3.3 Termination

DBSCAN will terminate when the graph is traversed. In algorithm proposed in this thesis, traversal is done when no new nodes are taken into next step of expansion. Thus, the build-in counter can help to decide when the algorithm should terminate.

4.3.4 Problems

While testing the code of DBSCAN, some I/O Exceptions happened occasionally when running on large graph. These exceptions are caused by overload of I/O operations on file system, because Hadoop is not optimized for large quantity of file operations. Thus, it is not avoidable if only use Hadoop to process vertices files.

Chapter 5

Evaluation

Not all of the implementations will be evaluated because of the time limitation. Bad strategies are discarded by short tests. Only strategies with the best performance will be taken into full evaluation phase.

5.1 Evaluation Environment

All of evaluations will be done on cluster which is hosted in Salesforce, located in San Francisco. The cluster consists of machines with Intel Xeon Processor W3680 (12M Cache, 3.33 GHz, 6 cores, 64-bits), 12 GB memory and 500GB hard disk drive. Operating systems used on these machines are Ubuntu Release 9.10, Kernel Linux 2.6.31-22-generic. All of these machines are linked via 1000M LAN.

Cloudera's Distribution for Apache Hadoop (CDH) is installed on all of these machines. The version of Hadoop is 0.20.2. One machine is taken as Jobtracker and Namenode, and others will be run as slave machines which will play roles of Tasktrackers and Datanodes. Each slave machine will have 2 map task slots and 2 reduce task slots. It means that, in maximum, 2 map tasks and 2 reduce tasks can be executed on a slave machine simultaneously.

Evaluations will be done on two different kinds of graphs:

1. Power Law Graph (PLG): the number of edges is varying as a power of number vertices which have this number of edges, and hence follows a power law. This kind of graph is called Power Law Graph
2. Uniform Random Graph (URG): when all of the vertices have number of edges around the average value, this kind of graph is called Uniform Random Graph.

The graphs used in evaluations will be generated randomly. The generator of URG is self-made, and the one for PLG is adapted from Owen Densmore's work [38] with his permission.

5.2 Goals of Evaluation

By evaluating MCL and DBSCAN on PLG and URG, the following goals are expected to be achieved:

1. Scalability of these algorithms performing on Hadoop platform is needed to be evaluated. It is necessary to see how the number of cluster machines influences the performance of these algorithms with a certain amount of work.
2. It is good to know the increment pattern of execution time when graph size is growing. Potential pattern could be linear, exponential or others.
3. Another goal is to evaluate the variety of performance influenced by density of graph. Same algorithm may show different characters on sparse and dense graphs.

5.3 Influence by Number of Machines

The size of graphs used in this evaluation is fixed. One PLG and one URG are generated for evaluations. Each graph has 1000 vertices and 4 edges per vertex in average. Each job will have 10 map tasks and 10 reduce tasks. When performing MCL algorithm, the inflation coefficient is always 1.5. Because we are using similarity graphs, the threshold of DBSCAN are set to when ε is no smaller than 0.5. K is set to 3. Same group of tests are executed on clusters with different number of slave nodes. The results are depicted in Figure 5.1 and 5.2:

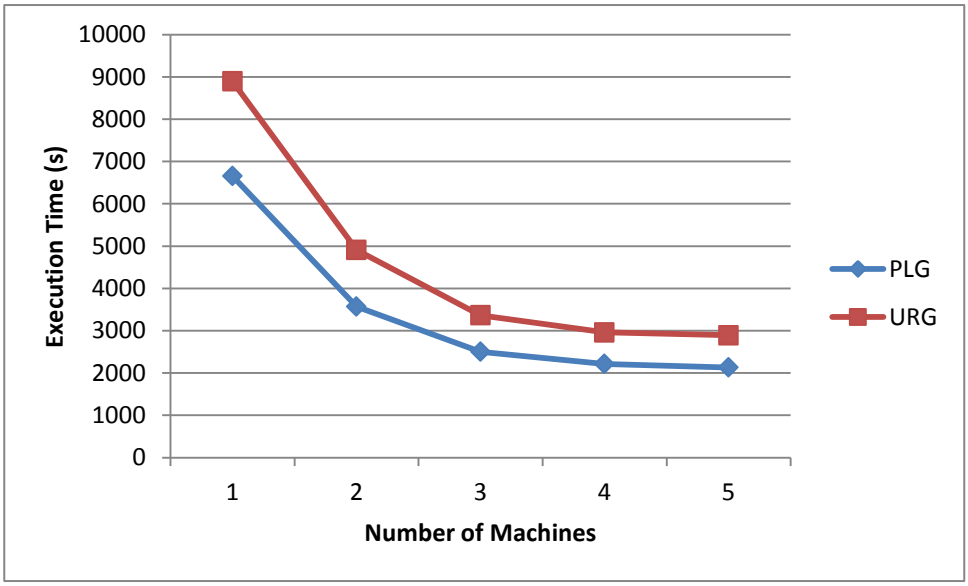


Figure 5.1: Performance Influenced by Number of Machines (MCL)

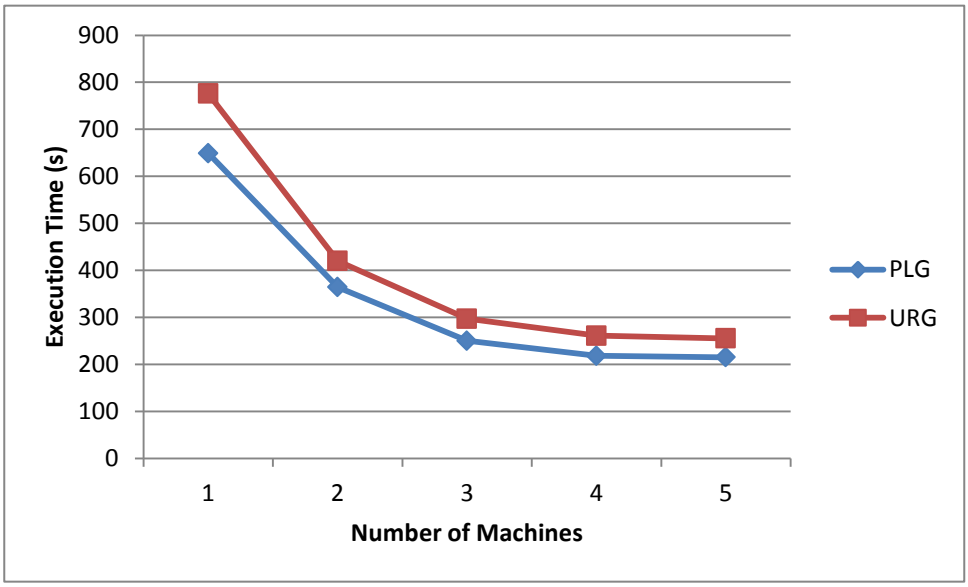


Figure 5.2: Performance Influenced by Number of Machines (DBSCAN)

Both MCL and DBSCAN show good scalability on Hadoop clusters. The execution time dropped significantly when running on 2 machines comparing to that running on only one machine, the value is almost halved while increasing the number of machines from one to two. While the number of machines growing, the decrease amount of execution time gets smaller. This is easy to understand. Theoretically said, the decrease of work amount will be 1/(n-1)-1/n (n is number of machines). It very well explained why the decrease amount is dropped. However, when n growing larger and larger, the value of 1/(n-1)-1/n approaches 0. In the graphs shown above, the curve almost converged when there are 5 machines, and of course the value is not 0. The reason causing this difference between theoretical and practical results is the number of tasks. Assume we have 10 tasks in a job, each machine can run 1 task at a time. To finish the job on one slave machine, it will run all of these 10 tasks. When there are 2 slave machines in the cluster, each machine only need to execute 5 tasks. In like manner, only 1 task per machine is needed to run if there are 10 machines. If the number of slave machines is 20, 10 of them will be chosen to run 1 task each. Comparing to the condition with 10 slave machines in the cluster, there will be no improvement in performance. Thus, no matter how many slave machines in the cluster, the lower limit of execution time is how long a slave machine can finish one task.

In the evaluation cases, each slave machine have 2 slots, 5 slave machines can run 10 tasks simultaneously. The speed up should not be significant while it has more than 5 slave machines.

5.4 Influence by Number of Vertices

To see the performance influenced by number of vertices, the size of graph is increased linearly, but the number of slave machines and average number of edges should be kept fixed.

Graphs in sizes from 1000 to 10000 are generated for evaluations, 1000 as a step. All of graphs have 4 edges per vertex in average. For each size of graph, 10 different graphs will be randomly generated. Thus, 200 graphs will be used in evaluation in total. All of jobs will be run on a cluster with 5 slave nodes. The size of each task will grow linearly, so each job will have 10 map tasks and 10 reduce tasks. All other parameters and factors are remained the same as previous evaluations.

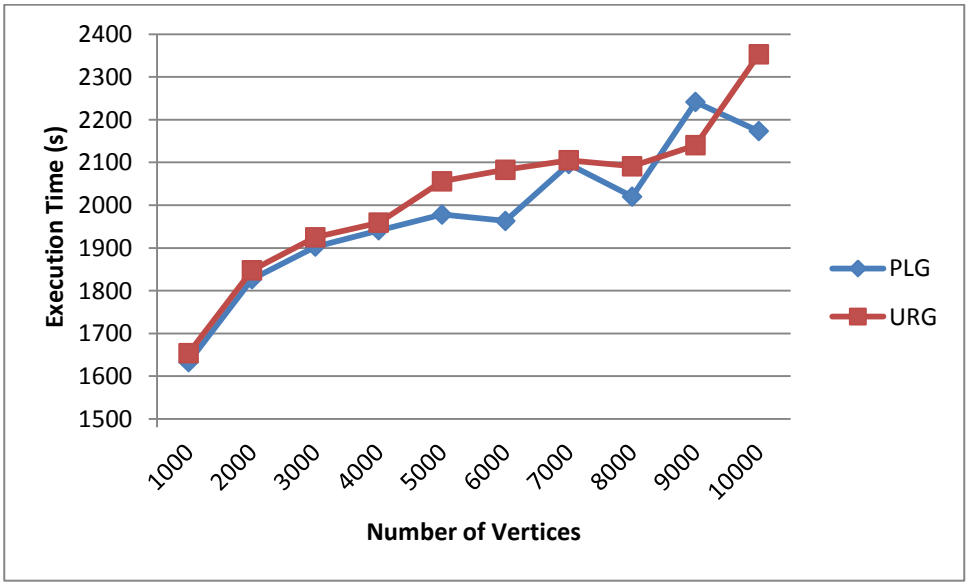


Figure 5.3: Performance influenced by Number of Vertices (MCL)

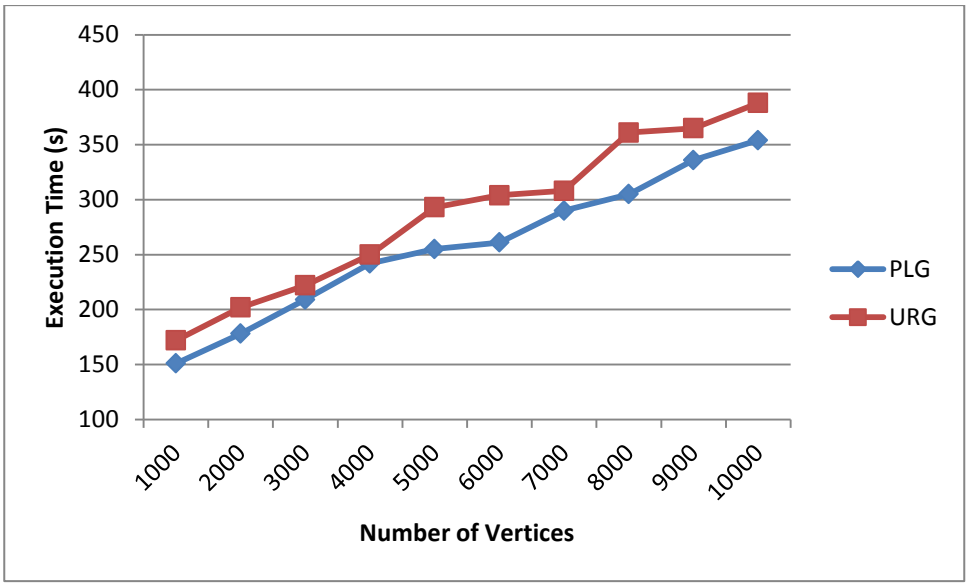


Figure 5.4: Performance influenced by Number of Vertices (DBSCAN)

The results can be seen from the Figure 5.3 and 5.4. In each job, there are always 10 map tasks and 10 reduce tasks. While the size of graph is growing, the number of vertices each slave machine should process is also growing. Because of the time limitation, it is unable to get large quantity of sample data, thus, the lines in charts are not smooth enough. But it is clear that the execution time of both MCL and DBSCAN are increasing almost linearly. Both MCL and DBSCAN have better performance when processing PLG. When the graph size is 1000, MCL will take 1600 to 1700 seconds to terminate, while DBSCAN only takes 150 to 180 seconds.

However, since MCL algorithm involved problem of memory consumption, it is not realistic to always grow the size of each task in MCL. An alternative way is to fix the size of each task in MCL, and make the number of tasks growing. To keep the task size of MCL unchanged, the size of sub-matrix should be keep fixed. In the following evaluation, the size of sub-matrix is fixed to 100, which means that, the number of tasks in each job will grow linearly from 20 to 200. This evaluation will only be done on PLG.

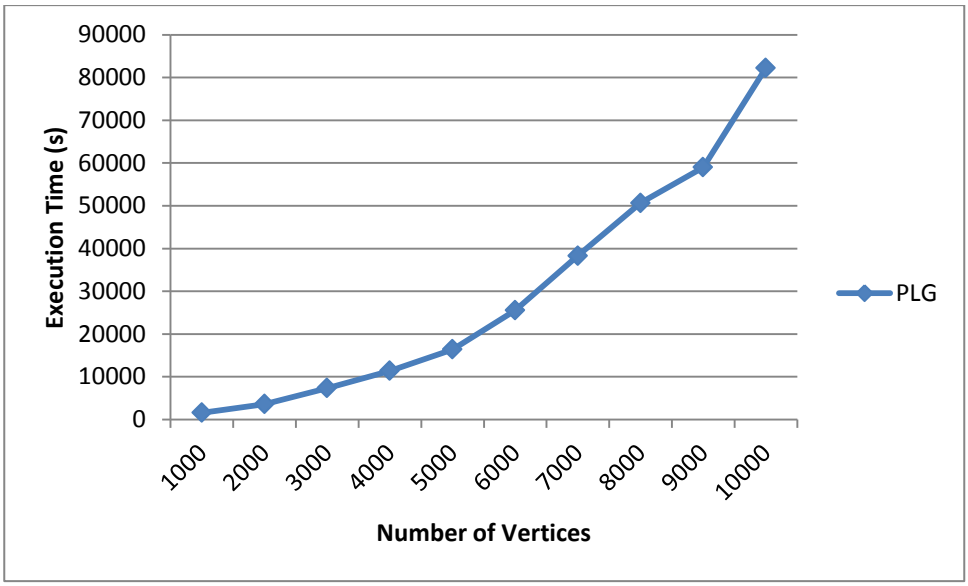


Figure 5.5: Performance influenced by Number of Tasks (MCL)

Figure 5.5 shows the result when size of graph is growing and size of sub-matrix in MCL is fixed. The curve shows an exponential trend of growing.

To further finding out the relationship between fixed number of tasks and fixed size of sub-matrix, 2 sets of data are taken into analysis. One PLG per graph size is chosen, and then acquire the execution time when the number of tasks are fixed and not fixed. By analyzing the data, we got the following result:

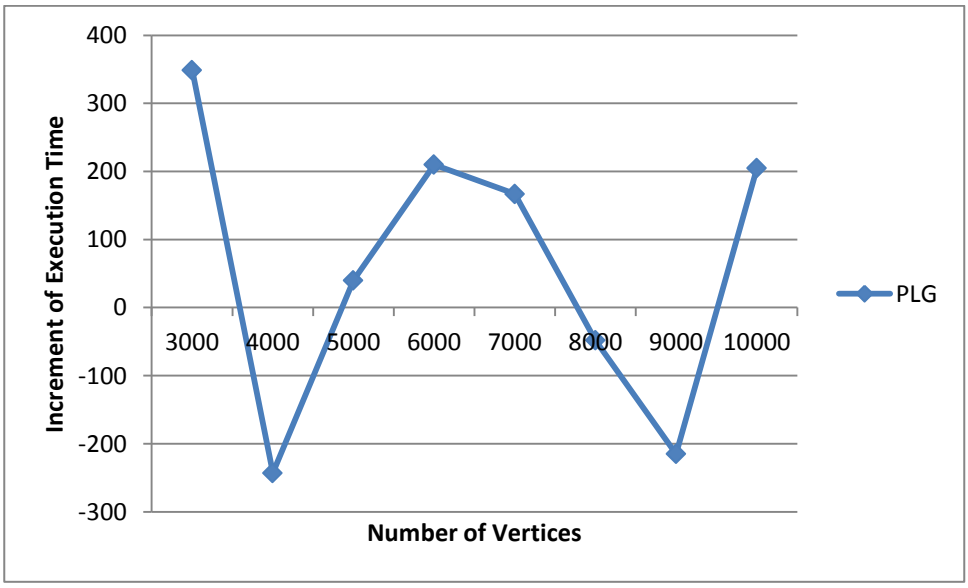


Figure 5.6: Increment of Execution Time (Number of Tasks Fixed)

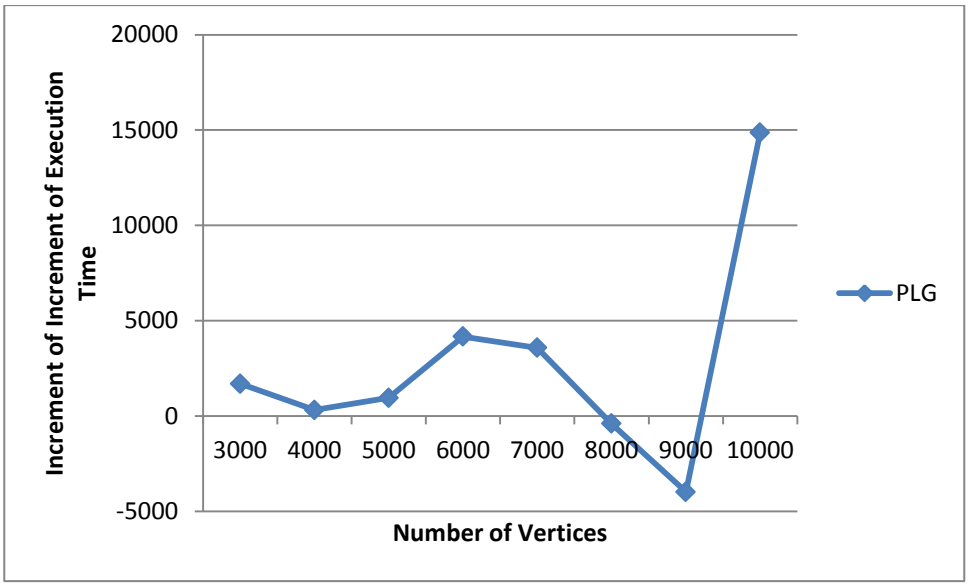


Figure 5.7: Increment of Increment of Execution Time (Size of Sub-Matrix Fixed)

The 2 fold lines in the graphs above show the same trend. But the y axis of former one is increment of execution time, and for the later one, we calculate the increment of execution time first, and then calculate the increment of the new values, thus, it is increment of increment of execution time. It is a good evidence to say that, when size of sub-matrix is fixed, the curve of execution time is one dimension higher (e.g. x^3 is one dimension higher than x^2) than the one when number of task is fixed. However, to further explain this phenomenon, more experiments are needed.

Theoretically said, the execution time would grow exponentially when size of graph growing linearly. However, when running on Hadoop, the time spent on data reading, writing and transmitting is much higher comparing to pure calculation. Thus, to a large extent, the evaluation result shows the increment of data amount.

When number of edges per vertex is fixed, the data amount of a sparse graph is equal proportional to the number of vertices. The duplication of graph data while calculating is 2*n-1 times (n is number of sub-matrices in one dimension). Therefore, if the number or sub-matrices is fixed (number of task is fixed), the data amount will be equal proportional to the size of graph. Otherwise, if the size of sub-matrices is fixed, the data amount will grow in a quadratic manner.

5.5 Influence by Number of Edges

Density is a vital character of graph, so finding out how does the algorithm perform on graphs with various size of edges is important. In the graphs used in evaluation, the number of edges in average is increased from 2 to 20, 2 edges as a step, and the number of slave nodes and number of vertices in a graph will not change and stay at 5 nodes and 10000 vertices.

10 different graphs will be randomly generated for each size of edges. Therefore, 200 graphs will be generated and used in evaluation. One job will be divided into 10 map tasks and 10 reduce tasks while running on Hadoop. The inflation coefficient r is still 1.5. The ε remains 0.5. The value K is the same as number of edges in average when performing on PLG and it is set to 0 when running on

URG. The reason of configuring the K value in various values is to guarantee a fixed ratio of core nodes in a graph.

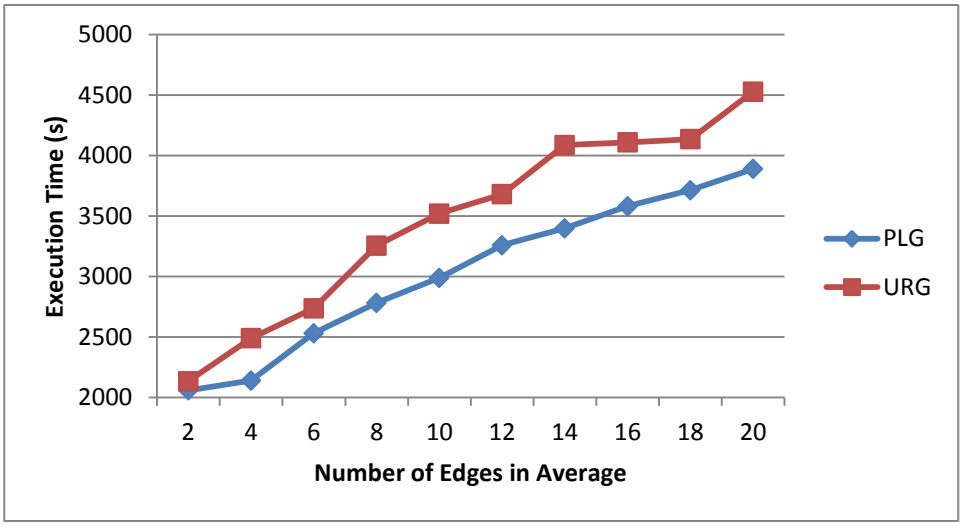


Figure 5.8: Performance influenced by Number of Edges (MCL)

Figure 5.8 shows the result of MCL algorithm. It can be seen clearly that the execution time is growing linearly when number of edges of each vertex increased. From former analysis, it is known that the execution time of MCL is equal proportional to the amount of data. Since the data amount in a sparse graph is *v***e* (*v* is number of vertices, *e* is number of edges per vertex in average), when the size of edges growing linearly, the execution time will go up in a same percentage.

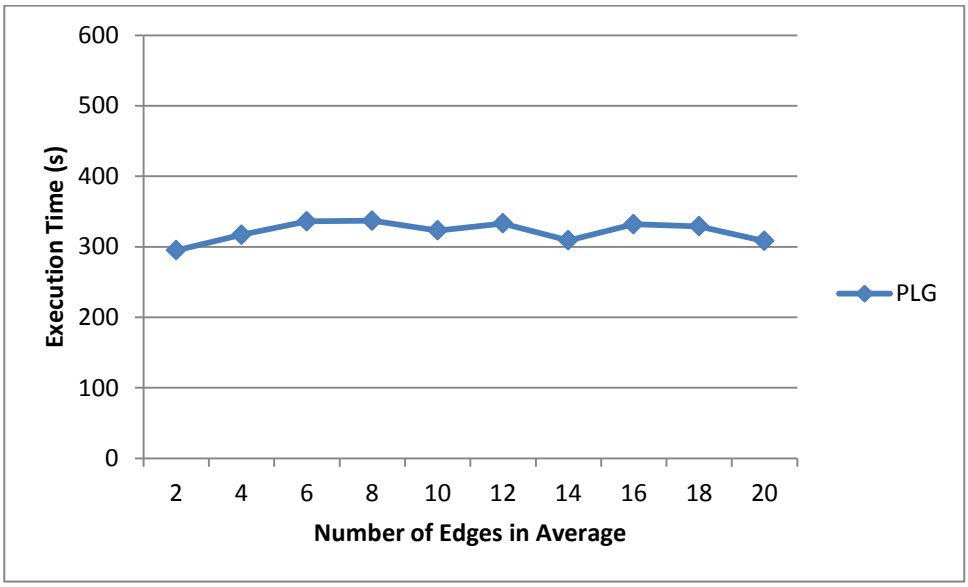


Figure 5.9: Performance influenced by Number of Edges (DBSCAN, PLG)

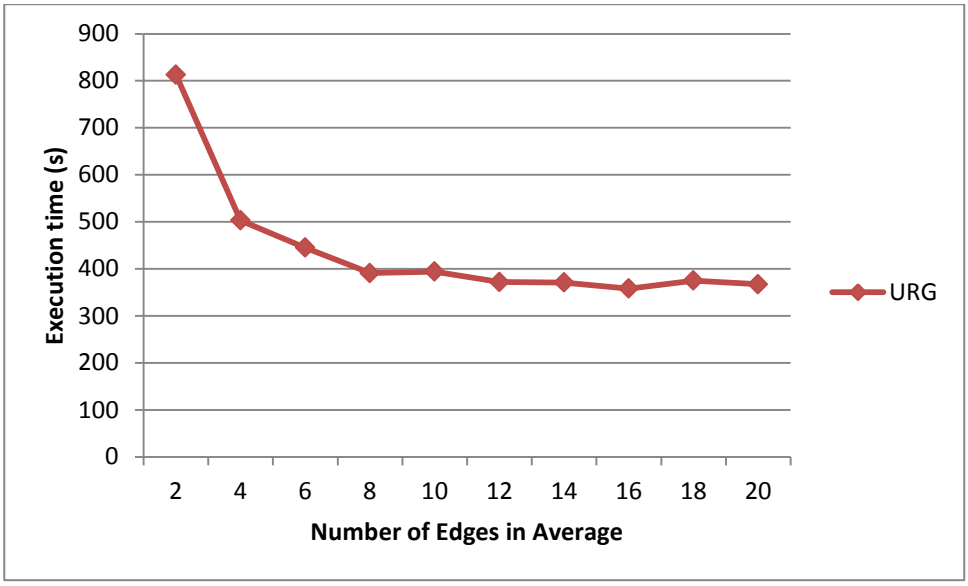


Figure 5.10: Performance influenced by Number of Edges (DBSCAN, URG)

The performance of DBSCAN is obviously different comparing to MCL in this evaluation.

In the first group of evaluations, parameters are controlled to get an unchanged ratio of core nodes in PLG. Assume the distribution of edges follows the equation $y = ax^k$, the ratio of core nodes in PLG is $\sqrt[k]{\frac{A-b}{a}}$ where A is the number of edges in average. And because of $\int_0^{10000} ax^k = 10000A$, the ratio equals to $\sqrt[k]{\frac{1}{k+1}}$. Thus, when k is a constant value, the ratio will be fixed. The performance is shown in Figure 5.9, the influence from number of edges can hardly be seen. No matter how many edges the graph had, the algorithm always terminated at 300 to 350 seconds.

All of vertices are chosen to be core nodes in the second group of evaluations. Since the parameters setting are not consistent with the first group, the results between 2 groups are not comparable. As shown in Figure 5.10, the execution time dropped significantly in the beginning then became stable when number of edges grew larger. When each node has more than 10 edges in average in URG, the curve becomes flat. The algorithm would take 350 to 400 seconds to terminate.

Because DBSCAN requires large amount of write operating which is the most time consuming part during executing, the performance is related to how many times it is going to write to file system. In the DBSCAN algorithm proposed in this thesis, most of vertices will be written only once. The number of league files, which influence the amount of write operations, is predictable. In a URG with 10000 vertices, the possibility of each vertex to be a PLN is:

$$p = \begin{cases} \frac{C_n^e}{C_{9999}^e} = \frac{P_n^e}{P_{9999}^e} (e \le n) \\ 0 \ (e > n) \end{cases}$$

The e is number of edges in average and n indicates the ID of this vertex (from 0 to 9999). Thus, the expected value of PLN size is $\sum_{i=e}^{9999} \frac{P_i^e}{P_{9999}^e}$ for a certain value of e. The graph of expected number of PLN in different edge sizes is shown below:

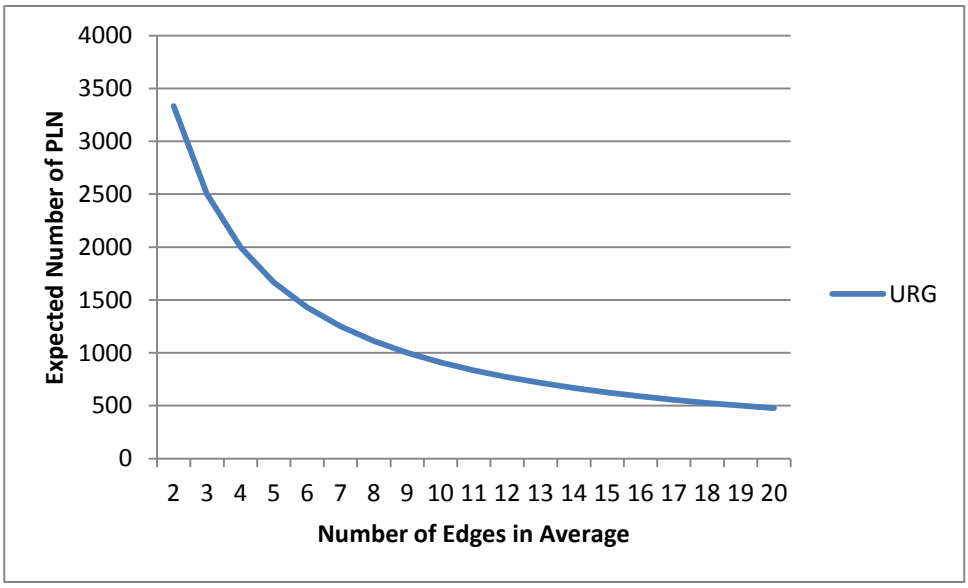


Figure 5.11: Expected Number of PLN influenced by Number of Edges (DBSCAN)

A similar curve can be observed in Figure 5.11 comparing to the one in Figure 5.10. It properly explained why the execution time dropped significantly in the beginning using DBSCAN algorithm proposed in this thesis when number of edges increased. In the implementation of algorithm, number of PLN equals to number of league files. The number of write operations can be seen as a constant value *C* (write on each vertex) plus a variable V (write on league files). Then the figure of execution time will be just looked like the one in Figure 5.10.

From the observations, the number of PLN in first group of evaluations (PLG) is controlled between 400 to 900 vertices. This will not make a huge difference on number of write operations. In other word, it won't influence the performance very much.

5.6 Summary

From the results above, a conclusion can be made. MCL performs faster on PowerLaw Graphs than Uniform Random Graphs. However the performance of DBSCAN will be influenced by the parameters of the algorithm, it's unable to say DBSCAN is executing faster on one kind of graph comparing to the other one. MCL will take more time to finish the calculation if the number of edges or number of vertices is growing. The execution time of DBSCAN is affected by size of vertices and number of PLN.

While executing a Hadoop job, more machines doesn't mean faster if there are not enough tasks for these machines. Each Hadoop job has a bottom border of execution time which is actually the execution time of 1map task plus 1 reduce task. If the problem size of each task remains the same,

the execution time won't dropped below this bottom value, no matter how many slave machines there are.

There is no directly comparable data between the MCL proposed in this thesis and the algorithms implemented on the PEGASUS, although they are both matrix computation related. However, some estimated values can be given, so that the audience can have a rough view on their performance.

On a 9 machines cluster, the PEGASUS takes 100 seconds to complete one iteration of PageRank on a graph with 59k vertices and 282m edges. On our machine cluster with 5 nodes, it takes 43 seconds to finish the matrix multiplication based on the graph with 10k vertices and 50k edges. The calculation amount of later one approximately equals to 50k vertices and 100m edges when doing PageRank.

The quality of clustering result of MCL and DBSCAN are controlled by the parameters they used. Their results can be similar when using proper values as parameters. In [40], Jean-Baptiste has done a detailed comparison of clustering result between MCL and DBSCAN.

Chapter 6

Conclusion

While social networks getting popular, the graph computation became essential on maintaining and processing the information from the social network graphs, and discovering potential features of the graph. When the number of social network users growing rapidly, the size of graph also grew larger and larger. It is impossible now to process a huge graph on a single machine in a "real time" level execution time. What we can do is to parallelize them and distribute them. Cloud computing provides a good foundation on fast graph calculation. Lots of works have been done on how to manage multiple machines processing graph cooperatively and improved ability of huge graph processing significantly. Some of them try to use Map-Reduce model to solve graph problems, and give out constructive suggestions on graph-based algorithm design on Map-Reduce model and lots of algorithms can be represented using those methods (i.e. matrix, BSP-like, etc.). Although a great progress have been made, it is only a start of research on graph-related algorithms based on Map-Reduce.

6.1 Summary of Work

In this thesis, a review of parallel and distributed graph processing techniques is done to summarize the works have been carried out, further finding out which kind of graph problems those techniques can solve.

Two algorithms, MCL and DBSCAN, are chosen to be redesigned, implemented and evaluated using Map-Reduce model. The methods proposed in this thesis also can be applied to some other algorithms. The results are analyzed to find out the scalability and performance influenced by various parameters. By the literature study and analysis of data, the adoptability of graph algorithms on Map-Reduce model can be found.

6.2 Research Findings

By reviewing related works of graph calculation, it is found that, most of graph-related algorithms can be categorized into Vertex-Oriented and Edge-Oriented. Edge-Oriented Algorithms are superior in numbers and frequently used. Vertex-Oriented problems are perfect on Map-Reduce model. Edge-Oriented ones are usually represented using BSP model and Matrices.

Both of BSP and Matrices can be simulated above Map-Reduce model, but in practice (i.e. Hadoop), the overhead caused by framework will influence the overall performance, especially when calculation time for each task is very short. The volume of network traffic is another factor which will affect the execution time of algorithms. Because message sending on network always takes more time comparing to processing time.

When using Map-Reduce to simulate BSP model, one Map-Reduce job can be seen as a superstep. The shuffle and fetch options can be seen as communication step of BSP model. Different to BSP, Map-Reduce model can do further analysis in reducers. The message routing is usually done by key and partitioner design.

No matter using BSP or Matrix to represent graph-based algorithms on Map-Reduce model, the algorithm logic is easy to described using map and reduce functions. However, the difficult part is, using the correct keys to route the right values to right reducers. Good routing can improve the time performance of algorithms.

While redesigning the MCL, several matrices multiplication methods are proposed which can be used universally. Although the MCL designed on Hadoop got a reasonable execution time, it is still too long to take MCL as a "real time" clustering algorithm.

Filter is used in reformulating and simplifying the DBSCAN algorithm. After filtering, the problem is changed into a connectivity problem. Then, random selecting nodes and Partial Largest Nodes are proposed as starting points of graph traversal. The DBSCAN proposed in this thesis needs a certain extent of data sharing. Too much read and write operations will cause the overloading of NameNode, thus, another sharing mechanism like Hbase is needed. Based on related work and practice experiments, some critical conclusions are made about graph-based algorithms design on Map-Reduce model:

1. Most of graph-related algorithms need to be represented using multiple Map-Reduce jobs. Do as much work as possible in a single Map-Reduce job, which means have as fewer jobs (iterations) as possible. Since the pure calculation time is short comparing to data transmitting and I/O operations, too many jobs will badly influence the performance.
2. Graph-related algorithms usually pre-load the sub-graph or sub-matrix into memory, thus, each task in a job usually can't be too large. The data structure which used to represent the sub-graph or sub-matrix is important. It directly decides the granularity of tasks. Control the size of each task in a job. Bad load balance will decrease the performance. Besides that, too many small tasks in a job also have bad influence on execution time.
3. For the algorithms with multiple iterations (e.g. MCL, PageRank), data locality, network traffic and number of key-value pairs are the most important elements to be considered.

6.3 Future Work

The work have been done in this thesis is a good beginning of research on how to map-reduce graph-based algorithm. Since the Hadoop platform have been built in many organizations and used by more and more researchers, new achievements will keep coming to the fore. Even though our research is limited by the time and environment, some meaningful results have been got. This thesis only contains the finished parts of the research. The ongoing researches and future research plan will be listed below:

- Since BSP model can solve most of Vertex-Oriented Algorithms, simulating a BSP model above Map-Reduce model is a wise way on algorithm design. A simple prototype of BSP

model has already been made during the research of this thesis. It works as a layer above Hadoop, and gives high flexibility on algorithm design. Further tests on this prototype are needed.

- HBase can be deployed on Hadoop cluster. It is interesting to see how HBase will influence the I/O ability of the DBSCAN proposed in the thesis.

- A new DBSCAN proposal is implemented and going to be tested. It works without individual vertex accessibility, thus, it is perfect for the flat data model of Map-Reduce. The algorithm splits the graph into several sub-graphs and do message exchanging among the sub-graphs. This DBSCAN will terminate within log(DB) iterations in worst case, where DB is diameter of file blocks.

- The data format in the MCL implementation is not the most optimized. The performance can be improved by customize a file format and file splitting method.

- To put the result into production, only processing the graph is not sufficient. Graph will be updated timely. Then, how to update the result of graph processing effectively is another subject that needs to be put into research.

References

[1] Geoffrey A. Fowler and Ben Worthen. The Internet Industry Is on a Cloud -- Whatever That May Mean. The Wall Street Journal, Mar. 2009.

[2] Greg Boss, Padma Malladi, Dennis Quan, Linda Legregni and Harold Hall. Cloud Computing. High Performance On Demand Solutions, IBM, Oct. 2007.

[3] Eric Knorr and Galen Gruman. What cloud computing really means. InfoWorld. http://www.infoworld.com/d/cloud-computing/what-cloud-computing-really-means-031?page=0,0, retrieved in Jan. 2011.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI'04: Proceeding of the 6th conference on Symposium on Operating Systems Design & Implementation, Dec. 2004, pages 137-150.

[5] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware, 2005

[6] Berge Claude. Théorie des graphes et ses applications. Dunod, Paris, 1958.

[7] Gallager, R.G., Humblet, P.A., and Spira, P.M. A Distributed Algorithm for Minimum-Weight Spanning Trees. In ACM Trans. Program. Lang. Syst., Vol. 5, Jan. 1983, pages 66-77.

[8] MM Flood. The traveling-salesman problem. In Operations Research, Vol. 4, 1956.

[9] Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. In Commentarii academiae scientiarum Petropolitanae 8, 1741, pages 128-140.

[10] Gianmarco De Francisci Morales, Claudio Lucchese and Ranieri Baraglia . Scaling Out All Pairs Similarity Search with MapReduce. In LSDS-IR'10, 2010.

[11] Vernica, R., Carey M.J. and Li C. Efficient Parallel Set-Similarity Joins Using MapReduce. In SIGMOD, 2010.

[12] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In WWW, 2008, pages 131-140.

[13] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with MapReduce. In HLT'08: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies, 2008, pages 265-268.

[14] Page Lawrence, Brin Sergey, Motwani Rajeev and Winograd Terry. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report. Stanford InfoLab, 1999.

[15] Gao Lianxiong, Wu Jianping and Rui Liu. Key nodes mining in transport networks based in PageRank algorithm. Sch. of Comput., Beijing Univ. of Posts & Telecommun. In Control and Decision Conference, 2009.

[16] S. Van Dongen. A new cluster algorithm for graphs. Report No. INS-R0010, Center for Mathematics and Computer Science, Amsterdam, 2000.

[17] A. J. Enright, S. Van Dongen, and C. A. Ouzounis. An efficient algorithm for large-scale detection of protein families. In Nucleic Acids Res 30, 2002, pages 1575-1584.

[18] Martin Ester, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), 1996, pages 226-231.

[19] Rishan Chen, Xuetian Weng, Bingsheng He and Mao Yang. Large Graph Processing in the Cloud. In SIGMOD, 2010.

[20] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In SIGMOD, 2010.

[21] Jimmy Lin, Michael Schatz. Design patterns for efficient graph algorithms in MapReduce. In MLG, 2010.

[22] Leslie G. Valiant. A bridging model for parallel computation. In Communications of the ACM, Volume 33 Issue 8, Aug. 1990.

[23] Albert Chan and Frank Dehne. CGMGRAPH/CGMLIB: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. In Intl. J. of High Performance Computing Applications 19(1), 2005, pages 81-97.

[24] Martin Beran. Computational Power of BSP Computers. In Proceedings of SOFSEM'98, 1998.

[25] Martin Beran. Decomposable Bulk Synchronous Parallel Computers. In Proceedings of SOFSEM'99, 1999.

[26] HDFS. The Apache Software Foundation. http://hadoop.apache.org/hdfs/, retrieved in Jan. 2011.

[27] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. The Google file system. In SOSP'03, 2003.

[28] U. Kang, Charalampos E. Tsourakakis and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System. In Ninth IEEE International Conference on Data Mining, 2009.

[29] Wei Xue, JuWei Shi, Bo Yang, X-RIME: Cloud-Based Large Scale Social Network Analysis, In IEEE International Conference on Services Computing, 2010.

[30] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In IEEE 26th International Conference on Data Engineering (ICDE 2010), 2010, pages 996-1005.

[31] Charalampos E. Tsourakakis. Data Mining with MAPREDUCE: Graph and Tensor Algorithms with Applications. Machine Learning Department, School of Computer Science, Carnegie Mellon University, Mar. 2010.

[32] Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim and Seungryoul Maeng, HAMA: An Efficient Matrix Computation with the MapReduce Framework. In IEEE Second International Conference on Cloud Computing Technology and Science, Jul. 2010, pages 721-726.

[33] A.A. Markov. Extension of the limit theorems of probability theory to a sum of variables connected in a chain. Reprinted in Appendix B of: R. Howard. Dynamic Probabilistic Systems, volume 1: Markov Chains. John Wiley and Sons, 1971.

[34] John Norstad. A MapReduce Algorithm for Matrix Multiplication. Northwestern University, Dec. 2009. http://homepage.mac.com/j.norstad/matrix-multiply/index.html, retrieved in Jan. 2011.

[35] Aoying Zhou, Shuigeng Zhou, Jing Cao, Ye Fan and Yunfa Hu. Approaches for Scaling DBSCAN Algorithm to Large Spatial Databases. Department of Computer Science, Fudan University, Shanghai , China, Jun. 2000.

[36] Domenica Arlia. Experiments in Parallel Clustering with DBSCAN. In the Proceedings of the Euro Par '01 conference, LNCS, 2001.

[37] HBase. The Apache Software Foundation. http://hbase.apache.org/, retrieved in Jan. 2011.

[38] Owen Densmore. An Exploration of Power-Law Networks. Sun Microsystems Laboratories. http://backspaces.net/sun/PLaw/, retrieved in Jan. 2011.

[39] E. W. Dijkstra. A note on two problems in connexion with graphs. In Numerische Mathematik, 1959, pages 269-271.

[40] Jean-Baptiste Chaubet. Detecting Trending Topic on Chatter. KTH, Sweden, Apr. 2011.

[41] Dhruba Borthakur. The Hadoop Distributed File System: Architecture and Design. The Apache Software Foundation. 2007.

[42] Stuart P. Lloyd. Least square quantization in PCM. IEEE Transactions on Information Theory 28 (2), 1982, pages 129-137.

[43] Raymond T. Ng and Jiawei Han. Efficient and Effective Clustering Methods for Spatial Data Mining. In the proceedings of the 20th International Conference on Very Large Data Bases, 1994, pages 144-155.

[44] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, Jörg Sander. OPTICS: Ordering Points To Identify the Clustering Structure. In SIGMOD, 1999.

[45] Kaufman L. and Rousseeuw P. Finding Groups in Data: An Introduction to Cluster Analysis. 1990.

TRITA-ICT-EX-2011:218

www.kth.se