UNIVERSITAT POLITÈCNICA
DE CATALUNYA

GraphLib: a Graph-Mining Library based on the Pregel Model

Maria Stylianou

A thesis submitted in partial fulfilment of the requirements for the degree of
European Master in Distributed Computing

Industrial Supervisor:	Dionysios Logothetis
Academic Supervisor:	David Carrera

July, 2013

Acknowledgements

I would like to express my sincere gratitude to my supervisors from Telefónica; Dionysios Logothetis and Georgos Siganos, for their guidance, support, unlimited patience and boosting discussions that gave me inspiration and strength throughout the semester. I would also like to thank my academic supervisor David Carrera for his advice and guidance. Moreover, I am thankful to all my colleagues from Telefónica who have been a great source of motivation; especially to the Giraph-expert Claudio Martella, recommender-experts Linas Baltrunas and Alexandros Karatzoglou, the systems-guy Ilias Leontiadis and Telefónica's research department lead Dina Papagiannaki.

A big thank you to all my EMDC classmates, especially to Mario for his daily cheerfulness and brain storming moments, and of course to my classmates and friends Ioanna and Manos for their compassion and helpful discussions we had every week in placa del Sol. They all played a big role for improving my self-confidence and giving me courage to confront any kind of problems.

I would like to thank Johan Montelius and Leandro Navarro for organising this masters program and giving me the opportunity to be part of it!

Last but never least, I would like to thank my parents and especially sister for all their support and patience without whom none of this would have been possible.

Thank you!

Barcelona, July, 2013

Maria Stylianou

To my lovely family,

and especially my sister

Stella.

EMDC, European Master in Distributed Computing

This thesis is part of the curricula of the European Master in Distributed Computing (EMDC), a joint program among Royal Institute of Technology, Sweden (KTH), Universitat Politécnica de Catalunya, Spain (UPC), and Instituto Superior Técnico, Portugal (IST) supported by the European Community via the Erasmus Mundus program.

The track of the author in this program has been as follows:

- First and second semester of studies: UPC
- Third semester of studies: KTH
- Fourth semester of studies (thesis): UPC

Abstract

Big data analytics form a new area of focus for many research studies. Analysed data are most commonly represented in graphs leading to the need of exploring and advancing graph mining techniques and algorithms. Pregel - a programming Bulk Synchronous Parallel model for processing large graphs - is gaining popularity for such analytics. Its flexible programming model and scalable architecture are two main aspects that make its spread rapid. Several Pregel-based systems and platforms have been developed, however with limited implementations of algorithms built on top of Pregel.

In this thesis project, we propose and present the development of a library with graph-mining algorithms based on the Pregel model. We distinguish Apache Giraph, a popular open-source Pregel-based implementation that tends to become prominent among other Pregel-based systems. The library is built on top of Giraph and will be soon contributed to the open-source community. Apart from the design and implementation, we present the banchmarking performed for evaluating Pregel's properties. We show that the algorithms implemented following the Pregel model can scale and be executed in an efficient way. We finally show that Pregel model is a promising model and can lead graph-mining analytics to be conducted following this direction.

Categories and Keywords

Categories and Subject Descriptors

G.2.2 [Mathematics of Computing]: Graph Theory, Graph-Mining Algorithms

H.3.4 [Systems and Software]: Distributed Systems

D.2.13 [Software Engineering]: Reusable Software, Reusable libraries

Keywords

Graph Processing

Apache Giraph

Pregel

Bulk Synchronous Parallel

Large scale optimization

Collaborative Filtering

Graph Partitioning

Index

List of Figures

List of Tables

List of Algorithms

Acronyms

ALS	Alternating Least Squares
BSP	Bulk Synchronous Parallel
CF	Collaborative Filtering
GPS	Graph Processing System
HDFS	Hadoop Distributed File System
RDD	Resilient Distributed Datasets
RMSE	Root Mean Square Error
SGD	Stochastic Gradient Descent
SVD	Singular Value Decomposition

1 Introduction

Mining large graphs, such as social graphs, is important for several big data analytics. Pregel is gaining popularity for such analytics, due to the flexibility of the programming model and the scalability of the underlying architecture. However, there is a lack of open-source implementations of algorithms on top of Pregel. In this project, we focus on two areas of graph-mining problems and we port existing popular algorithms and design new ones for the Pregel model. Apart from the design and implementation, the project involves workload analysis and benchmarking of the algorithms. Our library is developed on top of Apache Giraph, a popular open-source Pregel implementation.

1.1 Motivation

With the evolution of social networks, like Facebook[1] and Twitter[2], big data analytics are becoming a necessity for companies growth and a major topic in research centres. Data derived from such analytics tend to be highly personalized and interconnected due to the shaped strong community structure [1–3]. Therefore, graph mining forms an integral and necessary part for big data analytics, as it is suitable for processing such data structures. Several graph processing architectures made their appearance - and they are presented in more detail in the Related Work section - but Pregel seems to stand out and gain popularity, due to the flexibility of the programming model and the scalability of the underlying architecture. Apache Giraph is a popular open-source Pregel implementation which provides the framework to build graph-mining algorithms. Few widely-used algorithms, e.g. Shortest Paths and PageRank are present in Giraph, however the amount of algorithms implemented in Giraph is very limited. It is evident the need of more graph mining algorithms which would cover different graph problems.

[1]http://www.facebook.com
[2]http://www.twitter.com

1.2 Contributions

In the current master thesis, we conducted a study on graph-processing problems that require scalable solutions and identify two major areas of such problems; Collaborative Filtering and Graph Partitioning. Based on these findings, we port existing popular algorithms and design new ones utilizing the Pregel model. The algorithms constitute a library called GraphLib. Our library is built on top of Apache Giraph, an open-source Pregel-based infrastructure that allows the development of graph-processing algorithms. Apart from the algorithms design and implementation, the project involves experimentation and evaluation of the algorithms and subsequently of the Pregel model. As a result, this master thesis makes the following contributions:

- Implementation of two known algorithms used in Collaborative Filtering; a popular technique in recommendation systems for automatic predictions of users interests. The two algorithms, namely Stochastic Gradient Descent (SGD) and the Alternating Least Squares (ALS), fall in the category of optimization algorithms. Their aim is to minimize the error in such predictions by iteratively recalculating values that affect the error.
- Implementation of a scalable graph partitioning algorithm. Graph partitioning is an efficient technique that allows graph processing systems and graph databases to scale. The implemented algorithm partitions a graph in k-way balanced partitions while minimizing the cut-edges. It does not require the knowledge of the global view of the graph, which makes it scalable and efficient.
- Analysis and Evaluation of the algorithms. The main metrics we evaluate are; scalability by adding more processing units to execute the algorithm, scalability in terms of processing very large datasets, and cost of the algorithm in terms of computation and communication overhead. The evaluation was conducted using real-world datasets, e.g. MovieLens dataset, Tuenti, LiveJournal and Youtube social graphs. The datasets size ranges from one hundred thousand edges to one and a half billions of edges.

1.3 Research History

The main objective of this work is to design a library of scalable graph-mining algorithms and evaluate their time performance, scalability and efficiency to achieve the goals predefined for each. The library heavily lies on the Pregel model and is built on top of Apache Giraph.

The first phase of the project is to extendedly study the Giraph project and become familiar with the coding style. Then, we determine the areas of problems being in the center of research interest, which are collaborative filtering and graph partitioning, and based on our decisions we choose algorithms that address these problems; Stochastic Gradient Descent [4] and Alternate Least Squares [5] for the former area and a scalable partitioning algorithm proposed by Vaquero et al. [6] for the latter area. After studying and understanding the areas of problems and algorithms, we move on to the implementation part.

While the partitioning algorithm is kept to one version, in the collaborative filtering algorithms, we were able to design more versions due to their flexibility to demonstrate different characteristics. In order to validate the results produced by the ALS and SGD algorithms, a standard version of the algorithms was implemented. Additionally to the SGD algorithm, another implementation proposing a delta caching solution was designed. The motivation behind this implementation is to achieve lower communication traffic with the requirement of a bigger storage - a small trade-off that could be easily tolerated. In the end, we evaluate all versions for each algorithm using a variety of benchmarks and workloads, identifying their strengths and drawbacks. We also analyse the efficiency of the Pregel model, by measuring the execution time of algorithms on different datasets and number of workers.

During my work, I benefited from the fruitful collaboration with the other members of the team working on graph mining algorithms, namely Dionysios Logothetis, Georgios Siganos, Claudio Martella and Ilias Leontiadis.

1.4 Thesis Roadmap

The rest of this thesis report is organized as follows: Chapter 2 describes the background behind our work and surveys existing graph processing solutions. Chapter 3 gives a complete description of the proposed library and Chapter 5 describes implementation details and issues faced. In Chapter 6, we present the evaluation with the experiments conducted. Finally, we conclude in Chapter 7 by summarizing our findings and future work.

2 Background

Graph processing is vital in various areas; social networks, the world wide web, protein interaction networks and metabolic networks [7]. Achieving scalability of graph-processing algorithms can strive in efficiently processing peta-scale graph data. Over the last decade, several graph processing architectures are proposed and some of them adopted. However, none of them accomplishes scalability. In this chapter we first describe the research work that forms the basis and motivation for the development our library. Thereafter we survey relevant graph-mining systems used in both industry and the research area.

2.1 MapReduce

The MapReduce programming model is designed for processing vast amounts of data in a parallel manner [8]. Its efficiency on execution of algorithms is bounded by the requirement of using large clusters of commodity hardware which belong to a distributed file system. Apache Hadoop, an open-source implementation of MapReduce, uses the Hadoop Distributed File System (HDFS) for reading the input and writing the output [9]. The popularity of HDFS arises by virtue of being highly fault-tolerant, scalable, and able to handle large data sets.

MapReduce comprises two functions; Map and Reduce which are directly derived from functional programming. Figure 2.1 depicts the stages followed when submitting a MapReduce job. Initially, the Input Reader reads the input from a stable storage (HDFS) and transforms the entries into key-value pairs. Mappers receive blocks of these generated pairs in parallel as their input and execute the same user-defined function which is specified as a parameter at the beginning. The output from the mappers is again a set of key-value pairs which gets sorted by key, partitioned according to a specified partitioning function and introduced to the reducers. Similarly to the mappers, reducers execute a user-defined function which is also specified at the beginning. The results from all reducers are stored in parallel to the stable storage, through the Output Writer.

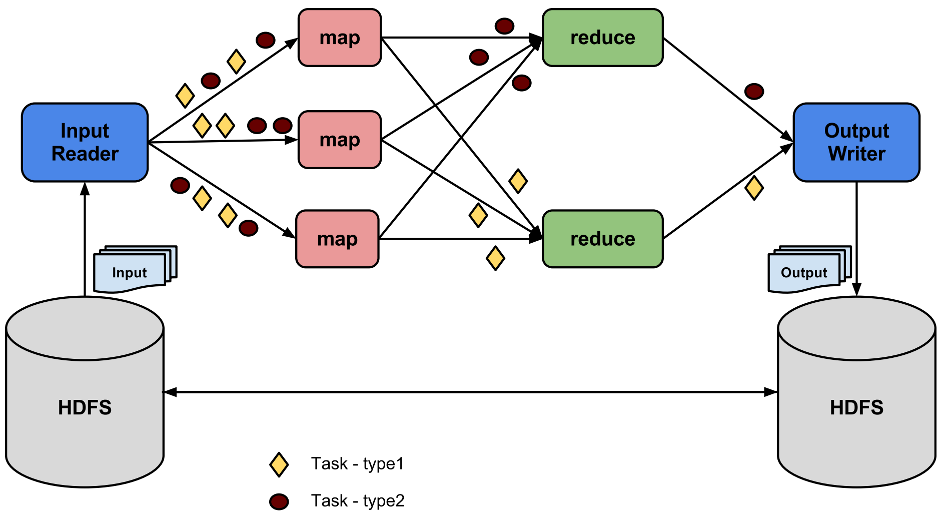


Figure 2.1: The MapReduce Programming Model

The widespread use of this model is attributable to its limited complexity. Users of MapReduce only have to write the two functions applied in the two phases, with the compromise of being less flexible when algorithms become more complex. To be precise, iterative algorithms require a sequence of executions, indicating multiple Map and Reduce executions, which can only be developed as a sequence of MapReduce jobs. This implies the necessity of repeatedly passing the data from one job to the other causing a disobliged overhead on the execution time. Processing large graphs is argued to also be inefficient when using MapReduce [10, 11]. The difficulty lies in the drawback described above; in the necessity to pass the entire graph state from one phase to the other causing excessive I/O traffic and significantly decreasing the performance.

2.2 Bulk Synchronous Parallel

Leslie G. Valiant proposed first the Bulk Synchronous Parallel (BSP) model for parallel computation [12]. A BSP computation is executed on a set of processors which are connected in a communication network but work independently by each other. The BSP computation consists of a sequence of iterations, called supersteps. In each superstep, three actions occur: (i) Concurrent computation performed by the set of processors. Each processor has its own local memory and uses local variables to independently complete its computations. This is the asynchronous part. (ii) Communication phase during which processors send and receive messages. (iii) Synchronization which is achieved by setting a barrier. When a processor completes its part of computation and communication, it reaches this barrier and waits for the other processors to finish. Figure 2.2 illustrates the actions applied in one superstep.

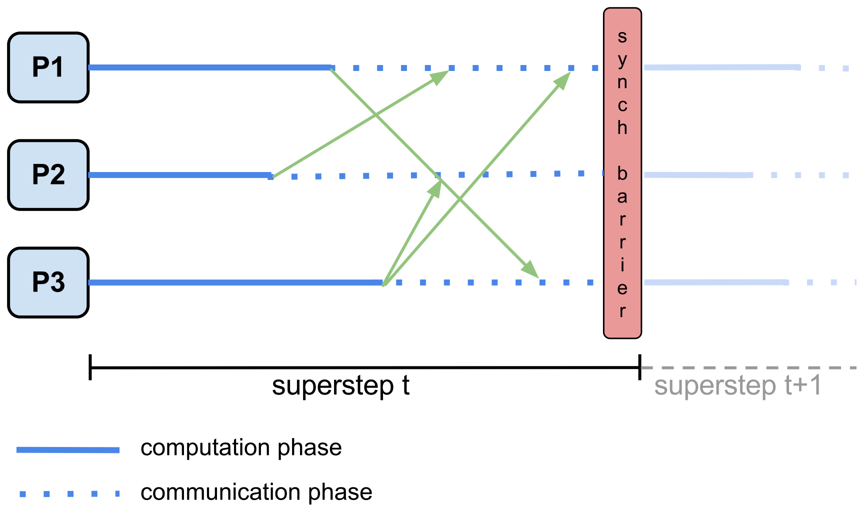


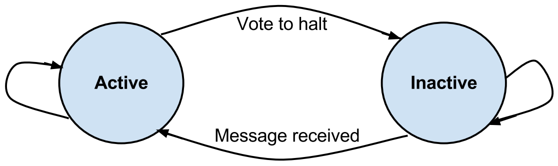
Figure 2.2: Bulk Synchronous Parallel model

BSP is an alternate model to MapReduce. MapReduce algorithms can be simulated on BSP and vice versa. BSP is beneficial for processing big data with complicated relationships, like graphs, as well as iterative algorithms; both categories being a great challenge for MapReduce.

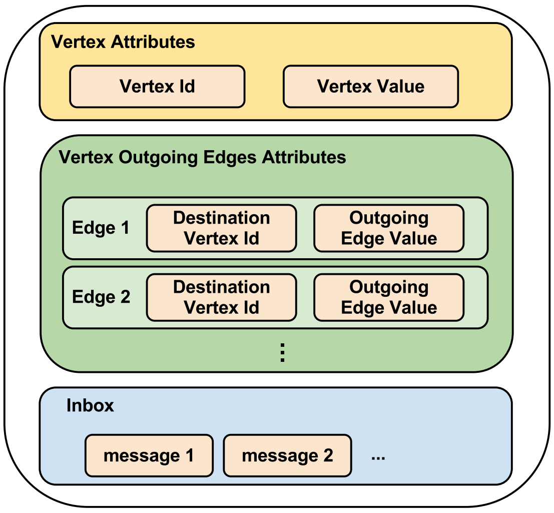
2.3 Pregel

Pregel - a powerful framework built by Google - is proposed for petabyte-scale graph processing [10]. The essence of programming in Pregel is to *"think like a vertex"*, which makes it extremely expressive and intuitive [13]. A vertex can be either active or inactive (Figure 2.3(a)). With the launch of a Pregel algorithm, a vertex is set to the active state and it remains there till it votes to halt, thus it passes to the inactive state. If it receives a message in one of the following supersteps, it becomes active again. One can think of the vertex as a *node* (Figure 2.3(b)) that contains: (i) its attributes; Vertex Id and Vertex Value, (ii) its outgoing edges and their attributes; Destination Vertex Id and Edge Value, and (iii) a logical *inbox* in which it receives messages from other vertices.

Pregel is heavily based on the BSP model. A Pregel computation is executed through synchronous iterations of asynchronous computation. The program consists of a user-defined function which is executed by each vertex during a sequence of supersteps. In each superstep, a vertex can first receive messages by



(a)



(b)

Figure 2.3: Vertex behaviour and structure: (a) States of a Vertex, (b) Structure of a Vertex

other vertices sent in the previous superstep, then execute the user-defined function - in which its state and its outgoing edges' state can be modified - and at the end send messages to other vertices which will be delivered in the next superstep. During a superstep vertices work independently and asynchronously, while at the end of a superstep, a global synchronization occurs among vertices in order to exchange messages. Two mechanisms are added to improve performance and usability; (i) combiners are adopted for reducing network traffic, (ii) aggregators are utilized for monitoring the global communication. This execution loop is terminated when all vertices vote to halt and there are no messages in transit i.e. to be sent in the next superstep.

A Master-Slave architecture is applied in Pregel. A node takes the role of the master for centralized decisions. It decides how many partitions of vertices the graph will have and assigns one or more partitions to each worker machine. The master is not assigned any portion of the graph, instead it coordinates the synchronization, while workers are responsible for maintaining the state of their section from the graph. They execute vertices computation and communicate with each other. Communication is achieved through message-passing and computation is executed in memory. Fault tolerance is achieved by directing workers to store their state (vertex Id, value, edge values and incoming messages) to a persistent storage, while the master stores aggregator values if any. This step is done at the beginning of a superstep, and the repetition of storing is determined by the checkpoint frequency.

2.4 Giraph

There is a wide range of systems built to either offer the BSP computation or specifically implement the Pregel framework. For this thesis project, we have chosen Apache Giraph [14], an open-source implementation inspired by both BSP and Pregel. Giraph is an Apache incubator project for processing large-scale graphs in the form of a Hadoop Map-only job. Due to this facilitation, Giraph code can be easily added in existing Hadoop infrastructures and used by Hadoop developers. By default HDFS is adopted as the distributed file system, and Zookeeper is used as the centralized coordination service, hence preserving fault tolerance. Giraph includes all Pregel optimizations, i.e. combiners, aggregators and other features like master computation and a no-single-point-of-failure design.

2.5 Related Work

Giraph is a **BSP** system based on Pregel. It supports **message passing** algorithms and graph processing is accomplished in a **synchronous** environment. In this section, we survey large-scale graph processing systems in respect to these characteristics.

2.5.1 Bulk Synchronous Parallel / Pregel-based Systems

Pregel has been the first work that introduced BSP computation in a synchronous distributed message-passing system. With the emergence of this model, several systems alike have been designed; Giraph [14], GoldenOrb [15], Phoebus [16], GPS [17] and Hama [18]. While Giraph is the most advanced system, GPS and Hamma also stand out from the rest.

GPS: A Graph Processing System - GPS is the closest system to Giraph, developed in Stanford University InfoLab and published as an open-source project. The execution of algorithms in GPS is identical to Giraph, hence the characteristics of the two systems are also similar; GPS is scalable, fault-tolerant, and designed to support algorithms executed on very large graphs, like Giraph. However, it differentiates from Giraph in two ways; (i) GPS offers a dynamic repartitioning scheme, in which vertices can get reassigned to different machines, and (ii) an optimization of GPS called *large adjacency list partitioning* is present for reducing the network traffic and assure improved performance.

Hama - Apache Hama [18] is a computing framework purely following the Bulk Synchronous Parallel model. It is inspired by Google′s Pregel, though it differs from Apache Giraph as it is suitable not exclusively for graph processing algorithms, but also for matrix and network algorithms. Being more generic is not necessarily negative, but in the case of Hama, a graph-algorithms developer would need to take care of other details like building the I/O API. Up until recently, Pregel optimizations like combiners and aggregators were nonexistent, making Hama less usable.

2.5.2 Hadoop-based Systems

As described in 2.1, MapReduce fails to express iterative computations. Researchers tried to solve this obstruction by "translating" each iteration to a MapReduce job. Hence, an iterative computation can be written as a sequence of MapReduce jobs. Systems have been built on top of Hadoop, in which users can write iterative algorithms as a sequence of MapReduce jobs; for instance Mahout [19], HaLoop [20], Pegasus [21], iMapReduce [22] and many others. The main disadvantage of these systems lies on the possible requirement of needing to pass the input graph in every MapReduce job, even if it has not changed. This causes additional overhead and decrease to the performance. Despite this drawback, some systems have become very popular and advanced; we describe them below.

Mahout - Apache Mahout [19] serves as a library for scalable machine learning and data mining. It specializes in problems like collaborative filtering and clustering, of which the first is a targeted area in our proposed library. Mahout can be also used for graph mining, in return of accepting the overhead that comes along.

HaLoop - HaLoop [20] is a modified version of Hadoop MapReduce framework with the improvement of supporting iterative algorithms. This is achieved by adding a loop-aware task scheduler and various caching mechanisms. Currently, the HaLoop exists as a prototype system with main focus on making it robust and stable.

2.5.3 Asynchronous Systems

In Giraph, algorithms are executed in a sequence of supersteps which are divided by synchronization barriers. Similarly, GPS and Hama support synchronous graph processing, paying the limitation of fast processors waiting for the slow ones to complete their computation part. Nevertheless, systems like GraphLab [11] and Signal/Collect [23] support asynchronous graph processing, overpassing this constraint. Though, in asynchronous environments, programming and reasoning the order of execution becomes very complex and hard for the programmer.

GraphLab - GraphLab [11] is a distributed computation framework deployed for graph processing. Initially, its usability was bounded for Machine Learning executions, however with its arising popularity, it is nowadays used for many other data-processing algorithms. GraphLab has a shared memory design and allows asynchronous iterative computations for achieving high performance.

Signal/Collect - Signal/Collect is a framework for large-scale graph processing, supporting both categories of algorithms; synchronous and asynchronous [23]. Similarly to Pregel, there are a sequence of iterations, during which signals are sent and collected among edges and vertices. Not only vertices have a compute method, but also edges behave as entities with their own computation part. Additionally, the synchronization barrier is loose in order to allow asynchronous implementations.

2.5.4 Other Systems

Several systems do not precisely fall in the categories above, nevertheless they are designed for data or graph processing and worth to be mentioned. Among them we distinguish HigG and Spark, which are briefly described below.

HipG - HipG is a distributed framework dedicated for high-level parallel processing of large-scale graphs. Apart from supporting graph algorithms, it also takes divide-and conquer algorithms as well as on-they-fly graph algorithms. HipG does not have a global synchronization barrier like Pregel, but it can specify several barriers for different subsets of vertices. Hence, a more fine-grained synchronization is achieved, preventing vertices to fall into idle-state while waiting for others to complete their execution.

Spark and Bagel - Spark [24] is a cluster computing system, designed for fast data analytics. The initial objective for developing Spark was to facilitate iterative algorithms and interactive data mining, by keeping data in memory. However, it was later used for more general data analytics purposes. One of the most important advantages of this open-source system is the way of implementing caching. By using Resilient Distributed Datasets (RDD) and caching such datasets in memory, it allows them to be reused across iterations making future actions faster. Following the Spark schema and inspired by Pregel, Bagel [25] makes its appearance. Similarly to Pregel, an algorithm is executed in a sequence of supersteps. This work includes aggregators and combiners, though Bagel is still initial supporting basic graph computation.

3 GraphLib

3.1 Overview

To the best of our knowledge, our proposed library GraphLib is the first attempt to fill the gap of graph-mining algorithms implemented following the Pregel framework. GraphLib is deployed on top of the Apache Giraph project. The main characteristics of GraphLib, acquired either by Giraph or by its exclusive objectives, are:

- **Scalability**: Algorithms in GraphLib follow the Giraph infrastructure which enforces Pregel's technique to 'think like a vertex'. Being vertex-centric, Giraph-based algorithms do not require a global view of the entire graph which makes them scalable and time effective.

- **Fault Tolerance**: Algorithms can make use of periodical checkpoints for storing various states of computation and handling possible workers failures. When a checkpoint is reached, master and workers store their data in HDFS; the master saves aggregated values and workers save the state of vertices assigned to them. When one or more workers fail, the current state of their associated vertices is lost. Hence, the master reassigns these vertices to available workers which reload their state from the most recent checkpoint. Perforce, the entire system restarts from that checkpoint.

- **Collaborative Filtering Oriented**: Collaborative filtering has received great popularity over the last decade. Social networks like Facebook and LinkedIn have evolved in this area by offering recommendation services to their users. The trust built between the users and the recommendation services is heavily based on the accuracy of the recommendations. SGD and ALS are two optimization algorithms that focus on the minimization of the error on recommendations. The followed approach in these algorithms suggests the study and correlation of users and users' previous preferences. With matrix factorization, they strike a very fine optimization and manage to deliver a high percentage of reliable predictions.

- **Graph Partitioning Oriented**: Graph partitioning is useful in many graph problems. Usually graph databases and graph processing system need to partition the underlying graph in order to run efficiently and scale. Additionally, machine learning algorithms also must partition the underlying graph to run more efficiently. However, there are no algorithms that scale to large graphs. Metis [26], which is the golden standard for partitioning graphs, does not scale to large graphs. Therefore, it is essential to introduce new algorithms able to scale and handle large graphs with billion of edges.

3.2 Collaborative Filtering

3.2.1 Problem Definition

Collaborative Filtering (CF) is a technique widely used by recommendation systems [5]. The main objective of such systems is to identify patterns of users' interests and suggest specific items to targeted users based on their preferences, views and purchases. The input in such systems varies. Users may give stars or numbers to rate an item, the so-called explicit feedback. They may just give a 'like' to show their interest, or buy the item without specifying the level of their interest; this feedback is called implicit and it is in general less helpful. For this project, we only deal with explicit feedback and we use *ratings* as the input type.

A fundamental need in CF is to relate the two entities; items and users. This is achieved with approaches from collaborative filtering, like *the neighbourhood approach* and *latent factor models* [4]. The former approach focuses on shaping relationships between items; this is also called item-based approach and it makes recommendations to a user based on his previous ratings to similar items. The latter approach - and the one we use later on - characterizes users and items on factors and brings both entities to the same latent factor space. Subsequently, recommendations to a user are brought by investigating his and other users' previous ratings. This correlation with other users justifies the alternative way of naming this technique as user-based.

Figure 3.1 demonstrates the process in user-based CF. At first (a), people give ratings to various items. These ratings are added in a user-item matrix for easier process of users interests (b). Then (c), the system makes a prediction about a user's rating for an item that the user has never rated before. The prediction is based on others users' ratings who have given similar ratings with the active user in the past.

Commonly, the datasets used in CF are very large i.e. at the scale of millions, and vary depending on the CF method. Latent factor models are successfully realized using *matrix factorization* [27]. Several matrix factorization algorithms are proposed with the aim of reducing the error in predictions. To name a few; ALS, SGD, SVD++ and non-negative matrix factorization. For the current project we have implemented ALS and SGD. Below, we provide some preliminary knowledge that prepares the reader for the description of the two algorithms.

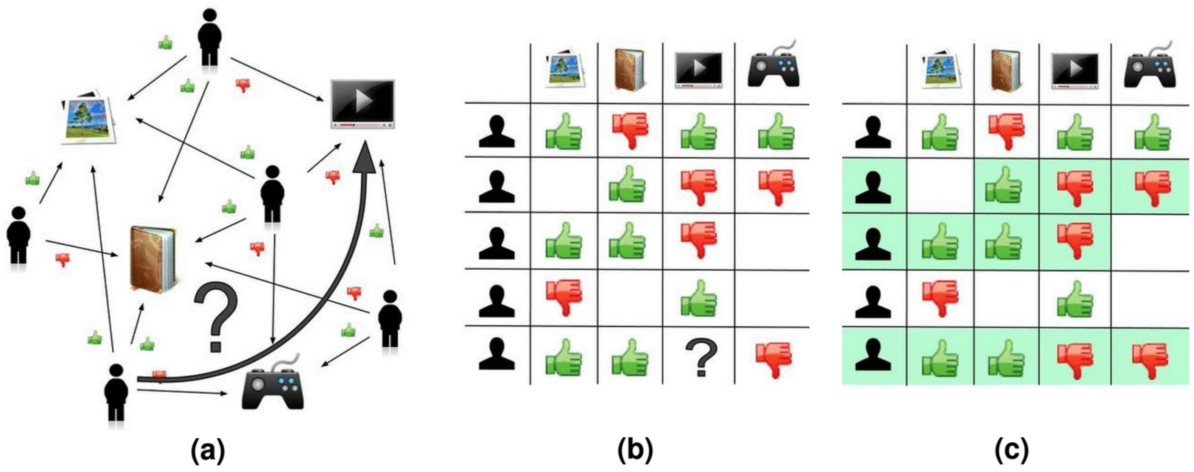


Figure 3.1: Steps followed in Collaborative Filtering[a]

[a]Source: http://en.wikipedia.org/wiki/File:Collaborative_filtering.gif

3.2.2 Preliminaries

3.2.2.1 Matrix Factorization

The algorithms implemented fall in the category of latent factor models and they specifically follow the matrix factorization model. Users and items are characterized by vectors of factors derived from previous item ratings and they are placed together in a joint latent factor space of dimensionality f. All users and items vectors have as factors items characteristics. Each user u is characterized by a vector $p_u \in \mathbb{R}^f$, in which its elements measure the extend of user's interest for each factor. Similarly, each item i is characterized by a vector $q_i \in \mathbb{R}^f$. q_i, and its elements measure the extend to which the item possesses those factors. These elements can be either positive or negative. The dot product $q_i^T p_u$ shows the interaction between the user u and the item i, i.e. the user's interest in the characteristics of the item, and thus the dot product is the predicted rating, denoted as $\hat{r}_{ui}$ (3.1).

$$\hat{r}_{ui} = q_i^T p_u \tag{3.1}$$

Table 3.1 gathers special characters and their representation, which we use while describing the algorithms. As it has been said, we deal with two sets of entities; *m* users and *n* items. Both entities are mapped in a matrix, therefore index characters are utilized; u and v for users, i and j for items. A user u gives a rating r_{ui} to item i. The rating can be an integer in the range from 1 (no interest) to 5 (strong interest). Known ratings are identified with r_{ui}, while predicted ratings take the notation $\hat{r}_{ui}$. The prediction is the result of the dot product of the latent vectors of the active user and item to be ranked by the user. The difference between the predicted and the known rating gives the error e_{ui} in the prediction (3.2).

$$e_{ui} = r_{ui} - \hat{r}_{ui} \tag{3.2}$$

Special Characters for CF	**Representation**
m	Number of Users
n	Number of Items
u, v	Indexing letters for users
i, j	Indexing letters for items
r_{ui}	Rating for item i from user u
$\hat{r}_{ui}$	Predicted value of r_{ui}
e_{ui}	Error

Table 3.1: List of Special Letters for Collaborative Filtering Algorithms

3.2.2.2 Halting Conditions

Both SGD and ALS are iterative algorithms, which implies the need of a halting condition to the loop. The termination of the algorithms may vary depending on the value we focus on and what we would like to achieve. Possible halting conditions can be:

- **Maximum number of Iterations**: The time of execution is an important factor that drives many scientists to decide whether to allow or interrupt the execution of an algorithm. In such cases, the halting condition can be a maximum amount of iterations. A programmer can set this number after some empirical experimentation and expect a good convergence. Certainly, a good result may not be expected in every scenario, since the algorithm behaves differently depending on the density, size and structure of the graph.

- **Root Mean Squared Error (RMSE)**: The RMSE is a metric popularized during the Netflix prize [28] for movies recommendation performance. It is a measure of the differences between the ratings predicted and the known ratings - otherwise called *observed*. RMSE is suitable when accuracy is an important factor to measure. RMSE aggregates the errors in predictions into a single measure, giving an overall observation of the total error. (3.3) gives the RMSE equation, in which $\|r\|$ is the total number of ratings.

$$RMSE = \sqrt{\frac{1}{\|r\|}\sum_{u,i}(r_{ui} - \hat{r}_{ui})^2} \tag{3.3}$$

- L^2**-Norm**: In some cases, it is important to observe the changes on the users' and items' values. The L^2-Norm is a measure of the difference between the initial and the final values of a user's (3.4) or item's (3.5) vector. In the equations below, the p'_u and q'_i are the initial vector values of the user and item respectively. By *initial*, we mean the latent vector values given during values initialization at the first superstep.

$$L^2 - Norm(p_u) = \sqrt{(p'_u - p_u)^2} \tag{3.4}$$

$$L^2 - Norm(q_i) = \sqrt{(q'_i - q_i)^2} \tag{3.5}$$

3.2.3 Stochastic Gradient Descent Algorithm

Stochastic Gradient Descent (SGD) algorithm is an optimization algorithm, typically used for the training of neural networks [29] as well as for predicting the ratings of users to items as described in the previous section [27]. The algorithm takes as an input a training set with user-item pairs and their corresponding known ratings. It then creates latent vectors for both users and items and initializes them using their Id. The aim is to train the vectors of both users and items, and make predictions of the ratings for every user-item pair, yielding to a minimum error between the predicted and the known ratings.

In addition to the list of parameters given in the previous subsection, the following parameters are also used in the SGD:

- **Learning Rate -** γ: In each iteration, the users and items vectors are adjusted. The learning rate γ controls the step size, i.e. how large the adjustments can be in each iteration.
- **Regularization Parameter -** λ: A common problem while training the system is overfitting [30]. Overfitting occurs when the system tries to learn the model by fitting the observed ratings, but it fails to generalize the ratings in order to be able to successfully predict new ones. A regularization parameter λ is used to penalize the magnitudes of the learned parameters and to subsequently ensure avoiding this issue.

An SGD optimization, proposed by Simon Funk [31] and later adapted by others [32, 33], suggest to iterate through all edges in the training set and recalculate the latent vectors in each iteration, till the error in prediction converges to a very small value defined beforehand, which implies that the predicted rating approached the known rating to the closest. For each edge, the system calculates the predicted rating (3.1) and computes the error (3.2). It then adjusts the user and item vectors by a magnitude proportional to γ. Equations (3.6) and (3.7) show the vectors' modification for the user u and item i respectively. The regularization parameter is also added to penalize the learned values.

$$p_u = p_u - \gamma \cdot (e_{ui} \cdot q_i + \lambda \cdot p_u) \tag{3.6}$$

$$q_i = q_i - \gamma \cdot (e_{ui} \cdot p_u + \lambda \cdot q_i) \tag{3.7}$$

3.2.4 Alternating Least Squares Algorithm

Alternating Least Squares (ALS) is a matrix factorization algorithm [5]. It is based on the observation that users and items vectors are unknown values, though by fixing one of the two vectors, the problem becomes quadratic and can be optimally solved. Hence, ALS tries to minimize the error between the predicted and known ratings by rotating between two steps:

1. Fix p_u and compute q_i by solving a least-squares problem (3.8).

2. Fix q_i and compute p_u by solving a least-squares problem (3.9).

The two steps are repeated, till the error converges to a predefined number.

$$q_i = q_i + e_{ui} \cdot p_u + \lambda \cdot q_i \cdot n_i \tag{3.8}$$

$$p_u = p_u + e_{ui} \cdot q_i + \lambda \cdot p_u \cdot n_i \tag{3.9}$$

Similarly to SGD, ALS may lead to overfitting, therefore a regularization parameter λ is used to penalize large parameters.

3.3 Graph Partitioning

3.3.1 Problem Definition

With the rise of social networks and the abrupt increase of users and user activity, mining large dynamic graphs becomes crucial. The performance time in processing such graphs is negatively affected by the dynamic nature and the very large size of the graphs. To achieve scalability and better performance, researchers turn to graph partitioning which is a big challenge itself.

While designing a graph partitioning algorithm, two requirements have to be met:

1. **Data Locality**: The partitioning should be done in such a way that the communication overhead will be minimum. This is achieved by minimizing the number of edges among different partitions, alternatively called *cut-edges*.
2. **Load Balancing**: the vertices should be placed in partitions in such a way that all partitions have approximately the same number of vertices or edges. Thus, the algorithm should ensure that it can produce k-way balanced partitions.

Along with these requirements, large dynamic graphs come with characteristics that give a boost in the challenges of partitioning:

- Data locality and load balancing often conflict, since the former requires many nodes to co-exist in one partition while the latter forces a balanced partition that may lead to partition of neighbours.
- The large size of graphs imply the need of a scalable and efficient implementation of a partitioning algorithm. The majority of the existing graph partitioning algorithms require a global view of the graph which make them not able to scale.
- The dynamic nature or graphs lead to the necessity of processing the graph continuously and therefore partitioning must quickly adapt to graph changes.

3.3.2 Preliminaries

Table 3.2 lists special characters we use while describing the partitioning algorithm. When partitioning a graph, a number of partitions $|P|$ as well as a maximum capacity C of nodes each partition can host have to be predefined. One of the algorithm objectives is to result to k-way balanced partitions, therefore the capacity C must be the same for all partitions. The algorithm is iterative, thus the letter t is used for identifying the iteration number, while i, j are used for identifying partitions. In each iteration t, a partition i has $P^t(i)$ users and $C^t(i)$ remaining capacity for future possible migrations of users from other partitions that may want to move to partition i. It can easily be deduced that the remaining capacity is the deduction of current users from the total capacity (3.10). $Q^t(i, j)$ is the amount of users that can migrate from partition i to partition j. This number is adequately explained in Section 5.

$$C^t(i) = C - P^t(i) \tag{3.10}$$

3.3.3 Dynamic Graph Partitioning Algorithm

To address the problem of graph partitioning, we chose an algorithm proposed by Vaquero et al [6]. The algorithm takes into consideration all challenges described above. Precisely, the authors propose a scalable graph partitioning algorithm that:

- Minimizes the number of cut-edges until convergence.
- Produces k-way balanced partitions.

Special Characters for Graph Partitioning	**Representation**
C	Maximum Capacity in a partition
i,j	Indexing letters for partitions
t	Indexing letter for iteration
$\|P\|$	Number of partitions
$P^t(i)$	Number of existing users in partition i during iteration t
$C^t(i)$	Remaining capacity of partition i during iteration t
$Q^t(i,j)$	Amount of users that can migrate from partition i to partition j over iteration t

Table 3.2: List of Special Letters for Partitioning Algorithms

- Requires only local per-vertex information.
- Supports dynamic graph changes.
- Adapts to graph changes with minimum cost.

The partitioning algorithm is based on an iterative vertex migration technique. A vertex represents a user and all vertices pass from a number of iterations. In every iteration, Algorithm 1 is executed. As depicted from this pseudocode, each vertex goes through its friends and checks in which partition they belong to, increasing a counter for each partition a friend exists. It then migrates to the partition with the highest counter. The `while loop` halts when a threshold of stabilization rounds is reached. A *stabilization round* is an iteration in which none of the users migrates to another partition, meaning that the partitioning is stabilized for that iteration.

Algorithm 1 Graph Partitioning Algorithm

```
1:  $N \leftarrow n$ // Stabilization Rounds
2:  $stabilization\_rounds \leftarrow 0$
3:  while $stabilization\_rounds < N$ do
4:      for user in graph do
5:          $friends \leftarrow get\_friends(user)$
6:          for friend in friends do
7:              $friend\_partition\_counters++$
8:          end for
9:          $migrate(user,\ max(friend\_partition\_counters))$
10:     end for
11: end while
```

Summary

This chapter gives an overview of GraphLib, its main characteristics and features. It also defines the problems that GraphLib currently focuses on; Collaborative Filtering and Graph Partitioning. For each problem, we describe the algorithms chosen to address the problem. Next chapter tries to adequately explain the Giraph programming model on top of which GraphLib bases its implementation style.

4 Giraph Programming Model

Giraph code structure is followed in the implementation of GraphLib algorithms. Thus, it is necessary to give an introductory description of the Giraph programming model; how the graph is loaded from HDFS into Giraph, steps followed during execution and how the output graph is stored back to HDFS. Giraph features used in GraphLib are also described in the current section.

4.1 Overview

Giraph is the most active and complete open-source implementation of the Pregel model. Its - so far - success can be attributed not only to its ability to exactly represent the Pregel model, but also to the additional features that facilitate the utility of the system and make the programming model more flexible. Giraph is developed in Java language and the source code can be found on GitHub[1].

All algorithms in Giraph are Pregel-based, especially designed for graph processing. A graph is composed of vertices and edges; vertices represent entities like people and items, while edges represent relationships between these vertices. A Giraph computation must define and deal with attributes about the two components and their relationship. These attributes are:

- **Vertex Id**: Is used to identify a vertex.
- **Vertex Value**: Is used to store a vertex value.
- **Edge Value**: Is used to store a value on an outgoing edge.
- **Message Value**: Is used to store a value on the message to be sent to other vertices through the outgoing edges.

[1]http://giraph.apache.org/source-repository.html

The user must define the types of those values, which can be chosen from a long list of Writable implementations, i.e. IntegerWritable, LongWritable, DoubleWritable, and many more. If an attribute is not used in the algorithm, its type can be set to NullWritable.

A Pregel-based graph algorithm is vertex-centric and iterative; the programmer needs to design the algorithm thinking like a vertex that computes iteratively. Initially, the Giraph algorithm receives an input file with the input graph and all vertices are set to active. In each superstep, active vertices realize the computation provided by the user, which is the graph algorithm to be executed on the input graph. The sequence of iterations is completed when all active vertices vote to halt and there are no messages in transit, that is to be sent in the next iteration. With the completion of the iterative computation, the Giraph algorithm creates an output file with the result. These functionalities are distinguished and undertaken by separate pieces of code.

4.2 Load the graph into Giraph

The input graph can be either vertex-centric or edge-centric. A vertex-centric dataset consists of lines that represent vertices, i.e. a line provides the Vertex Id and depending on the algorithm it can have the Vertex Value, Vertex Id of neighbours and Edge Value. An edge-centric dataset consists of lines that represent edges, i.e. a line provides a pair of vertices; Vertex Id1 and Vertex Id2 and if necessary the Edge Value.

After deciding the type of the input graph, the programmer must write the *Input Format* code or choose one of the existing ones in Giraph, which is responsible to read the graph data and set the objects to be used in the algorithm. The Input Format can be either Vertex Input Format or Edge Input Format for a vertex-centric and edge-centric type respectively. Giraph offers several Input Formats for reading input graphs given in various formats; in text, in JSON format, etc. Certainly, a programmer can write his/her own customized Input Format for a specific input graph.

4.3 Store the graph from Giraph

With the completion of the Giraph computation, results should be stored to a persistent storage. Similarly to input graph, the output may be vertex-centric or edge-centric, thus using a Vertex Output Format or an

Edge Output Format respectively. The Output Format code can be more flexible than the Input Format; the programmer can choose to store any other data, statistics, information he/she wants by creating a customized Output Format which will write the desired values into the persistent storage. Each vertex executes the Output Format in order to output its final vertex value and any other local information needed.

4.4 Main Computation in Giraph

Apart from the Input Format and Output Format, an algorithm requires the main code which is the heart of computation. This code includes the method `compute()` which is executed by each vertex in each superstep. Typically, the `compute()` is composed by three parts:

- The vertex reads the messages sent by other vertices in the previous superstep.
- The vertex executes some computation considering the messages received as well as the vertex and edges values.
- The vertex **may** prepare a message and send it to its neighbours.

If a vertex does not receive any message, the vertex becomes inactive and the `compute()` is not executed. Moreover, if the halting condition is met, the vertex might not execute any computation nor send messages, depending on the position of the halting condition in the code. Though, if a vertex declares itself as inactive, but in the next superstep receives a message, it is then reactivated. The computation of the graph algorithm halts at the end of the superstep in which all vertices vote to halt and no messages are sent for the next superstep. It is important to mention that the `compute()` method does not have direct access to other vertices' values and outgoing edges. Such values can only be retrieved by receiving messages from other vertices.

4.4.1 Synchronization Barrier

Between consecutive supersteps there is a barrier, which implies the following:

- A supestep is considered complete only when all vertices have completed their computation.

- Vertices can start computation in the next superstep only when the current superstep is completed.
- Any message sent in a current superstep gets delivered only in the next superstep.
- Values remain the same across barriers i.e. at the beginning of a superstep, the values of vertices and edges are equal to the ones at the end of the previous superstep.

4.4.2 Master Compute

The Master Compute is an additional feature in Giraph that provides centralization in the algorithm. While the workers run independently by each other and asynchronously during a superstep, the Master Compute runs sole after the synchronization barrier is met by all workers. It is the first piece of code executed at the beginning of each superstep, before the workers start the `compute()` method. The Master Compute is useful in many ways; (i) it executes global computations between superstep, (ii) it makes checks based on values workers values and states, (iii) it can decide whether to halt the whole computation; on behalf of the workers as well.

4.4.3 Aggregator

The Aggregator is a feature provided by Giraph to allow global computation. During a superstep, vertices send values to an aggregator. The aggregator then aggregates these values to produce a global result (sum, maximum, minimum, etc.) or to check if a global condition is met. Vertices can retrieve the aggregated result in the next superstep.

There are two types of aggregators; the Regular Aggregators and the Persistent Aggregators. A *Regular Aggregator* resets its value to the initial one in each superstep, while the *Persistent Aggregator* preserves the value calculated each time.

Typically, the Master Compute and the Aggregators are used in combination. An aggregator must be registered by the master in order to run during computation. Between supersteps, the master code is executed, which can include the aggreated values from aggregators for global checks and computations.

Summary

In this chapter we gave a short presentation of how Giraph executes an algorithm, what files are needed and what important features should be taken into consideration before implementing in Giraph. Next chapter describes implementation details about the implemented algorithms as well as challenges and restrictions arose during implementation. Moreover, all developed packages that shape our library are listed for reference.

5 Implementation Details

GraphLib is a Pregel-based library, built on top of Giraph. Following the theoretical description of our library from Chapter 3, we now try to give the practical information on how we actually built the library and the algorithms themselves. Each algorithm implementation is divided in three parts; loading the input graph from HDFS, storing the output graph to HDFS and executing the main computation. Below we describe these parts separately as well as other features from Giraph used. Moreover, we make an analysis of the challenges and difficulties we confronted along with the decisions taken for each case.

5.1 Stochastic Gradient Descent

SGD is an optimization algorithm that aims to the minimization of a value. As described in section 3.2.3, SGD receives a set with user-item pairs and their ratings as an input. It then creates latent vectors for both users and items and predicts the ratings based on these vectors. By training the vectors through a sequence of supersteps, it achieves minimization of the error between the observed and the known ratings. Like every Giraph algorithm, SGD is composed by three pieces of java code, listed below.

5.1.1 Input Format

The input file given to SGD algorithm consists of user-item pairs and their ratings. Each line has three values; user's Id, item's Id, the rating from user to item. For reading the input, we wrote an Input Format, called **`IntDoubleTextEdgeInputFormat.java`**. It is edge-centric and expects values of type integer for the Ids and a type double for the rating.

5.1.2 Output Format

The Output Format **`IntDoubleArrayHashMapTextVertexOutputFormat.java`** is responsible to output information for each vertex. Each line includes the Vertex Id and the Vertex Value as calculated in the last superstep. It can optionally include the error value for each rating, the number of updates of the Vertex Value and the number of messages received from the Vertex in the whole computation time. The error value can represent the RMSE, the L2Norm or the common error calculated by the dot product of the predicted and the known rating. The choice of which value to output depends on the halting condition chosen before the execution of the algorithm. The three values: error value, number of updates and number of messages can be specified by the user whether to be printed or not.

5.1.3 Main Computation

The Giraph computation of SGD consists of vertices which represent both users and items. At Superstep 0 and during all the next even-numbered supersteps, vertices represent users; they train the latent vectors of users. Similarly, at Superstep 1 and during all the next odd-numbered supersteps, vertices represent items; items' latent vectors get trained. The training of the latent vectors for both users and items is the same and is presented in the Algorithm 2. Let us assume we are in an even-numbered superstep and a vertex represents a user. At the beginning of the superstep, the vertex reads the messages received. For each message, the vertex calculates a predicted rating based on its own latent vector and on the item's latent vector which is included in the message (line 2). It then calculates the error; the deviation of the predicted rating from the known rating (line 3). The known rating given from the user to the specific item is stored on the edge between them. Based on this error, the vertex recalculates its own latent vector aiming to the reduction of the error. In the recalculation, λ and γ are taken into consideration to avoid possible overfitting and control the learning rate as explained in Section 3.2.3.

Algorithm 2 SGD Computation for calculating a vertex latent vector

```
1: for message in received_ messages do
2:     $predictedRating \leftarrow dotProduct(latentVector, message.getLatentVector())$
3:     $err \leftarrow predictedRating - knownRating$
4:     $part1 \leftarrow \lambda * latentVector$
5:     $part2 \leftarrow err * message.getLatentVector()$
6:     $part3 \leftarrow -\gamma * (part1 + part2)$
7:     $latentVector \leftarrow latentVector + part3$
8: end for
```

Halt of Computation

Algorithm 2 is repeated for a vertex as many times as the number of messages received in the current superstep. At the end of the for-loop, the vertex sends its final latent vector to all its neighbours, which are the items rated by the user. Hence, in the next superstep, the vertices representing items go through the same algorithm to recalculate items' latent vectors. This interchange between users and items is repeated till the halting condition is met. Recalling from Section 3.2.2.2, the algorithm can halt depending on three different conditions: (a) after a maximum number of iterations, (b) when the RMSE calculated based on each vertex error is lower than a value specified by the user, (c) when the L2Norm calculated based on each vertex value is lower than a value specified by the user.

It is important to clarify that in each superstep, all vertices vote to halt. However, the computation really terminates when the second requirement is also met; that is, there are no messages in transit. The halting condition specifies whether a vertex will send a message to its neighbours or not. Hence, each vertex checks locally the halting condition. If all vertices vote to halt and none of them sends messages, then the algorithm ends. Halting condition (a) is easy to interpret; all vertices reach the same number of supersteps and therefore at the same superstep all vertices will not send any message. Conditions (b) and (c) are subjective and differ for each vertex. Every vertex checks locally if its local RMSE (for condition (b)) or L^2-Norm (for condition (c)) value is smaller than the user-defined value. It is highly probable - and expected - that not all vertices will reach a small RMSE or L^2-Norm at the same superstep or even at all. Therefore, the possibility of at least one vertex will fail to satisfy the halting condition is extremely high, and in this case the vertex will again send messages, causing its neighbours to wake up again. This can lead to an endless loop. Consequently, for both conditions (b) and (c) we also set the 'safety' halting condition (a) of reaching a maximum number of iterations.

Parameters of Computation

Let us now define the attributes of the SGD computation, their values, types and way of retrieval:

- **Vertex Id**: Users and Items Ids are given as integers in the input file, thus they are stored as `IntegerWritable` objects in the SGD code.
- **Vertex Value**: Users and Items values hold their latent vectors, i.e. vectors of some size which is given by the user as a command line parameter. For simplicity, the by-default size of the latent vectors is set to 2. In order to store a vector in a vertex value, we created a new Writable class, called `DoubleArrayListWritable`[1], which creates an array of `DoubleWritable` objects.
- **Edge Value**: An outgoing edge from a user to an item holds the known rating the user gives that item. Inversely, an outgoing edge from an item to a user holds the known rating the item was given

[1]The class `DoubleArrayListWritable` is located in the `es.tid.graphlib.utils` package.

by the user. Since the known rating is an integer

- **Message Value**: A message sent from a user to an item or vice-versa contains its new latent vector calculated in the current superstep, hence it contains an object of the class `DoubleArrayListWritable`.

5.1.4 Master Compute and Aggregator

Apart from the RMSE or L^2-Norm values calculated in each vertex, we were interested in having a global view of the RMSE for all vertices in each superstep. Therefore, we added the functionalities of a master compute and an aggregator. With its initialization, the master compute registers a non-persistent aggregator which receives and adds all the RMSE values calculated locally by the vertices in each superstep. To be precise, the value sent by each vertex to the aggregator is the initial state of the equation (3.3), and it is shown below (5.1).

$$RMSE_{init} = \sum_{u,i} (r_{ui} - \hat{r}_{ui})^2 \tag{5.1}$$

At the beginning of each superstep, the master compute calculates the global average RMSE value of all vertices which were active in the previous superstep. More precisely, it gets the aggregated value from the aggregator (5.2) - which holds the first part of the RMSE calculation for all vertices - and completes the equation as shown in (5.3).

$$AGG_{RMSE_{init}} = \sum_{\|r\|} (RMSE_{init}) \tag{5.2}$$

$$RMSE = \sqrt{\frac{1}{\|r\|}(AGG_{RMSE_{init}})} \tag{5.3}$$

5.1.5 Delta Caching: SGD Optimization

SGD is composed by a computation and a communication part. Bigger size of data implies more ratings, i.e. more edges and therefore higher traffic and communication overhead. Delta caching is an optimization to SGD that aims to decrease this communication overhead. The objective is to cache values that may be needed in the following supersteps, in order to avoid sending messages over the network with the same data. In SGD, messages only include the vertex value which is a latent vector of size defined by the user. Sending the vertex value can be omitted if the vertex value is already stored to the neighbours local memory and the value has not been changed from the previous superstep. Such an action brings a limitation to the storage needed in each vertex, however by empirical experimentation, delta caching does not lead to excessive additional storage. Delta caching is optional and can be enabled by the user.

5.1.6 Execution of the Algorithm

Giraph gives the ability to the programmer to pre-define the values of some parameters before the execution of the algorithm. For the SGD algorithm, the parameters offered for specification are:

- **`sgd.halt.factor`**: This parameter sets the halting condition. It receives one of the keywords: '*basic*', '*rmse*' or '*l2norm*' representing the three halting conditions listed in 3.2.2.2.
- **`sgd.halting.tolerance`**: The tolerance parameter is the value checked during the halting condition. If the halting condition is L2Norm, then the tolerance is the maximum L2Norm value that can reached and halt the execution. Similarly, with the RMSE as halting condition, the tolerance parameter holds the maximum tolerable value in order to reach termination.
- **`sgd.rmse.aggregator`**: This float-type parameter enables the aggregator to be executed during the algorithm. The by-default value is 0 which means that the aggregator is disable.
- **`sgd.delta.caching`**: This parameter takes a boolean value; *True* if delta caching is enabled, otherwise *False*, which is set by default.
- **`sgd.iterations`**: This parameter sets the number of iterations. It is used for the halting condition (a) as described in the subsection Halt of Computation in 5.1.3. By-default, the number of iterations is set to 10.
- **`sgd.lambda`**: Value for the regularization parameter. The by-default value is 0.01.
- **`sgd.gamma`**: Value for the learning rating. The by-default value is 0.005.
- **`sgd.vector.size`**: Size of the vertex latent vector, by-default set to 2.

- **`sgd.print.error`**: This parameter is used in the Output Format and decides whether the error will be printed in the output file or not. By-default, it is set to false, that means it is disable.
- **`sgd.print.updates`**: This parameter is used in the Output Format and decides whether the number of updates of each vertex latent vector will be printed in the output file or not. By-default, it is set to false, that means it is disable.
- **`sgd.print.messages`**: This parameter is used in the Output Format and decides whether the number of messages of each vertex latent vector will be printed in the output file or not. By-default, it is disable and set to false.

The user can set the computation parameters in the execution command line, otherwise they get initialized with a by-default value as given above. The command line is given in the Appendix A.1.

5.1.7 Restrictions and Trade-offs

During implementation we had to face and solve challenges, restrictions and trade-offs that appeared on the spot. For SGD, two small issues arose and solved successfully.

- **Handling the 'late' creation of items**
  The input dataset given to SGD is a directed graph; it includes ratings from users to items, hence each line begins with the User Id, and then the Item Id follows. The reversed edges with the Item Id first and the User Id to follow do not exist. Therefore, when the Input Format reads the input file, it creates only the users with their outgoing edges to link to the items they have rated. At the end of Superstep 0, users send messages to (non existed) items and at Superstep 1, items get created[2] lacking though their outgoing edges. Without the outgoing edges, items would not be able to send messages to the users, and the compute method would reach the end. It is mandatory that items should receive somehow the users Ids. Hence, we created a MessageWrapper that wraps together the message to be sent between supersteps with the Id of the vertex sending the message. Apart from creating the outgoing edges, the items also need to learn the ratings from their users. To solve this issue, at Superstep 0 users temporarily include the rating at the end of their latent vector, which is then stored in the outgoing edge of the items at Superstep 1.

[2]By definition of Pregel, a vertex is created if it is specified in the input file **or** if it receives a message.

- **Calculating the RMSE in the Master Compute**

  As described in 5.1.4, the RMSE calculated by the aggregator in the Master Compute uses the number of ratings from the input dataset. A rating is represented as two edges in the algorithm; one edge from the user to the item and vice versa. Thus, in order to retrieve this number, the Master Compute calls the method `getTotalNumEdges()` and divides it by 2 in order to omit the edges representing the same rating. However, as we described above, the edges from items to users are created during Superstep 1. Thus the Master Compute should not divide the returned number from the method `getTotalNumEdges()` for the Supersteps 0 and 1 - remember that the Master Compute runs at the beginning of each superstep.

5.2 Alternating Least Squares

ALS is another optimization algorithm that tries to minimize the error between the known and predicted ratings. It is iterative, performing matrix factorization on the latent vectors of users and items. On the input dataset, the algorithm iterates through the ratings and executes two operations; (i) It fixes the user latent vector and solves a least-square problem for computing the item latent vector. (ii) It fixes the item latent vector and solves a least-square problem for computing the user latent vector. By executing these two operations for each user-item pair and by repeating the executions for a number of iterations, it is expected to observe the error between the predicted and known ratings becoming small.

5.2.1 Input Format

Similarly to SGD, the input file consists of user-item pairs and their ratings. Each line has three values; user's Id, item's Id, the rating from user to item. We use the same Input Format[3] that we created for SGD.

5.2.2 Output Format

The output file is created by **`IntDoubleArrayHashMapTextVertexOutputFormat.java`**[4] and outputs information for each vertex as the SGD does. Each line includes the Vertex Id and the

[3]Input Format name: **`IntDoubleTextEdgeInputFormat.java`**

[4]The Output Format classes for SGD and ALS have the same name because they receive the same type of Vertex. However, they print different output depending on the algorithm, hence they are located in the package of the corresponding algorithm

Vertex Value as calculated in the last superstep. Optionally, the error value for each rating, the number of updates of the Vertex Value and the number of messages received from the Vertex can be also printed. The error value represents the RMSE, the L2Norm or the common error calculated by the dot product of the predicted and the known rating, and is chosen based on the halting condition set in the execution command line.

5.2.3 Main Computation

The ALS computation is conducted by vertices that represent both users and items. At Superstep 0 and during all the next even-numbered supersteps, vertices represent users. Similarly, at Superstep 1 and during all the next odd-numbered supersteps, vertices represent items. The training of the latent vectors for both users and items is the same and is presented in the Algorithm 3. At the beginning of the superstep, the vertex creates a two-dimension matrix `matN` to store all neighbours latent vectors and a one-dimension matrix `matR` to store the ratings from all its neighbours. It then reads the messages received and fills in the two matrices. Afterwards, the recalculation of the vertex value is executed, as the Algorithm 3 depicts. The `matIdentity` (line 5) is an Identity matrix of size `[vectorSize X vectorSize]`; with the value set to `1` in its diagonal fields and `0` in the rest.

Algorithm 3 ALS Computation for calculating a vertex latent vector

```
1: for message in received_ messages do
2:     $matN.fill(message.getValue())$
3:     $matR.fill(message.getRating())$
4: end for
5: $matTemp \leftarrow matIdentity * \lambda * numEdges$
6: $aMatrix \leftarrow matN * matN.transpose() + matTemp$
7: $vMatrix \leftarrow matN * matR$
8: $latentVector \leftarrow QRDecomposition(aMatrix).solve(vMatrix)$
```

In contrast to SGD, the recalculation of the vertex value is not affected by the previous error between the predicted and the known rating. Nor it recalculates the vertex value as many times as the amount of neighbours - which is the case in SGD. Instead, it is altered only once during a superstep by taking into account all neighbours latent vectors and all ratings at the same time. Following the recalculation, the vertex iterates through all its outgoing edges, predicts the rating for each connection using its new value and then calculates the error between the predicted and the real value of rating. Initially one may naively argue that this execution is ‘faster’ since the vertex value is calculated only once during a superstep. However, this implementation causes very high computational overhead due to the several matrix computations, which are known to be expensive.

Halt of Computation

Algorithm 3 is executed only once, while the the prediction for each rating and the error on the prediction is done afterwards in a separate `for-loop` where the vertex iterates through its outgoing edges. At the end of all the calculations, the vertex sends its final latent vector to all its neighbours. Hence, in the next superstep, the vertices will go through the same algorithm to recalculate their own latent vectors. This interchange between users and items is repeated till the halting condition is met. Exactly like is determined in SGD, the algorithm can halt depending on the same three conditions: (a) after a maximum number of iterations, (b) when the RMSE calculated based on each vertex error is lower than a value specified by the user, (c) when the L2Norm calculated based on each vertex value is lower than a value specified by the user. In every superstep, all vertices vote to halt and the computation actually terminates when no messages are in transit. To avoid repetition, we settle all details to be the same provided in SGD (5.1.3).

Parameters of Computation

The parameters of the ALS compute(), are the same as in SGD and listed below:

- **Vertex Id**: Users and Items Ids are given as integers in the input file, thus they are stored as `IntegerWritable` objects in the SGD code.

- **Vertex Value**: Users and Items values hold their latent vectors, i.e. vectors of some size which is given by the user as a command line parameter. For simplicity, the by-default size of the latent vectors is set to 2. In order to store a vector in a vertex value, we created a new Writable class, called `DoubleArrayListWritable`[5], which creates an array of `DoubleWritable` objects.

- **Edge Value**: An outgoing edge from a user to an item holds the known rating the user gives that item. Inversely, an outgoing edge from an item to a user holds the known rating the item was given by the user. Since the known rating is an integer

- **Message Value**: A message sent from a user to an item or vice-versa contains its new latent vector calculated in the current superstep, hence it contains an object of the class `DoubleArrayListWritable`.

[5]The class `DoubleArrayListWritable` is located in the `es.tid.graphlib.utils` package.

5.2.4 Master Compute and Aggregator

Similarly to SGD, the Master Compute class calls a non-persistent aggregator for calculating the global RMSE for all vertices in every superstep. The procedure followed in the Master Compute is identical to the one described in SGD 5.1.4.

5.2.5 Delta Caching: ALS Requirement

While in SGD, delta caching is an optional optimization 5.1.5, in ALS it becomes a requirement. By observing the way of recalculating the vertex value (3), one can notice the essentiality of having all neighbours latent vectors stored in a matrix. This automatically enables the idea of delta caching; that is, storing data that may be need later on. The first time a vertex receives messages, it stores the latent vectors from its neighbours. In the following supersteps, it updates only the latent vectors of the neighbours from which it receives messages, while the rest neighbours latent vectors remain as they are. Thereafter, for the recalculation of its own value, it utilizes all neighbours latent vectors as stored in the matrix. With delta caching, messages sent over the network and subsequently communication overhead decrease while more storage is in demand. This trade-off worth being taken as it is proven in the Evaluation (6).

5.2.6 Execution of the Algorithm

For the ALS algorithm, the parameters offered to be pre-defined by the user are:

- **`als.halt.factor`**: This parameter sets the halting condition. It receives one of the keywords: '*basic*', '*rmse*' or '*l2norm*' representing the three halting conditions listed in 3.2.2.2.
- **`als.halting.tolerance`**: The tolerance parameter is the value checked during the halting condition. If the halting condition is L2Norm, then the tolerance is the maximum L2Norm value that can reached and halt the execution. Similarly, with the RMSE as halting condition, the tolerance parameter holds the maximum tolerable value in order to reach termination.
- **`als.rmse.aggregator`**: This float-type parameter enables the aggregator to be executed during the algorithm. The by-default value is 0 which means that the aggregator is disable.
- **`als.iterations`**: This parameter sets the number of iterations. It is used for the halting

condition (a) as described in the subsection Halt of Computation of SGD in 5.1.3. By-default, the number of iterations is set to 10.

- **`als.lambda`**: Value for the regularization parameter. The by-default value is 0.01.
- **`als.vector.size`**: Size of the vertex latent vector, by-default set to 2.
- **`als.print.error`**: This parameter is used in the Output Format and decides whether the error will be printed in the output file or not. By-default, it is set to false, that means it is disable.
- **`als.print.updates`**: This parameter is used in the Output Format and decides whether the number of updates of each vertex latent vector will be printed in the output file or not. By-default, it is disable and set to false.
- **`als.print.messages`**: This parameter is used in the Output Format and decides whether the number of messages received from each vertex latent vector will be printed in the output file or not. By-default, it is disable and set to false.

The user can set the computation parameters in the execution command line, otherwise they get initialized with a by-default value as given above. The command line is given in the Appendix section A.2.

5.2.7 Restrictions and Trade-offs

During implementation of ALS we had issues to consider and solve. Below, we list the main issues we confronted.

- **Matrix Computation**
  The main issue during the implementation of the ALS algorithm was to find the cheapest way - in terms of time - to solve matrix computations. For the recalculation of a vertex value the following matrix computations occur: (i) matrix-matrix multiplication, (ii) matrix-matrix addition, (iii) construction and multiplication with a matrix transpose[6], (iv) construction and multiplication with an Identity matrix[7], (v) solve a linear equation `A * X = B`. Implementing this equations from scratch would cost time and add inefficiency to the algorithm. Instead, we used a linear

[6]The transpose of a matrix A is another matrix which reflects A over its main diagonal
[7]An Identity matrix of size n is a n x n square matrix with ones on the main diagonal and zeros elsewhere.

algebra library built in Java, called **jblas**[8]. jblas is known to be fast a very light library, therefore it had a very good fit and easy integration into our algorithm.

- **Handling the 'late' creation of items**
  As it is explained in the corresponding section of the SGD algorithm, items get created in the second superstep of the computation by receiving messages from the users they have rated them. The issue in this case is that items lack of the rating they get from the users. Subsequently, the recalculation of their latent vectors cannot be executed correctly. Therefore only in the first superstep of the execution, users include the rating with their latent vectors on the message they are about to send to the items. This issue is identical with the one from SGD 5.1.7 and it was solved in the same way.

- **Calculating the RMSE in the Master Compute**
  Similarly to the issue from SGD algorithm, the master is responsible for calculating the global RMSE; by aggregating the RMSE values from all vertices and dividing them by the number of edges. The latter value is retrieved from the number of edges that exist during computation. Since the edges exist twice - from user to item and from item to user - it is important to divide this number by two to omit repetition.

5.3 Dynamic Graph Partitioning Algorithm

The chosen graph partitioning algorithm is a dynamic algorithm that partitions a given graph into k-way balanced partitions while minimizing the number of cut-edges between partitions. It does not require a global view, rather just local information regarding a vertex, which makes it scale. Additionally, it supports dynamic graph changes by adapting with minimum cost. Beneath, we explain the three implementation parts and all details and trade-offs we came across. For this algorithm, we first introduce the description of the Master Compute and Aggregators. This is necessary in order to better understand the compute() later on.

[8]jblas source: `http://mikiobraun.github.io/jblas/`

5.3.1 Input Format

The input file consists of user-user pairs. Each line represents an edge from user Id1 to user Id2; a user Id can be of type Integer. For loading the graph into GraphLib, we wrote the Input Format called **`IntIntDefaultEdgeValueTextEdgeInputFormat.java`**. It is edge-centric and expects two values for the Ids, while it automatically sets the edge value to be -1, as it is not needed in this algorithm.

5.3.2 Output Format

The Output Format **`IntIntTextVertexOutputFormat.java`** is responsible to output information for each vertex. Each line begins with the Vertex Id. It then includes the Vertex Value as it was initialized in the first superstep of execution and as it was calculated in the last superstep. As we will further explain, the value of a vertex represents the partition Id in which the vertex is located. By printing the vertex value at the beginning and at the end of the execution, we could immediately know whether the vertex has changed partition or remained in the same one. A line also includes the Number of migrations, i.e. how many times the vertex has moved from one partition to another. Additionally, the Number of Local Edges and the Number of Total Edges of the vertex are printed. Local edges are the ones that link to another vertex in the same partition, while Total edges is the total number of edges of the specific vertex.

5.3.3 Master Compute and Aggregator

With its initialization, the Master Compute registers two sets of aggregators. Each set has as many aggregators as the number of partitions and are explained below:

- **CAPACITY_ AGGREGATORS**: This set consists of persistent sum aggregators; each one is responsible for one partition and its has the same Id as the specific partition. A vertex migrating from partition `p` to partition `q` sends a `-1` to aggregator `p` and a `+1` to the aggregator `q`. Thus each capacity aggregator `i` holds the number of users located in the partition `i` it represents. By having persistent aggregators, the counting is preserved throughout the execution and is changed depending on the messages sent by the vertices. Note that when a vertex sends a `+1` to an aggregator, it is also required to send a `-1` to another aggregator, and vice-versa.

- **DEMAND_ AGGREGATORS**: This set consists of non-persistent sum aggregators; each one is responsible for one partition and has the same Id as the specific partition. A vertex showing interest to migrate to partition `p` sends a `+1` to the corresponding demand aggregator `p`. Therefore, each demand aggregator `i` holds the number of vertices showed interest to migrate to partition `i` in the next superstep.

An additional aggregator for counting the total local edges from all vertices is registered. It is a non-persistent sum aggregator responsible to receive the local edges from each vertex and sum them in one value. This value is then printed by the Master Compute in each superstep. It is only used for debugging and observation of the increase of local edges.

5.3.4 Main Computation

The computation of the Partitioning algorithm consists of vertices that represent users. Superstep 0 is an initialization round; every vertex (i) initializes its value, which shows the partition Id in which it is located, (ii) sends a `+1` to the CAPACITY_ AGGREGATOR responsible for its partition, (iii) sends a message to all its neighbours advertising its own value. Thus, by the end of Superstep 0, all CAPACITY_ AGGREGATORS are initialized with the amount of users their partitions have and in the next superstep, all vertices know where their neighbours are located.

The actual computation begins at Superstep 1 and consists of two iterations:

1. Phase A - *Show interest to migrate*: The vertex declares an interest to migrate to another partition. This phase always occurs in an odd-numbered superstep. The main question is *"How does a vertex choose the partition?"*. Firstly, the vertex creates two empty Hash Tables; the first called `countNeighbours` holds pairs `<Partition Id, counter>` and the second called `partitionWeight` holds pairs `<Partition Id, interestWeight>`. The former keeps a track of the amount of neighbours exist in each partition, while the latter keeps a weight of interest which is calculated by the vertex. For each message received, the vertex fills in the first table either by adding an entry - if the key/Partition Id does not already exist - or by increasing the counter for the Partition hosting its neighbour. Then, the vertex proceeds to calculate the weight of interest for each partition following the steps depicted in Algorithm 4. Note that the `migrationProbability` is a randomly generated value between the range [0,1). If this

value is smaller than the user-defined parameter `PROBABILITY`, the vertex can proceed and show interest of migration, otherwise it remains in its current partition and Phase A is completed.

Algorithm 4 Calculation of weight of interest for migration to each partition

```
1: if migrationProbability $\leq$ PROBABILITY then
2:     for partition i in partitions do
3:         $load \leftarrow CAPACITY\_AGGREGATOR[i]$
4:         $totalNeighbours \leftarrow number\_of\_outgoing\_edges$
5:         $numNeighbours\_in\_i \leftarrow countNeighbours.get(i)$
6:         $weight \leftarrow (1 \; / \; load) * numNeighbours\_in\_i \; / \; totalNeighbours$
7:         $partitionWeight.put(i, \; weight)$
8:     end for
9: end if
```

Thereafter, the vertex chooses the partition with the highest weight as the ‘desired’ destination partition and declares interest of migrating to it by sending a message `+1` to the DEMAND_ AGGREGATOR responsible for the ‘desired’ partition. In order to avoid unnecessary migrations and extra communication overhead, we added a control to handle the case that two partitions have the highest weight and one of them is the current partition of the vertex. In this case, the vertex chooses to stay in its partition and it does not show interest to migrate. Phase A is completed and the vertex does not send any message.

2. Phase B - *Migrate*: The vertex may migrate to another partition. This phase always occurs in an even-numbered superstep. If a vertex has declared an interest to migrate in the previous superstep then it is allowed to go through this phase, otherwise it does not proceed with any additional computation and waits for the superstep to be completed. A vertex that did declared interest, proceeds with the calculation of a probability threshold, which is shown in Algorithm 5.

Algorithm 5 Calculation of probability to actually migrated to ‘desired’ partition

```
1: $load \leftarrow CAPACITY\_AGGREGATOR[desired\_partition]$
2: $availability \leftarrow TOTAL\_CAPACITY \; - \; load$
3: $demand \leftarrow DEMAND\_AGGREGATOR[desired\_partition]$
4: $threshold \leftarrow availability \; / \; demand$
```

Note that the CAPACITY is the total capacity of nodes a partition can host. Since it is not maintained throughout the supersteps, CAPACITY is calculated in every superstep by all vertices in the same way; divide the total number of nodes by the number of partitions and add some space for tolerance (5.4). Thus the rule of k-way balanced partitions is preserved in all supersteps of the execution.

$$CAPACITY \ = \ num\_Nodes \, / \, num\_Partitions \ + \ (num\_Nodes \, / \, num\_Partitions * 0.2) \tag{5.4}$$

Thenceforth, the vertex generates a random number in the range [0,1) and migrates if and only if this number is less than the calculated threshold. Migration includes: (i) sending a message `-1` to the CAPACITY_ AGGREGATOR of its current partition, (ii) sending a message `+1` to the CAPACITY_ AGGREGATOR of the partition to be migrated to, (iii) setting its value to the new partition, (iv) sending messages to its neighbours advertising its new host.

Halt of Computation

A parameter defined in the execution command line is the number of stabilization rounds. A stabilization round is a superstep in which the vertex had the opportunity to migrate but did not. If the vertex does not migrate for `x` consecutive supersteps - with `x` to be the user-predefined parameter - it is then allowed to vote to halt. Note that the counter for stabilization rounds counts consecutive stabilization supersteps. When the vertex migrates to another partition, this counter becomes zero. Reaching this stabilization threshold may be hard. It also varies from one vertex to another causing the phenomenon of vertices voting to halt but re-starting their execution because a neighbour sent them a message. Therefore, we also set as halting condition a number of iterations, after which all vertices vote to halt.

Parameters of Computation

The parameters of the Partitioning compute(), are as follows:

- **Vertex Id**: Users are given as integers in the input file, thus they are stored as `IntegerWritable` objects in the Partitioning compute method.
- **Vertex Value**: Users hold as value the partition Id in which they belong to; thus this parameter is an `IntegerWritable` object.
- **Edge Value**: An outgoing edge from a vertex holds the Vertex Value; its host partition Id, therefore it is a `IntegerWritable` object.
- **Message Value**: A message sent from a vertex contains its value in order to advertise to its neighbours the partition it belongs to. Similarly, this is an object `IntegerWritable`. Since its value

can only change in the second phase - which occurs only in even-numbered supersteps, vertices do not send any messages in the first phase, which takes place only in odd-numbered supersteps.

5.3.5 Execution of the Algorithm

For the Graph Partitioning algorithm, the parameters offered to be pre-defined by the user are:

- **`partitioning.num.partition`**: With this parameter, the user can pre-defined the number of partitions. By-default the parameter is set to 1. This float-type parameter enables the aggregator to be executed during the algorithm. The by-default value is 0 which means that the aggregator is disable.
- **`partitioning.stabilization`**: The stabilization parameter defines the number of consecutive stabilization rounds the vertex must ‘experience’ in order to be able to vote to halt. By-default, its value is set to 30.
- **`partitioning.iterations`**: This parameter sets the number of iterations and the halting condition if the maximum number of stabilization rounds is never reached. By-default, the number of iterations is set to 100.
- **`partitioning.probability`**: This float-type parameter sets the probability that a vertex would like to migrate and change partition. It ranges from 0 to 1 with 0 meaning the vertex will never migrate and 1 meaning the vertex will always show interest to migrate. Its by-default value is set to 0.5.

The user can set the computation parameters in the execution command line, otherwise they get initialized with a by-default value as given above. The command line is given in the Appendix section A.3.

5.3.6 Restrictions and Trade-offs

- **Partition Overload**

  The most serious issue we needed to face concerns the superstep in which vertices declare their interest to migrate to a partition. In the description of the algorithm, the authors omit this necessity which results by the following observation; vertices only have local information, that is, they do not know how many vertices are located in the same partition, neither how many of them want to migrate to the same destination partition. Subsequently, at the end of a superstep it is probable that many vertices from the same partition choose to go to the same partition causing overloading of the destination partition. Hence, controlling the migration is necessary. In order to front this problem, we introduce an additional superstep for vertices to declare their interest to move to a partition. If there is enough residual capacity in the destination partition, then all vertices who made a request, do migrate. Otherwise, they migrate depending on a probability shaped by the load in that partition and the number of neighbours vertices have in that partition.

- **Initialization of Vertices Values**
  The vertex value represents the Partition Id the vertex belongs to. A simple way to initialize each vertex value - and therefore put it in a partition - is the following:

$$Vertex\_Value \ = \ Vertex\_Id \, \% \, num\_Partitions \tag{5.5}$$

5.4 GraphLib Programming Model

GraphLib is developed on top of Giraph, therefore we chose the same implementation language; Java. We implemented GraphLib as a separate project under Giraph codebase. The first phase of this project included an extended study of the Giraph codebase, and experimentation with existing examples and simple problems. Once we became familiar with the programming style, structure and conventions of the Giraph project, we followed the same style in our implementation following the same style. Several Giraph classes are reused and new classes were added in the library wherever needed. Some wrappers were also created to solve compatibility issues when necessary.

In total, 6 Java packages and 30 new classes were created. Below, we briefly present the packages and their functionality:

- **`es.tid.graphlib.io.formats`**
  This package contains 7 classes for reading input files and writing output files. These are the so-called Input Format and Output Format classes. They typically extend abstract classes provided in the Giraph package `org.apache.giraph.io.formats` in order to specify the types of the attributes expected to have in the input and output files.

- **`es.tid.graphlib.utils`**

  This package contains 10 classes with code that facilitates our programming. It includes new Writable objects, like ArrayList Writable and HashMap Writable, which extend abstract classes from `org.apache.giraph.utils` and `org.apache.hadoop.io.Writable`. Additionally, classes representing objects of pairs are located in this package and they extend classes again from `org.apache.giraph.utils`. Last but not least, a message wrapper class is provided which wraps together the message with the Id of the source vertex.

- **`es.tid.graphlib.sgd`**

  This package contains 6 classes for the SGD algorithm, 5 of them are different versions of the SGD algorithm. It is recommended to use the `Sgd.java` code which is generic and includes all versions of the SGD algorithm. The Output Format class `IntDoubleArrayHashMapTextVertexOutputFormat` is also included in this package because it is personalized for the SGD algorithm. It is included in this package because it retrieves values calculated in the SGD and which are not part of the vertices attributes.

- **`es.tid.graphlib.als`**

  This package contains 2 classes used for the ALS algorithm. The `Als.java` contains the `compute()` method, while the `IntDoubleArrayHashMapTextVertexOutputFormat.java` is the Output Format code, personalized for the specific algorithm. The OutputFormat is kept in this package because it retrieves values calculated in the `Als.java` and are not part of the vertices attributes.

- **`es.tid.graphlib.partitioning`**

  This package contains 3 classes used for the partitioning algorithm. The `Partitioning.java` contains the `compute()` method, the `IntIntTextVertexOutputFormat` is the Output Format, personalized for the partitioning algorithm, and the `IntMessageWrapper` which is a message wrapper but again specific for this algorithm and it extends from `org.apache.hadoop.io.WritableComparable`.

- **`es.tid.graphlib.examples`**

  In this package we stored examples we ran while learning Giraph. It includes the Shortest Paths and PageRank algorithms from Giraph but with some changes and optimizations done as small exercises to become familiar with the project. Optimizations include the addition of aggregators to calculate some global values. The package also includes a simple code for counting incoming edges and a Master Compute code in order to learn how this feature is used in Giraph.

Summary

In this chapter we explored implementation details about GraphLib. A full description of the algorithms including parameters, code files and execution commands was given. Thereafter, an overview of the packages and structure of GraphLib was provided. In the next chapter, we present our evaluation on the algorithms and derived findings.

6 Evaluation

6.1 Introduction

Scalability and time efficiency of the algorithms are important characteristics that have to be evaluated. The former includes scalability in terms of dataset size and number of workers. The latter characteristic implies efficiency in terms of computation time and communication time. GraphLib currently addresses two different types of problems, thus the experiments slightly differ between the partitioning algorithm and the SGD and ALS algorithms.

6.2 Clusters Configuration

The debugging of the algorithms was done in a local machine with 64-bit Ubuntu. The evaluation was then conducted in two clusters at Telefonica:

- Cluster # 1

  This is a cluster with 8 nodes. The specifications of each node is given in the Table 6.1. All nodes have Ubuntu SMP as Operating System and Java(TM) SE Runtime Environment of version 1.6.0. The Hadoop version is 0.20.203.0 and the Hadoop configuration used for our experiments is:

  - 4GB of RAM per map task
  - 40 maximum concurrent tasks per node
  - 64MB HDFS block size

- Cluster # 2

  This is a homogeneous cluster with 5 machines. Each cluster node has 8GB of RAM, 8 cores with 2GHz each. All nodes have Red Hat as Operating System and Java(TM) SE Runtime Environment of version 1.6.0. The Hadoop version is 0.20.203.0. The Hadoop configuration used for our experiments is:

- 4GB of RAM per node
- 40 maximum concurrent tasks per node
- 64MB HDFS block size

Name	Cores	Core Speed	RAM
bd001	16	2.26Ghz	64GB
bd006	16	2.93Ghz	72GB
bd008	16	2.26Ghz	16GB
bd009	16	2.26Ghz	16GB
bd010	16	2.26Ghz	32GB
bd023	4	2.40Ghz	16GB
hdp-00	24	2.66Ghz	96GB
hdp-01	24	2.66Ghz	96GB

Table 6.1: Big Cluster Specifications

6.3 Datasets

It is of high importance to highlight that the evaluation was conducted on real-world datasets. For the Collaborative Filtering algorithms, the input file should give ratings from users to items. This can be represented as `<UserId, ItemId, Rating>`per line. Datasets from Netflix [28] with one million ratings/edges were initially used for debugging. For the evaluation of SGD and ALS, we used three datasets from MovieLens [34] with: (i) $\sim 100,000$ ratings, (ii) ~ 1 million ratings, (iii) ~ 10 millions ratings.

For the Partitioning algorithm, the input file should give edges, which can be represented as `<User1Id, User2Id>`per line. Datasets used in the evaluation come from: (i) Youtube with ~ 4 million edges [35], (ii) Live Journal with ~ 86 million edges [36], and (iii) Tuenti with ~ 1.6 billion edges [37].

6.4 Evaluation Criteria

The execution time is one of the most important metrics to be evaluated in our experiments. It is expected this metric to be highly depended on the amount of workers and the datasets size. Other evaluation criteria differ among the algorithms. For the SGD and ALS, we evaluate how the number of messages increases while setting a more strict RMSE target. For the partitioning algorithm, we consider the changes in locality percentage with different number of iterations and partitions.

6.5 Collaborative Filtering

Both SGD and ALS were evaluated based on the same experiments, parameters and settings. Throughout all experiments the λ was set to `0.1` which is empirically a good regularization value. In SGD, the γ was set to `0.005` which was decided after some experimentation.

6.5.1 Scalability in terms of Dataset Size

The first experiment evaluates the algorithms scalability on different dataset size. As illustrated in Figure 6.1, the datasets have sizes of 100K, 1M and 10M ratings and are shown on the x-axis in logarithmic scale. The y-axis represents the execution time in seconds. It is easy to deduce that SGD scales better than ALS. ALS scalability is negatively affected by the matrix computation which is typically a heavy procedure.

6.5.2 Scalability in terms of Number of Workers

Another way to evaluate the algorithms scalability is to test them with different number of workers. For this experiment we run the algorithms SGD and ALS with 1, 2, 4, 8, 16, 32, 64 and 128 workers. Figures 6.2(a) and 6.2(b) demonstrate the SGD and ALS execution time with different number of workers, respectively. On the dataset of 1M ratings, SGD scales up to 4 workers. With the bigger dataset of 10M though, it is more clear that SGD scales up to 16 workers. From 32 workers and more the execution time increases because the communication overhead exceeds the computation time. On the other hand, ALS has a heavy computation part, thus it is clear that ALS scales even with the small dataset of 1M. By

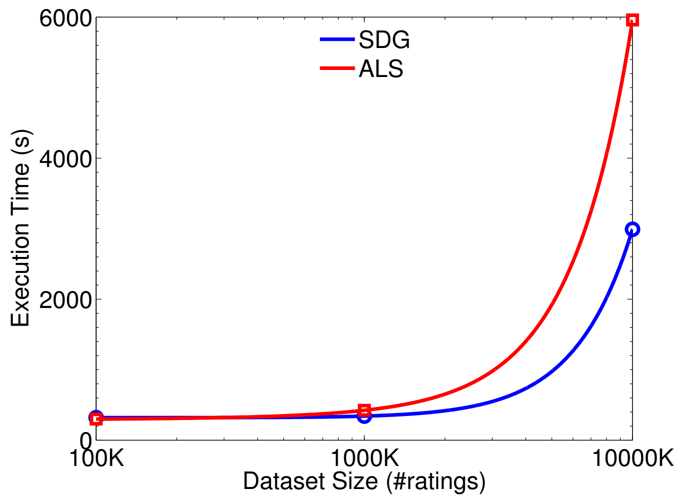


Figure 6.1: Execution Time based on Dataset Size: SGD and ALS

adding more workers, it helps partitioning the computation in more machines while the communication overhead does not significantly exceed the computation time. Though, with the 10M dataset, 64 workers seems to be the ideal amount of workers for processing the graph dataset. With more workers the communication time exceeds the computation time, causing additional overhead.

6.5.3 RMSE Convergence Rate with Different Features enabled

In this set of experiments, we would like to observe how fast the RMSE calculated by the master compute decreases with different features enabled. It is shown in previous works, that a *'good'* RMSE value lies around `1.3`. We run SGD for 40 iterations with three different settings:

- **Basic**: In this SGD version, vertices always send their value to all their neighbours and halt after 40 iterations.
- **L2Norm**: Every vertex calculates its L2Norm at the end of each superstep and sends a message to its neighbours if the calculated value is less than `0.5`. The vertex does not use any mechanism to check if it received a message from all its neighbours, hence the calculation of its value is solely done with the messages received.
- **L2Norm & Delta Caching**: This is the same as L2Norm but with Delta Caching enabled. Delta Caching is a technique for storing neighbours values locally - inside the vertex. With this capability, it is ensured that the vertex will calculate its value considering all its neighbours values, independently of receiving a message from them.

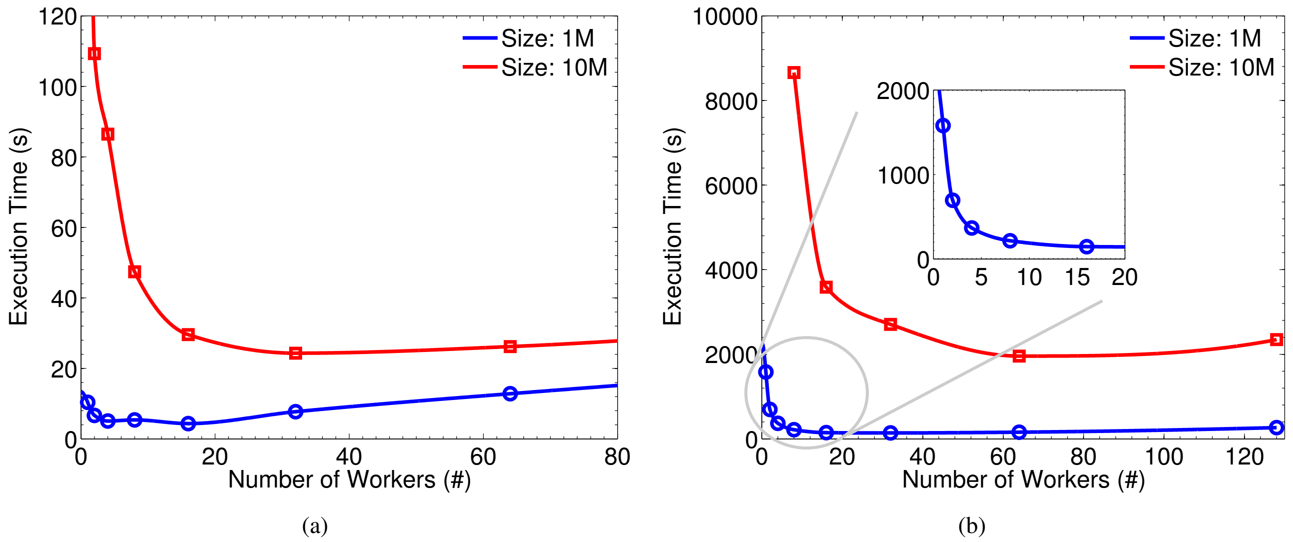
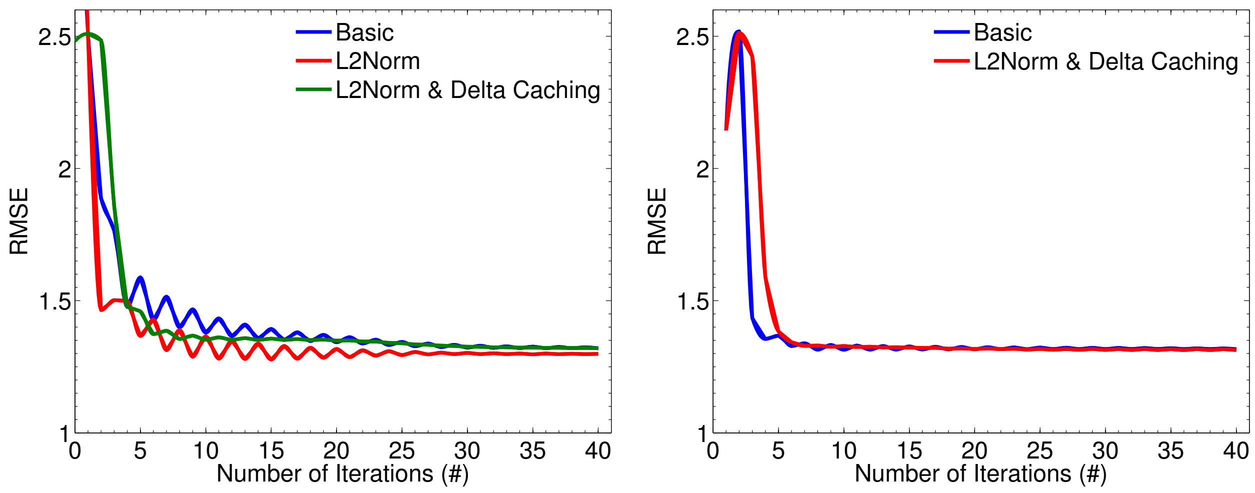


Figure 6.2: Execution Time based on Number of Workers: (a) SGD, (b) ALS

In ALS, we run the versions **Basic** and **L2Norm & Delta Caching**. L2Norm without Delta caching is not possible as explained in the Implementation section.

As it is shown in Figure 6.3(a), the RMSE drops abruptly from 2.5 to 1.5 in less than 5 iterations. The L2Norm version converges faster and better than the other two versions. A possible reason would be that the vertex value - subsequently the RMSE value - is calculated every time only with the neighbours values that sent a message to the vertex, which implies that the value is calculated considering the neighbours values that have changed to a significant extend. Taking this thought a step further, we can conclude that considering the vertices values that have not significantly changed may prevent the RMSE to reach a small value fast.

In ALS, the RMSE converges differently. As illustrated in Figure 6.3(b), the RMSE converges to 1.3 in the first 5 iterations. Later on, it is shown to be 'smoother'. This is easily justified because the calculation of the vertex value is done only once in each superstep by considering the latent vectors by all its neighbours. With a more accurate vertex value, the predicted rating is more accurate, the error between the predicted and known rating is smaller and, subsequently, the RMSE is gets smaller and more consistent than then one in SGD.



(a) (b)

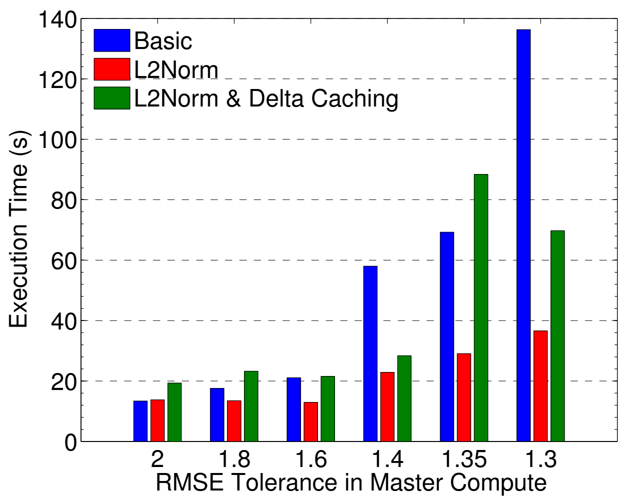
Figure 6.3: RMSE convergence rate with Different Features Enabled: (a) SGD, (b) ALS

6.5.4 Impact of different RMSE Targets

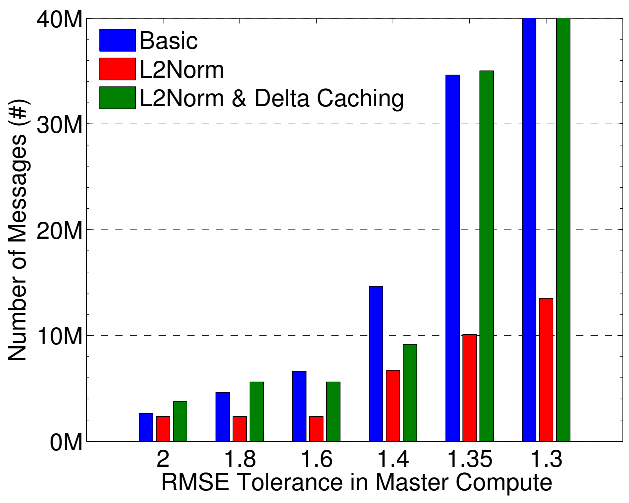
Our next experiments concern the impact of different RMSE targets in the Master Compute, on the execution time and number of messages sent during computation. We run the three versions of SGD and the two versions of ALS with RMSE target: 2, 1.8, 1.6, 1.4, 1.35 and 1.3. Results are plotted in Figures 6.4(a) and 6.4(b) for the SGD and in Figures 6.5(a) and 6.5(b) for the ALS.

For SGD, the L2Norm version always completes faster because the vertex value calculation is executed only as many times as the number of messages received. L2Norm with Delta Caching takes more time than the simple L2Norm because (i) as observed, it needs a couple of additional iterations to reach RMSE target, and (ii) the vertex calculates its value as many times as the number of **all** its neighbours. Because of the additional iterations needed to converge, more messages are sent than in the simple L2Norm. From the side of the RMSE target, the values 2, 1.8 and 1.6 are very high and as a result the execution time as well as the number of messages for each SGD version is the same because they reach all three targets in 2 supersteps. Moreover, the RMSE target 1.3 is never reached for the duration of the 40 iterations, thus the execution time shown in the plot represents the time for the completion of all 40 iterations and the messages sent actually reach the 78 million but were omitted from the plot for better view of the rest of the results.

Similarly to SGD, ALS gives the same execution time for RMSE targets 2, 1.8 and 1.6, while the 1.3 is never reached in the 40 iterations. The actual execution time for the 1.3 target is 176 seconds and the



(a)



(b)

Figure 6.4: SGD: Impact of different RMSE target on: (a) Execution Time, (b) Number of messages

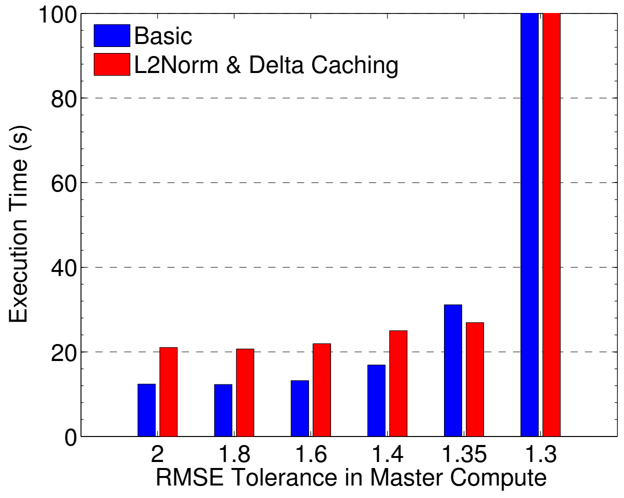
number of messages sent are about 78 million. They are also omitted from the plot in order to better show the rest of the results. ALS appears to converge faster than SGD, reaching the 1.35 RMSE target in 30 seconds for both versions while in SGD only the L2Norm version reaches 1.35 in 30 seconds. The other two versions require more than double time - and therefore messages - to reach 1.35.

6.6 Graph Partitioning

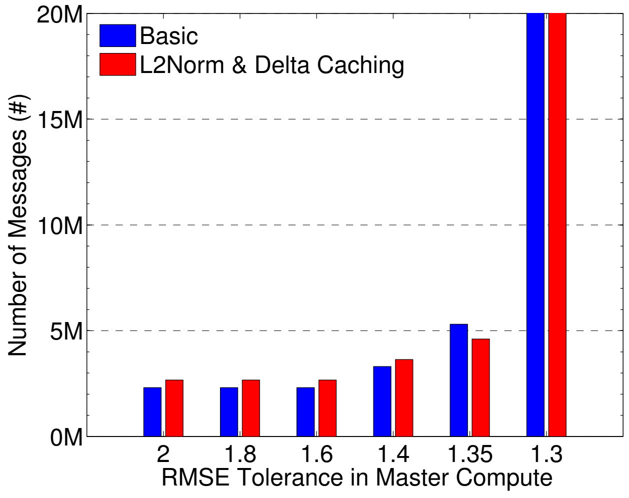
The graph partitioning algorithm is evaluated in order to observe its scalability in terms of dataset size and number of workers. Additionally, we evaluate the resulted locality of the nodes by the end of the execution of the algorithm based on the number of iterations and partitions. For the execution of the graph partitioning algorithm, we set a parameter to have the same value in all scenarios; that is the probability parameter. This value defines how possible is for a vertex to show interest for migration and it ranges between [0,1]. For the evaluation of the algorithm, this parameter is set to `0.8`.

6.6.1 Scalability in terms of Dataset Size

In our first experiment with the graph partitioning algorithm, we evaluate its scalability with various dataset size. 64 workers are used and 64 partitions are set to be the resulting shape of the graph. The algorithm ran for 100 iterations in all datasets. Figure 6.6 illustrates the datasets with sizes of 4 millions,



(a)



(b)

Figure 6.5: ALS: Impact of different RMSE target on: (a) Execution Time, (b) Number of messages

86 millions and 1.6 billions edges on the X axis in logarithmic scale. The Y axis represents the execution time in seconds. The graph shows that the partitioning algorithm scales satisfactorily.

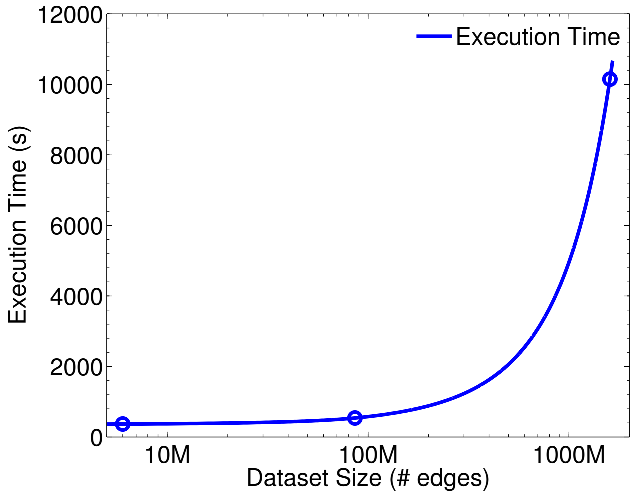


Figure 6.6: Partitioning: Execution Time based on Dataset Size

6.6.2 Scalability in terms of Number of Workers

The second part of the scalability evaluation concerns the change in execution time with different number of workers. The settings for this experiment are the following: (i) dataset size: 86 millions edges, (ii) number of iterations: 100, (iii) number of partitions: 64, (iv) migration probability: 0.8. The scenario is repeated for 1, 2, 4, 8, 16, 32, 64 and 128 workers. As depicted in Figure 6.7, the algorithm improves in execution time with the increase of the workers till it reaches the 64 workers, which is shown to be the optimal amount of workers. More workers cause the communication overhead to surpass the computation time and cause delay in the completion of the algorithm.

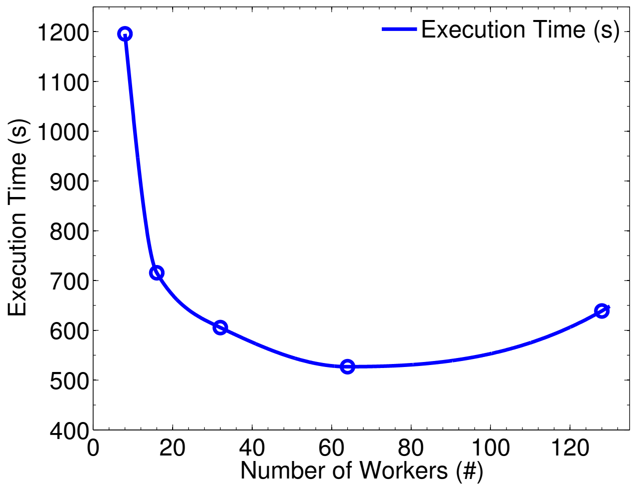


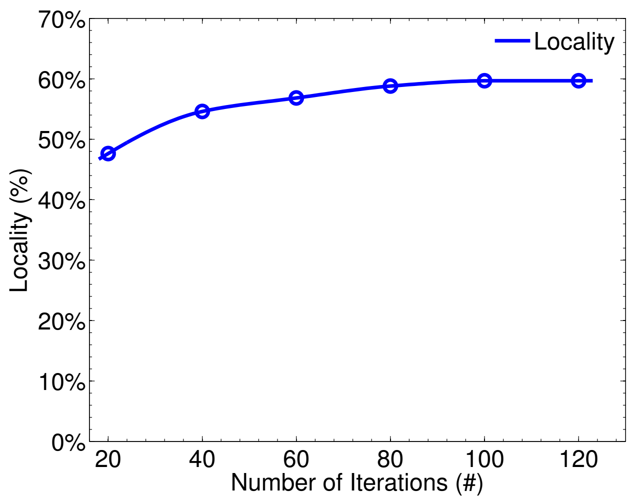
Figure 6.7: Partitioning: Execution Time based on Number of Workers

6.6.3 Locality Results

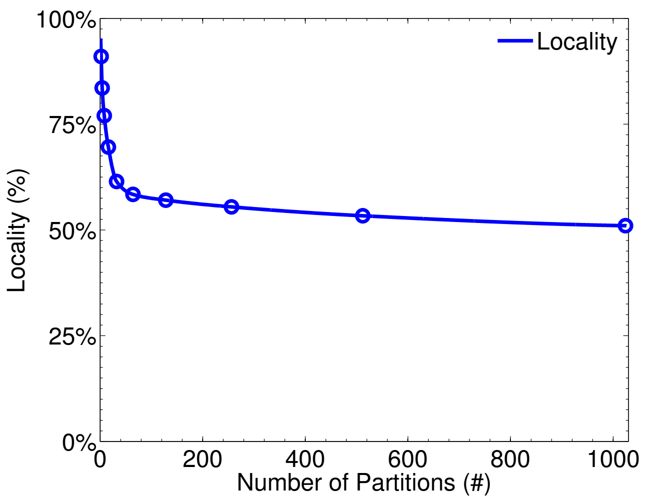
The partitioning algorithm aims at the minimization of cut-edges, which means the increase of local edges connecting co-located nodes. In order to measure this characteristic we conduct two experiments: the first experiment executes the algorithm with different number of iterations; 20, 40, 60, 80, 100 and 120. The second experiment executes the algorithm with different number of partitions; 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024. For both sets of experiments, the following settings are used; (i) dataset size: 86 millions edges, (ii) number of workers: 16. For the former experiment, the number of partitions is set to 64 and for the latter experiment the number of iterations is set to 100.

Figures 6.8(a) and 6.8(b) present the results from this evaluation. With the increase of iterations, the locality improves from 47.64% in the 20 iterations to 59.67% in the 120 iterations. The local edges, from 100 to 120 iterations, increase only by 1000, thus it can be cautiously deduced that 100 iterations are enough for executing this algorithm in case that execution time is a constraint.

In the plot on the right, it is shown how locality is affected with the addition of more partitions. While at the beginning there is a sharp drop of the percentage of locality, the algorithm manages to keep this



(a)



(b)

Figure 6.8: Locality Percentage based on Number of: (a) Iterations, (b) Partitions

percentage above 50% for all experiments. This percentage is higher than many other partitioning algorithms with the additional advantages of being scalable for large datasets.

Summary

This chapter described the process we followed to carry out the experiments. We listed all information regarding the environment of experimentation, the datasets used and metrics evaluated. We have discussed and shown that all three algorithms can scale and be time-efficient. This is attributable to the scalable distributed model, Pregel.

7 Conclusions

Large scale graph processing is becoming an increasing trend and an essential requirement in systems dealing with big data. MapReduce, a programming model for large scale data processing has become very successful and widely adopted by many industrial entities. However, MapReduce fails to efficiently process either graph algorithms or iterative algorithms. Pregel - a Bulk Synchronous Parallel model - makes its appearance for processing very large graphs. Based on this model Apache Giraph is developed, an open-source project in which large scale graph processing algorithms can be implemented. So far, no library exists with scalable graph processing algorithms. Hence, we take this opportunity to develop *GraphLib*, a library built on top of Apache Giraph with scalable graph processing algorithms, based on the Pregel model.

In this thesis project, we studied Apache Giraph, as well as various graph areas and problems that exist out there. We specified GraphLib to focus on problems from Collaborative Filtering and Graph Partitioning, which form two of the most crucial graph areas with several works to be in progress around them. It consists of two algorithms concerning Collaborative Filtering; the Stochastic Gradient Descent algorithm and the Alternating Least Squares algorithms, both of which are optimization algorithms aiming at the reduction of errors in predictions. Moreover, GraphLib includes one dynamic graph partitioning algorithm. This algorithm achieves k-way balanced partitions of an input graph while minimizing the cut-edges that link nodes from different partitions.

Apart from the design of GraphLib and implementation of the algorithms, we present an extensive experimentation performed for evaluation of our work. Results show that all three algorithms significantly scale in terms of dataset input size and number of workers in use.

7.1 Future Work

The development of a library with scalable graph processing algorithms is certainly just the beginning for more work to come. Several algorithms can be added in both categories of Collaborative Filtering and Graph Partitioning, as well as algorithms can be developed to confront new categories of problems. We aim to contribute this library to the open-source community of Giraph believing that these algorithms can be of great interest for other scientists to use and extend. Furthermore, a more extensive and wider evaluation of the algorithms could help us identify whether Giraph performs better for a specific category of problems and take advantage of possible foundings.

A Execution Command Lines

Command Line A.1: Stochastic Gradient Descent

```
hadoop jar /directory-to-graphlib/target/
        graphlib-1.0-SNAPSHOT-jar-with-dependencies.jar \
org.apache.giraph.GiraphRunner es.tid.graphlib.sgd.Sgd \
-eif es.tid.graphlib.io.formats.IntDoubleTextEdgeInputFormat \
-eip /directory-in-HDFS/input_file \
-of es.tid.graphlib.sgd.IntDoubleArrayHashMapTextVertexOutputFormat \
-op /directory-in-HDFS/output_file \
-w 16 \                                 # Number of workers
-ca giraph.zkManagerDirectory = /user/marsty5/_bsp \
-ca giraph.useSuperstepCounters = false \
-ca mapred.child.java.opts = -Xmx2048m \
-mc es.tid.graphlib.sgd.Sgd\$MasterCompute \
-ca sgd.rmse.aggregator = 1.3 \
-ca sgd.halt.factor = "l2norm" \        # Options: "basic", "rmse", "l2norm"
-ca sgd.halting.tolerance = 0.0005 \
-ca sgd.delta.caching = true \
-ca sgd.iterations = 100 \
-ca sgd.vector.size = 2 \
-ca sgd.lambda = 0.1 \
-ca sgd.gamma = 0.005 \
-ca sgd.print.error = true \            # Used in the Output Format
-ca sgd.print.updates = true \          # Used in the Output Format
-ca sgd.print.messages = true           # Used in the Output Format
```

Command Line A.2: Alternating Least Squares

```
hadoop jar /directory-to-graphlibs/graphlib/target/
        graphlib-1.0-SNAPSHOT-jar-with-dependencies.jar \
org.apache.giraph.GiraphRunner es.tid.graphlib.als.Als \
-eif es.tid.graphlib.io.formats.IntDoubleTextEdgeInputFormat \
-eip /directory-in-HDFS/input_file \
-of es.tid.graphlib.als.IntDoubleArrayHashMapTextVertexOutputFormat \
-op /directory-in-HDFS/output_file \
-w 16 \                                    # Number of workers
-ca giraph.useSuperstepCounters = false \
-ca mapred.child.java.opts = -Xmx2048m \
-mc es.tid.graphlib.als.Als\$MasterCompute \
-ca als.rmse.aggregator = 0 \
-ca als.halting.factor = "basic" \        # Options: "basic", "rmse", "l2norm"
-ca als.halting.tolerance = 1 \
-ca als.iterations = 10 \
-ca als.vector.size = 2 \
-ca als.lambda = 0.1 \
-ca als.print.error = true \               # Used in the Output Format
-ca als.print.updates = true \             # Used in the Output Format
-ca als.print.messages = true              # Used in the Output Format
```

Command Line A.3: Graph Partitioning Algorithm

```
hadoop jar /directory-to-graphlib/target/
        graphlib-1.0-SNAPSHOT-jar-with-dependencies.jar \
org.apache.giraph.GiraphRunner es.tid.graphlib.partitioning.Partitioning \
-eif es.tid.graphlib.io.formats.IntIntDefaultEdgeValueTextEdgeInputFormat \
-eip /directory-in-HDFS/input_file \
-of es.tid.graphlib.partitioning.TextIntIntVertexOutputFormat \
-op /directory-in-HDFS/output_file \
-w 64 \
-ca giraph.useSuperstepCounters = false \
-ca mapred.child.java.opts = -Xmx2048m \
-mc es.tid.graphlib.partitioning.Partitioning\$MasterCompute
-ca partitioning.num.partition=3 \
-ca partitioning.iterations=100 \
-ca partitioning.probability=0.9
```

References

[1] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, “Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters,” *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[2] M. Newman and J. Park, “Why social networks are different from other types of networks,” *Physical Review E*, vol. 68, no. 3, p. 036122, 2003.

[3] M. Newman, “Modularity and community structure in networks,” *Proceedings of the National Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[4] F. Ricci and B. Shapira, *Recommender systems handbook*. Springer, 2011.

[5] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, “Large-scale parallel collaborative filtering for the netflix prize,” in *Algorithmic Aspects in Information and Management*, pp. 337–348, Springer, 2008.

[6] C. M. Luis M. Vaquero, Felix Cuadrado and D. Logothetis, “Adaptive partitioning for large-scale dynamic graphs,” tech. rep., Queen Mary University of London, School of Electronic Engineering and Computer Science, 2013.

[7] C. Borgelt, “Graph mining: An overview,” in *Proc. 19th GMA/GI Workshop Computational Intelligence*, pp. 189–203, 2009.

[8] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] D. Borthakur, “Hdfs architecture guide,” *Hadoop Apache Project*. `http://hadoop.apache.org/common/docs/current/hdfs_design.pdf`, 2008.

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

[11] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, “Graphlab: A new

framework for parallel machine learning," *arXiv preprint arXiv:1006.4990*, 2010.

[12] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[13] "Research blog: Large-scale graph computing at google." http://googleresearch.blogspot.com.es/2009/06/large-scale-graph-computing-at-google.html. Online; Last accessed 15-May-2013.

[14] "Distributed graph processing with giraph." http://giraph.apache.org/, 2013. Online; Last accessed 15-May-2013.

[15] "goldenorb." http://goldenorbos.org/. Online; Last accessed 15-May-2013.

[16] "Phoebus." https://github.com/xslogic/phoebus. Online; Last accessed 15-May-2013.

[17] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Scientific and Statistical Database Management*, Stanford InfoLab, July 2013.

[18] "The apache hama project." http://hama.apache.org/, 2013. Online; Last accessed 15-May-2013.

[19] "Apache mahout." http://mahout.apache.org/, 2013. Online; Last accessed 15-May-2013.

[20] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[21] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pp. 229–238, IEEE, 2009.

[22] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," *Journal of Grid Computing*, vol. 10, no. 1, pp. 47–68, 2012.

[23] P. Stutz, A. Bernstein, and W. Cohen, "Signal/collect: Graph algorithms for the (semantic) web," in *The Semantic Web–ISWC 2010*, pp. 764–780, Springer, 2010.

[24] "Spark: Lighting-fast cluster computing." http://spark-project.org. Online; Last accessed 15-May-2013.

[25] "Bagel." https://github.com/mesos/spark/wiki/Bagel-Programming-Guide. Online; Last accessed 15-May-2013.

[26] "metis." http://glaros.dtc.umn.edu/gkhome/views/metis. Online; Last accessed 15-May-2013.

[27] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[28] "Netflix prize." http://www.netflixprize.com/. Online; Last accessed 15-May-2013.

[29] R. G. Wijnhoven *et al.*, "Fast training of object detection using stochastic gradient descent," in *Pattern Recognition (ICPR), 2010 20th International Conference on*, pp. 424–427, IEEE, 2010.

[30] R. Salakhutdinov and A. Mnih, "Probabilistic matrix factorization," *Advances in neural information processing systems*, vol. 20, pp. 1257–1264, 2008.

[31] S. Funk, "Netflix update: Try this at home." http://sifter.org/{~}simon/journal/20061211.html, Nov. 2006.

[32] A. Paterek, "Improving regularized singular value decomposition for collaborative filtering," in *Proceedings of KDD cup and workshop*, vol. 2007, pp. 5–8, 2007.

[33] Y. Koren, "Factorization meets the neighborhood: a multifaceted collaborative filtering model," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 426–434, ACM, 2008.

[34] "Grouplens research." http://www.grouplens.org/node/73. Online; Last accessed 15-May-2013.

[35] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, vol. abs/1205.6233, 2012.

[36] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, vol. abs/1205.6233, 2012.

[37] "tuenti." https://www.tuenti.com. Online; Last accessed 15-May-2013.