A Survey on Factorization Methods in MapReduce Environment

M. Rajathi
Department of Computer Applications
Madurai Kamaraj University
Madurai, India

M. Ramaswami
Department of Computer Applications
Madurai Kamaraj University
Madurai, India

***Abstract* - Big data is a term encompassing complex types of large datasets that is hard to process with the traditional data processing systems. Innumerable challenges are encountered with big data like storage, transition, visualization, searching, analysis, security and privacy violations and sharing. Parallelism is a computational mechanism used to process such a large amount of data in an inexpensive and more efficient way. Hadoop is the core platform for handling massive data and it runs applications using the Map Reduce algorithm, where the data is processed in parallel on different CPU nodes. MapReduce is a software framework for applications which process vast amount of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. Due to the increasing availability of massive data sets in the form of matrices, researchers are facing the problem because the matrices which are to be factorized are having dimensions in the orders of millions. Recent research has shown that it is possible to factorize such large matrices within tens of hours using the MapReduce distributed programming platform. In this paper, we discuss two different matrix decomposition implementations using MapReduce; QR factorization and the SVD are two fundamental matrix factorization methods with applications around scientific computing and data analysis. The QR method decompose the matrix into partitions and by applying multiple processes to compute the QR decomposition in parallel provides decomposition process much faster than computing the QR decomposition on the original matrix. In the same way, Singular value decomposition (SVD) shows strong vitality in the area of information analysis and has significant application value in many scientific big data fields. For a large-scale matrix, applying SVD computation directly is both time consuming and memory demanding. To speed up the computation of SVD, a MapReduce model has many advantages over a message passing interface model, such as fault tolerance, load balancing and simplicity. This survey paper discusses different QR and SVD factorization methods and how they are implemented in MapReduce environment.**

Keywords: Big Data, Hadoop, MapReduce, Factorization

1. INTRODUCTION

The term “Big Data” refers to very large and complex data sets made up of a variety of structured and unstructured data which are too big, fast, and hard to be managed by conventional techniques. Big Data is characterized by the 4Vs(Dr.R.Gunavathi & P.Sudha, September 2016) volume, velocity, variety, and veracity. Volume refers to the quantity of data, variety refers to the diversity of data types, velocity refers both how fast data are generated and processed, and veracity is the ability to trust the data to be accurate and reliable when making crucial decisions.

Big Data is data with diversity and complexity requires new architecture, analytics, techniques and algorithms to manage and extract value, hidden knowledge from it. Traditional databases analytics says what is happened and however gives the predictive analysis of what is likely to happen in future. Infrastructure requirements of big data are data acquisition, data organization and data analysis. Hadoop is the open source software founded by Apache (A.P.S.Aslin & D.Usha, April 2014) for processing large datasets. Hadoop architecture includes a fault-tolerant storage system called Hadoop Distributed File System or HDFS. HDFS is capable of storing huge amounts of information, scale up gradually and survive the failure of significant parts of the storage infrastructure without losing data. Hadoop creates inexpensive machine clusters and coordinates to work among them. Hadoop continues to operate the cluster without losing data or interrupting work, by shifting work to the

remaining clusters even if one fails. HDFS manages storage on the cluster by breaking incoming files into pieces called “blocks” and storing each of the blocks redundantly across the pool of servers. In general, HDFS stores three complete copies of each file by copying each piece to three different servers(Gadekar & P S Bhosale, October 2014).

Map Reduce(Dean & Ghemawat, 2004),(HuangLan, Xiao-wei, ZhaiYan-dong, & Bin, September 24 - 26, 2009) is a simplified programming model and is a major component of Hadoop for parallel processing of large amount of data. It helps the programmers free from the parallelization issues while allowing them to concentrate on application development. The Hadoop architecture has shown in Figure 1. Two important data processing functions contained in MapReduce programming are Map and Reduce. The input data will be given into the Map phase which performs processing as per the program coded by the programmers to generate intermediate results. Parallel Map tasks will run at a time. First, the input data is divided into fixed sized blocks on which Map tasks run in parallel. The output of this stage is a collection of <key, value> pairs which is still an intermediate output. These <key,value> pairs undergo a shuffling phase across reduce tasks. Only one key is accepted by each reduce task and based on this key the processing will be done. The output will be again in the form of <key, value>pairs.

The Hadoop MapReduce framework consists of one master node and many worker nodes called as Job Tracker and Task Trackers. The submitted user jobs are given as input to the Job Tracker which transforms them into as many Map and Reduce tasks. These tasks are then assigned to the Task Trackers. The TaskTrackers in turn scrutinizes the execution of these tasks and the user is notified about job completion, when all tasks are accomplished. HDFS provides for fault tolerance and reliability by storing and replicating the inputs and outputs of a Hadoop job.

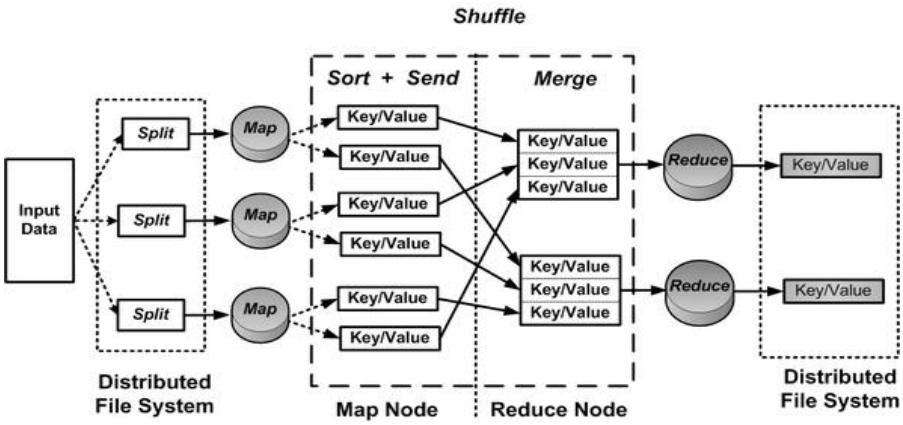

Figure 1. Hadoop Architecture

2. FACTORIZATION METHODS

Factorization (Decomposition) techniques used for many purposes, including solving a linear system more efficiently. Factorization methods decompose given matrix into two pieces that are easy to find the solution in efficient manner than inverting the matrix. More over this strategy improve the speed and/or accuracy of algorithms and also provides additional insights into the properties of the matrix in it. The QR factorization and the SVD are two fundamental matrix decompositions with applications throughout scientific computing and data analytics. The implementation of factorization method in MapReduce environment further enhance speed of decomposition process by partitions the matrix and compute the decomposition in parallel.

2.1. QR Factorization

The QR method factorizing given matrix A into product of two sub matrices Q and R, A = QR where Q is orthogonal matrix (or semi-orthogonal, if A is not square) and R is upper triangular matrix. The matrix Q is orthogonal if$Q^T Q = I$, where I is the identity matrix(Benson, Gleich, & Demmel, 2013) and the matrix R is

upper triangular, i.e., all entries below the main diagonal are 0. The implementation details of major QR factorization methods under MapReduce environment are discussed below:

2.1.1. Cholesky-QR Factorization

The Cholesky factorization method decompose a symmetric, positive and definite real-valued matrix A into product of two sub matrices L and LT, where L is an n × n lower triangular matrix. The Cholesky factor L for the matrix ATA is exactly same as the matrix R in the QR factorization. Since R is upper triangular matrix and L is unique, RTR =LLT. The method of computing R via the Cholesky decomposition of AT A matrix is called Cholesky-QR. Thus, the problem of finding R becomes the computing of ATA. This task is straightforward in MapReduce paradigm as shown in Figure 2. In the map stage, each task collects rows in the form of <key, value>pairs – to forma local matrix $A_i$and then computes $A_i^T A_i$. Thesematrices are small, n × n, and are output by row. In fact, $A_i^T A_i$is symmetric, and there are ways to reducethe computation by utilizing this symmetry. In the reduce stage, each individual reduce function takes in multiple instances of each row of ATA from the mapper. These rows are summed to produce a row of ATA. Formally, this method computes:$A^T A = \sum_{i=1}^{\#(maptasks)} A_i^T A_i$where Ai is the input to each map-task. Extending the ATA computation to Cholesky-QR simply consists of gathering all rows of ATA on one processor and serially computing the Cholesky factorization ATA = LLT .The serial Cholesky factorization is fast since ATA is small, n×n matrix.

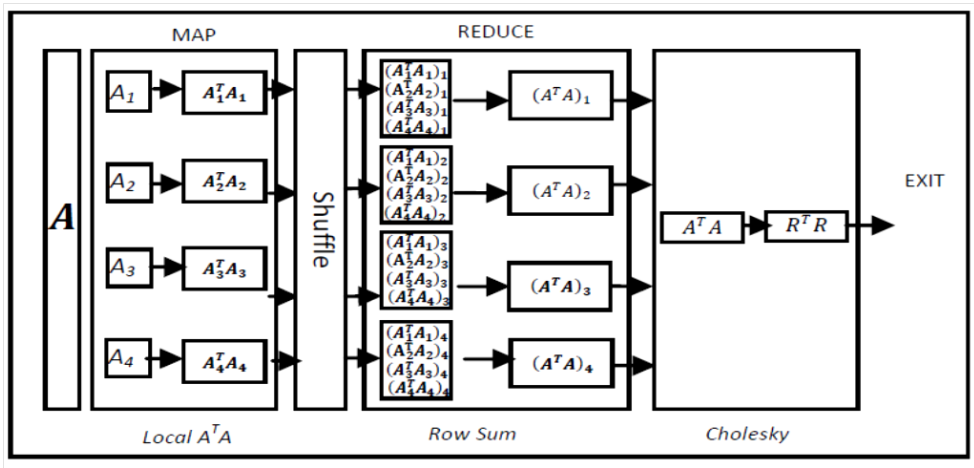

Figure 1. Map Reduce of Cholesky QR

2.1.2. Tall and Skinny-QR

The Tall and Skinny QR (TS-QR)(Constantine & David, June 8, 2011) factorization algorithm produces the factorization A = QR of a matrix A that is designed for the case where A is a tall and skinny matrix. A matrix$A \in R_{m\times n}$ is said to be tall and skinny if m≫n. Such matrices have many applications, one of which is solving a least squares problem of $Ax = b\ where\ x, b \in R_n$. This matrix would be very tall and skinny as the number of observations (m) often exceeds several thousand whereas the number of variables in the model (n) is usually less than very small in comparison and it is often less than 10 in many cases. There are two types of TS-QR factorization methods are available.

2.1.3. Indirect TSQR

We will now briefly review the Indirect TS-QR algorithm and its implementation to facilitate the explanation of the more intricate direct version. Let A be a matrix with m rows and n columns, which is partitioned across four map tasks as shown in equation 1:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} \tag{1}$$

Each map task computes a local QR factorization as shown in equation 2

$$A = \begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{bmatrix} * \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} \tag{2}$$

The matrix of stacked upper triangular matrices on the right is then passed to a reduce task and factored into $\tilde{Q}\tilde{R}$. At this point, we have the QR factorization of A in product form $A = \begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{bmatrix} \tilde{Q}\tilde{R}$

The Indirect TS-QR method ignores the intermediate Q factors and simply outputs the n × n factors Ri in the intermediate stage and $\tilde{Q}\tilde{R}$ in the final stage. Figure 3 illustrates each map and reduces output. Given the matrix R, the simplest method for computing Q is computing the inverse of R and multiplying by A, i.e. computing AR−1. Since R is n×n, upper-triangular matrix and computation of its inverse is straightforward. Figure 3 illustrates how the matrix multiplication and iterative refinement step translate to MapReduce. This "indirect" method of the inverse computation is not backwards stable. Thus, a step of iterative refinement may be used to get Q within desired accuracy.

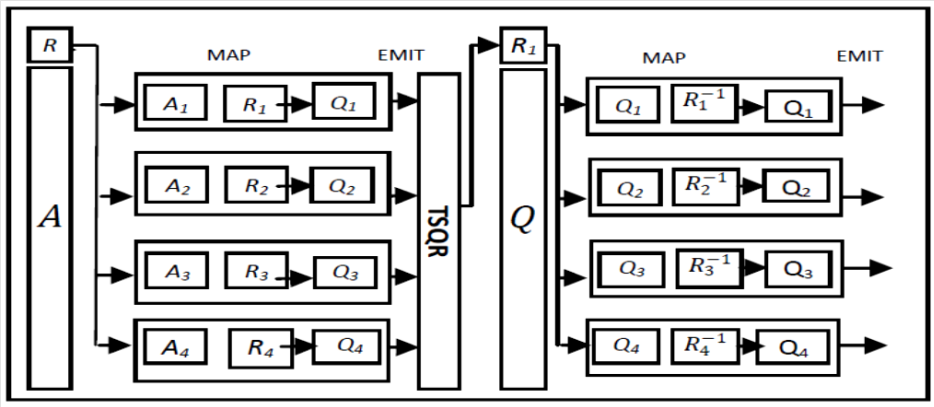

Figure 3. MapReduce of Indirect QR

2.1.4. Direct TS-QR

In direct TS-QR method, the direct computation of QR decomposition of matrix A in three steps using two map functions and one reduce function, as illustrated in Fig. 4. Consider a matrix A with m rows and n columns as defined in equation 1, which is partitioned across four map tasks for the first step as in Figure 4. The first step uses only map tasks. Each task collects data as a local matrix, computes a single QR decomposition, and emits Q and R to separate files. The factorization of A then looks as equation 2, with $Q_j\, R_j$ the computed factorization on the jth task. The second step is a single reduce task. The input is the set of R factors from the first step. The R factors are collected as a matrix and a single QR decomposition is performed. The sections of Q corresponding to each R factor are emitted as values. $\tilde{R}$is the final upper triangular factor in our QR decomposition of A:

$$\begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} = \begin{bmatrix} Q_1^2 \\ Q_2^2 \\ Q_3^2 \\ Q_4^2 \end{bmatrix} \tilde{R}$$

The third step also uses only map tasks. The input is the set of Q factors from the first step. The Q factors from the second step are small enough that we distribute the data in a file to all map tasks. The corresponding Q factors are multiplied together to emit the final Q:

$$Q = \begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{bmatrix} \begin{bmatrix} Q_1^2 \\ Q_2^2 \\ Q_3^2 \\ Q_4^2 \end{bmatrix} = \begin{bmatrix} Q_1 Q_1^2 \\ Q_2 Q_2^2 \\ Q_3 Q_3^2 \\ Q_4 Q_4^2 \end{bmatrix}$$

$$A = Q\tilde{R}$$

In the first step, the key-value pairs emitted use a unique map task identifier as the key and the Q or R factor as the value. The reduce task in the second step maintains an ordered list of the keys read. The kth key in the list corresponds to rows $((k-1)(n+1)$to $(kn)$of the locally computed Q factor. The map tasks in the third step parse a data file containing the Q factors from the second step, and this redundant parsing allows us to skip the shuffle and reduce.

2.1.5. Householder QR

This algorithm is not as friendly to MapReduce environment as either Cholesky QR or Indirect TS-QR methods. The main rationale behind for this phenomenon is the iterative nature of the algorithm. At each step of the algorithm, the matrix A is completely updated. In MapReduce, this has to be achieved by means of constantly rewrite the matrix on disk. Conceptually, each step of the Householder QR method corresponds to three MapReduce calls.

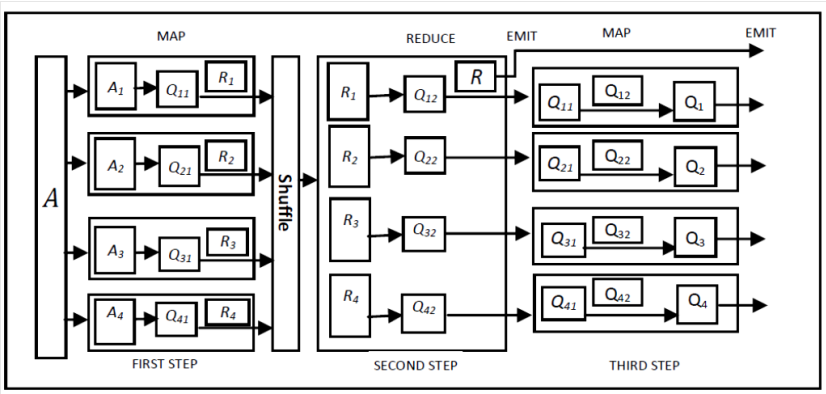

Figure 4. MapReduce Direct QR – First, Second and Third Step

As illustrated in Figure 5, the first step of the algorithm computes the norm of a column of A to help form the Householder reflector. The second and third steps of this algorithm update the matrix with$A \leftarrow (A - 2\,V(A^T\,V)^T)$, where V is the Householder reflector. However, in the actual implementation, the first and third steps are combined because we can compute the norm for the next step immediately after updating the matrix in the third step. Thus, the MapReduce Householder QR algorithm uses 2n passes over the data for the matrix A with n columns. Every other pass requires rewriting the matrix on disk. As n grows, the performance of this algorithm becomes significantly worse than other algorithms. This MapReduce implementation of Householder QR is a BLAS 2 algorithm, whereas standard Sca/LAPACK uses a BLAS 3 algorithm. The central reason for this is the row-wise layout of the matrix in the HDFS. For tall-and-skinny matrices, the canonical key-value pair

stored in HDFS uses a row as the matrix as the value and a unique row identifier for the key. Thus, reading the leading columns of the matrix has the same cost as reading the entire matrix. The stock BLAS 3 algorithm for LAPACK is a much better choice for their column-wise matrix layout.

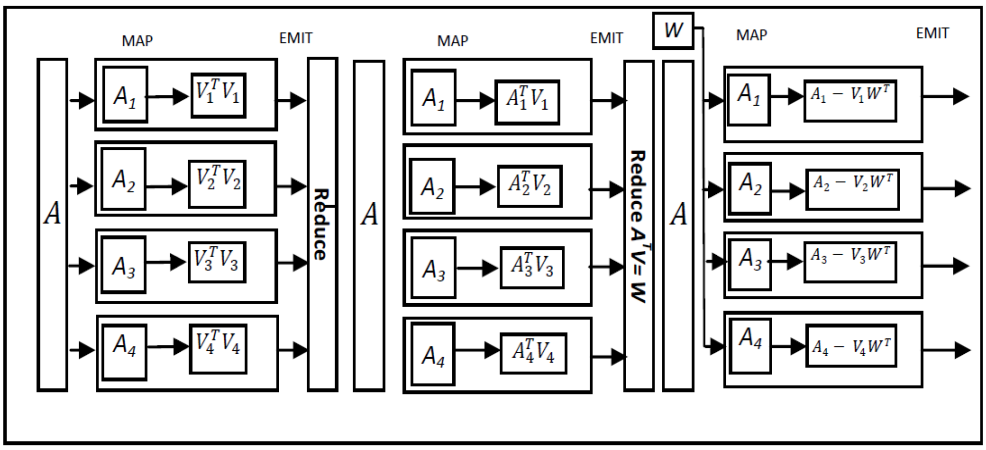

Figure 5. Map Reduce Householder QR

2.2 Singular Value Decomposition (SVD)

Singular value decomposition (SVD) shows strong vitality in the area of information analysis and has significant application value in most of the scientific big data fields. However, with the rapid development of internet, the information online reveals fast growing trend. For a large-scale matrix, applying SVD computation directly is both time consuming and memory demanding. There are many mechanisms are available to speed up the computation of SVD based on the message passing interface model. However, to deal with largescale data processing, a MapReduce model has many advantages over a message passing interface model, such as fault tolerance, load balancing and simplicity. Suppose M is an m × n matrix whose entries come from the field K, which is either the field of real numbers or the field of complex numbers. Then there exists a factorization, called a singular value decomposition of M, of the form $M = U\sum V^{*}M${\displaystyle \mathbf {M} =\mathbf {U} {\boldsymbol {\Sigma }}\mathbf {V} ^{*}}M where U is a m × m unitary matrix, Σ is a diagonal m × n matrix with non-negative real numbers on the diagonal, and V∗ is a n × n, unitary matrix over K. (If K = R, unitary matrices are orthogonal matrices.) V∗ is the conjugate transpose of the n × n unitary matrix, V. Some of the methods in SVD are discussed below

2.2.1. Divide and Conquer SVD:

Divide-and-conquer approach is efficient way for solving full rank SVD problem (Zhao, et al., 28 November 2014). The key point of this algorithm is to split the original problem into many sub-problems using a division strategy and merge the result of the sub-problems to produce the final result. Therefore, this algorithm inherently has satisfactory parallelization and scalability when applied into distributed systems. The main components of this algorithm are blocked matrix multiplication and solving the secular equation. Both of them can be further divided and solved using multiple smaller sub-problems. Also, the recursive fashion of this algorithm demonstrates that it is optimal in terms of number of iterations among different types of full rank SVD algorithms. Generally, the SVD of a general matrix $A \in R^{m*n}$where $(m > n)$, is computed in two phases:

- I: Orthogonally reduce A to a bi-diagonal matrix B:$A = (U_1U_2) * \binom{B}{0} * V^T$
- II: Compute the SVD of B: $B = Q * \sum * W^T$

The SVD of A is then computed as $A = \begin{pmatrix} U_1 & Q & U_2 \end{pmatrix} * \begin{pmatrix} \Sigma \\ 0 \end{pmatrix} * \begin{pmatrix} V & W \end{pmatrix}^T$

Phase II consumes the majority of the computation time. It can be implemented using several algorithms. The basic idea of this algorithm is to merge the SVD results of two similar sub-matrices from a larger matrix. Given

the (N + 1) ∗ N lower bi-diagonal matrix B, divide-and-conquer recursively divides B into two sub-matrices as $B = \begin{pmatrix} B_1 & \alpha_k e_k & 0 \\ 0 & \beta_k e_1 & B_2 \end{pmatrix}$

where B1 and B2 are k ∗ (k − 1) and (N − k + 1) ∗ (N − k) lower bi-diagonal matrices, respectively, and e_j is the j^{th} unit vector of appropriate dimension.

2.2.2. Divide-And-Conquer SVD Algorithm Using MapReduce:

In MapReduce implementation, each merge task of SVD algorithm can be treated as a node in a binary tree and carry out a two-stage task scheduling strategy which dynamically parallelize the computation of merging process. Further, the input matrix spilt into leaf problems according to the division strategy of the algorithm using MapReduce. Third, matrix multiplication of the whole divide-and-conquer SVD algorithm carried out by organizing as a binary tree. In the binary tree, leaf nodes denote minimal sub-problems, and non-leaf nodes denote merge tasks to merge SVD result of two smaller sub-problems. Divide the original problem recursively until to reach one leaf problem and solve the leaf problem using QR iteration-based SVD approach and then merge the SVD results of two sibling nodes to form the result of their parent node according to the recursion branches generated by the program.

In order to keep the global information of the whole binary tree, for a given fixed size of matrix, first split the given original matrix into a set of leaf sub-problems using a specific division algorithm and then solve all the leaf nodes and subsequently execute the merge tasks level by level. In Hadoop runtime environment, a simple way of parallelization is to let one MapReduce job execute one level of merge tasks. When the merging stage of SVD moves toward upper level of the binary tree, the number of map or reduce tasks will doubly reduce level by level. As a result, at the root level, there will be only one task to execute the merging task, while the amount of computation for each task will doubly increase as the size of matrix for each task increase level by level. This trend will lead to a considerable load imbalance problem.

To address these problems, another way of parallelization is to let one or more MapReduce jobs execute one merge task of the whole tree. In this way, Hadoop framework will dynamically distribute the computation work to all nodes in distributed systems.

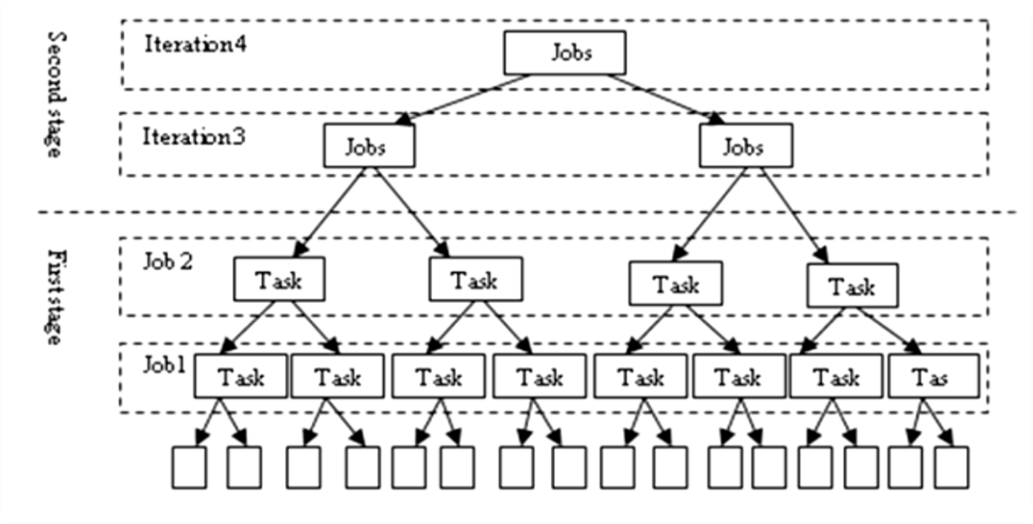

Figure 6. Two-Stage Map Reduce Strategy

However, there are still limitations in this approach. Consider one situation, where we define a very small size of matrix as the minimal sub-problem for a very big size of original problem, which will generate many small size leaf-problems. Therefore, there will be too many Map Reduce jobs to be launched for the bottom level of

the tree, while the size of input data for these jobs is very small. For a matrix whose size is small, using the memory of a single node to deal with the computation is quite enough. If we parallelize small size matrix in a distributed manner, additional cost of I/O and network transfer during the algorithm execution will be a significant bottleneck. Weighing the merits and demerits of each approach, a two-stage task scheduling algorithm is suggested. Namely, at the bottom level of the tree, if the size of matrix is less than a specific threshold s1 apply the first way of parallelization call it as first stage. As the program moves to a specific level when the size of matrix is equal or larger than s1, then use the second way of parallelization named as second stage. Thus, by taking advantage of both approaches to significantly improve the performance of the algorithm running in distributed environment. The architecture is shown in Figure 6.

During the process of the second stage, each merge task of one level in the tree is composed of several steps and each step will be assigned to one MapReduce job. Different steps of one same merge task should be executed one by one serially according to the algorithm logic. Meanwhile, different merge tasks belonging to the same or different levels can be executed independently in pipelined manner. Figure7 shows the procedure of pipelined task scheduling strategy. Assuming at time t1, there are four threads scheduling task 1000, 1001, 1010, and 1011. And then at time t2, both merge tasks 1000 and 1001 are finished, the thread for task 100 will start immediately without the necessity to wait for task 1010 and 1011 to complete. Therefore, by adopting the pipelined task scheduling strategy, at any time, full use of the computation resources in distributed environment can be achieved. In a distributed runtime environment, for a large-scale lower bi-diagonal matrix, even if only store its non-zero elements, it is impossible for a single node to store all its elements in memory. Therefore, traditional recursive division approach is no longer applicable. As a result, the first problem that the divide-and-conquer SVD algorithm should deal with to efficiently extract all leaf problems from the original matrix, which exists in distributed file system.

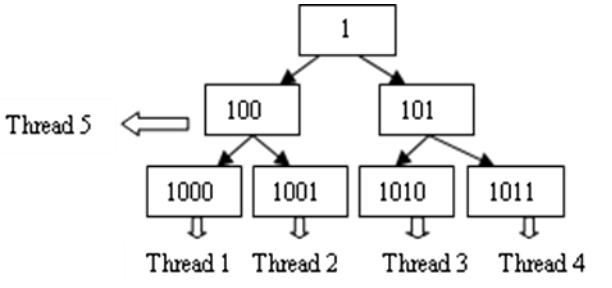

Figure 7. Pipelined Task Scheduling

2.2.3. Jacobi SVD

The Jacobi method is a stationary iterative algorithm for determining a solution of a linear system (Barret, et al., 1994). This method makes two assumptions: first, the system $Ax = b$ has a unique solution. And second, the coefficient matrix A has no zeros on its main diagonal. The Jacobi method is initialized to values of an approximate solution. Then the method is performed iteratively to improve the approximated values to a more accurate solution. This method stops when the desired accuracy is reached. The method is guaranteed to converge if the matrix A is strictly diagonally dominant(Kacamarga, Pardamean, & Baurley, 2014).

In Map Reduce implementation of the Jacobi method, each iteration is one MapReduce job. Each job consists of a map function and a reduce function. In this application, the Map function is used to parse input data and perform a Jacobi calculation to find the solution x. After that, each result is aggregated by the reduce function and exported into output. The job is described in Table 1. At the end of reduce function, the job increments the iteration variable by one and check whether the value is equal the number of iterations k specified by the user. If the iteration variable is less than the variable k, another job is launched. Figure 8 shows the flowchart of Jacobi method implemented in map reduce environment.

Table 1: MapReduce Jacobi method

Job name	Job description	Map function	Reduce function
Calculation x values	Calculate the next approximated solution x	Calculate x values	Aggregate all result values, export into output and increment the number of iterations by one

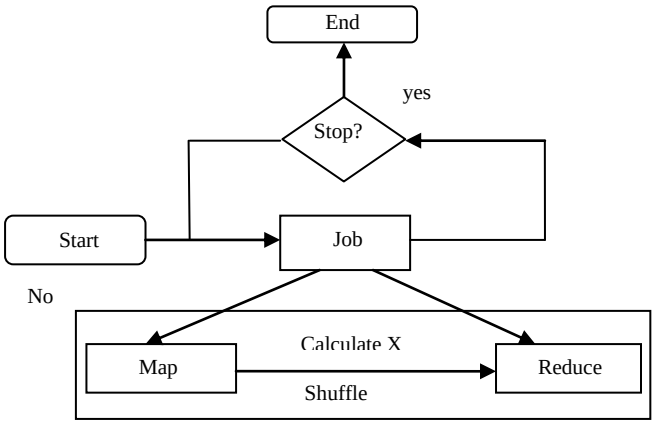

Figure 8. Jacobi Method Implemented in Map reduce

2.2.4. Stochastic SVD (SSVD)

Stochastic SVD is a stochastic technique for computing large dimensional approximate low rank SVDs with rank reaching potentially into hundreds of singular values with just very few passes over data. The main use of SSVD is its accuracy in terms of precision and recall, but reduced computational cost. The SSVD algorithm is more suitable in the distributed computing environment because its computation can be parallelized easily (Yu, Kao, & Lee, 2016). SSVD uses at most 3 MR sequential steps (map-only + map-reduce + 2 optional parallel map-reduce jobs) to produce reduced rank approximation of U, V and S matrices. Additionally, two more map-reduce steps are added for each power iteration step if requested(Lee & Chang, 2013).

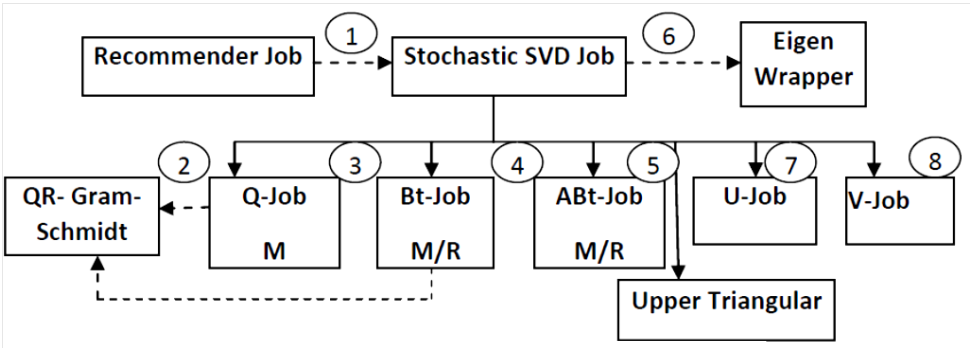

Figure 9. Working of SSVD

The Mahout library (Lyubimov, 2010)has the MapReduce implementation of the stochastic SVD. It contains five MapReduce jobs, as explained below.

1. Q-Job: a map-only job that computes Y=AG and performs the QR decomposition of the block sub matrices of Y.
2. Bt-Job: a job that merges the Q-factor in the QR decomposition, and produces the matrix BT.
3. ABt-Job: simply computes matrix BBT.
4. U-Job: generates U matrix upon requests.
5. V-Job: generates V matrix upon requests.

The working of the SSVD implementation is explained in Figure 9. First, the Q-Job computes the QR decomposition and the Bt-Job generates the matrix BT. The generated Q factor and R-factor are stored in the QR Gram Schmidt object. Next, the AB t-Job computes the symmetric matrix BBT and stores it to an Upper Triangular object. Then the sequential code of eigen-decomposition of the small matrix BBT is performed, whose results are put in the Eigen Wrapper object. If the U-factor or and V-factor are needed, the U-Job and the V-Job will be invoked to compute them from the data stored in Upper Triangular and Eigen Wrapper. The idea of SSVD is to reduce problem dimensions by using random projection while keeping major driving factors of the dataset more or less intact. As a result, the problem may be reduced in size quite a bit. In terms of MapReduce implementation, we'd run problem projection & pre-processing using Map Reduce (bulk number crunching) and then solve small problem in a jiffy inside the front-end process. The trade-off is a rather heavy loss in precision compared to other methods. Bottom line, as such, Stochastic SVD method is unlikely suitable to crunch numbers for a rocket booster design but quite likely is what bulk computational linguistics needs.

3.CONCLUSION

To store and process large volume of data with complex datatypes, Hadoop technology help us to handle in efficient manner and process those data with best computational power with minimum time. Factorization is a mathematical concept to decompose the given matrix into two sub matrices and find the solution in an effective manner. In this paper we have studied elaborately about two predominant factorization techniques QR and SVD methods and its implementation details on Hadoop distributed file system with MapReduce technology.

REFERENCES

A.P.S.Aslin, J., & D.Usha (2014). A Survey of Big Data Processing in Perspective of Hadoop and MapReduce. International Journal of Current Engineering and Technology, 4(2).

Barret, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., et al. (1994). Templates for the Solution of Linear Systems. Building Blocks for Iterative Methods . Philadelphia.

Benson, A. R., Gleich, D. F., & Demmel, J. (2013). Direct QR Factorizations for tall-and-skinny matrices in MapReduce architectures. IEEE International Conference on Big Data.

Constantine, P., & David, F. G. (2011). Tall and Skinny QR Factorizations in Map reduce architectures. MapReduce'11. San Jose, California, USA.

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI2004), 137–150.

Dr.R.Gunavathi, & P.Sudha. (2016). A Survey Paper on Map Reduce in Big Data. International Journal of Science and Research (IJSR).

Gadekar, H., & P S Bhosale, p. D. (2014). A Review Paper on Big Data and Hadoop. International Journal of Scientific and Research Publications, 4(10).

HuangLan, Xiao-wei, W., ZhaiYan-dong, & Bin, Y. (2009). Extraction of User Profile Based on the Hadoop Framework. WiCOM'09 Proceedings of the 5th International Conference on Wireless communications. Beijing, China .

Kacamarga, M. F., Pardamean, B., & Baurley, J. (2014). Comparison of Conjugate Gradient Method and Jacobi Method Algorithm on MapReduce Framework. Applied Mathematical Sciences, 8(17), 837 – 849.

Lee, C. R., & Chang, Y. F. (2013). Enhancing Accuracy and Performance of Collaborative Filtering Algorithm by Stochastic SVD and Its MapReduce Implementation. IEEE 27th International Symposium on Parallel &Distributed Processing Workshops.

Lyubimov, D. (2010). MapReduce QR decomposition MapReduce SSVD Working Notes .

Yu, S. C., Kao, Q.-L., & Lee, C.-R. (2016). Performance Optimization of the SSVD Collaborative Filtering Algorithm on MapReduce Architectures. IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing.

Zhao, S., Li, R., Tian, W., Xiao, W., Dong, X., Liao, D., et al. (2014). Divide-and-conquer approach for solving singular value decomposition based on MapReduce. Concurrency and Computation: Practice And Experience.