CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3436

SPECIAL ISSUE PAPER

Divide-and-conquer approach for solving singular value decomposition based on MapReduce

Shuoyi Zhao[1], Ruixuan Li[1,*,†], Wenlong Tian[2], Weijun Xiao[3], Xinhua Dong[1], Dongjie Liao[1], Samee U. Khan[4] and Keqin Li[5]

[1]*School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China*
[2]*School of Software Engineering, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China*
[3]*Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA 23284, USA*
[4]*Department of Electrical and Computer Engineering, North Dakota State University, Fargo, ND 58108, USA*
[5]*Department of Computer Science, State University of New York–New Paltz, New Paltz, New York, NY 12561, USA*

SUMMARY

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 works available to speed up the computation of SVD based on the message passing interface model. However, to deal with large-scale data processing, a MapReduce model has many advantages over a message passing interface model, such as fault tolerance, load balancing and simplicity. For a MapReduce environment, existing approaches only focus on low rank SVD approximation and tall-and-skinny matrix SVD computation, and there are no implementations of full rank SVD computation. In this paper, we propose a MapReduce-based implementation for solving divide-and-conquer SVD algorithm. To achieve high performance, we design a two-stage task scheduling strategy based on the mathematical characteristics of divide-and-conquer SVD algorithm. To further strengthen the performance, we propose a row-index-based divide algorithm, a pipelined task scheduling method, and revised block matrix multiplication in MapReduce framework. Experimental result shows the efficiency of our algorithm. Our implementation can accommodate full rank SVD computation of large-scale matrix very efficiently. Copyright © 2014 John Wiley & Sons, Ltd.

Received 14 June 2014; Revised 7 October 2014; Accepted 29 October 2014

KEY WORDS: distributed computation; divide-and-conquer; MapReduce; singular value decomposition

1. INTRODUCTION

Singular value decomposition (SVD) [1] is widely used in scientific big data and engineering fields, including signal and image processing, system distinguishing, social network analysis, and recommender systems. SVD plays an important role in complex information processing. However, the amount of information generated from the World Wide Web is big and has been growing very rapidly. The cost of a large-scale SVD computation is stupendous, so that it is infeasible for an individual machine to meet the storage and SVD computation needs of large-scale matrix.

MapReduce [2], first introduced by Google, has been a very attractive way of processing massive datasets. Apache Hadoop project is the most popular implementation of MapReduce programming

*Correspondence to: Ruixuan Li, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China.
†E-mail: rxli@hust.edu.cn

Copyright © 2014 John Wiley & Sons, Ltd.

model. Hadoop relies on a distributed file system to share large-scale dataset among clusters. MapReduce procedure includes a shuffle phase where intermediate result is written on disk. Thus, we do not expect algorithms running in MapReduce framework to be faster than algorithms deployed in a state-of-the-art in-memory message passing interface(MPI) cluster. However, MapReduce model has many advantages over MPI model, which makes it attractive to process large-scale dataset. MapReduce model is designed to be fault tolerant while MPI model is not. Moreover, MapReduce model offers mechanisms to automatically deal with load balancing issues as well as input and output operations in a distributed environment, which greatly simplifies the programming. In today's information technology industry, many companies have employed Hadoop-distributed systems to deal with their daily business data.

There are already implementations of SVD computation in MapReduce available. However, these implementations only focus on two aspects. The first one is low rank SVD approximation of matrices [3–5]. These research works are designed for those applications such as large-scale recommender systems in which only small fraction of singular values are needed. Nevertheless, for applications, such as matrix pseudo-inverse calculations that are widely used in large-scale medical and astronomy image processing field nowadays [6] and full rank principal component analysis in face recognition [7], performing a full rank SVD computation is required. Obviously, current implementations of SVD computation in MapReduce framework can hardly deal with them. The second aspect is tall-and-skinny matrix SVD computation. However, these methods only make sense for special matrices [8, 9].

Among all the approaches that are intended for full rank SVD computation, the most commonly used SVD algorithm is based on quick response QR iteration [10]. However, this algorithm has one major drawback when deployed in distributed computing systems. QR iteration-based SVD algorithm requires huge amount of iterations to produce the final result. While, on the other hand, for one iteration of a MapReduce job, Hadoop has to read the input data from a distributed file system and write the result back. This repeated read and write input/output (I/O) operations become the bottleneck of the algorithm. From this respective, QR iteration-based SVD algorithm is not suitable in MapReduce framework. Another approach for full rank SVD computation is Jacobi method. Nonetheless, this method also requires large number of iterations to produce the final result [11].

In comparison with the QR iteration-based SVD algorithm, divide-and-conquer approach is another efficient way for solving full rank SVD problem. 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 has satisfactory parallelization and scalability when applied into distributed systems. 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. We believe that this algorithm is well suited to MapReduce framework.

In this paper, we propose a parallelized divide-and-conquer SVD algorithm using MapReduce programming model. We have three main contributions in our work. First, each merge task of SVD algorithm is organized as a node in a binary tree. We propose a two-stage task scheduling strategy to dynamically parallelize the computation of merging tasks in a level of tree according to their matrix size. Second, we design an efficient way to fully split the input matrix into leaf problems according to the division strategy of the algorithm using MapReduce. Third, matrix multiplication is the most expensive step of the whole divide-and-conquer SVD algorithm. We improve the basic block matrix multiplication for divide-and-conquer SVD algorithm by avoiding zero elements transfer and computation in the cluster.

The rest of this paper is organized as follows. Section 2 provides an overview of related work. The detailed implementation of SVD is given in Section 3. Our overall architecture of SVD algorithm in MapReduce framework is presented in Section 4, and Section 5 describes the whole process of SVD parallelization in MapReduce framework. Section 6 presents the experiment and analysis. Finally, we draw a conclusion in Section 7.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

2. RELATED WORK

Full rank SVD algorithms can be generally divided into three categories. The first one is QR iteration approach, which is most widely used. This approach has high accuracy and numerical stability [10]. To obtain higher performance, Cuppen first proposed a divide-and-conquer approach to solve eigenproblems [12], and it was improved by many other researchers. Gu and Eisenstat [13] proposed a divide-and-conquer SVD algorithm, which is regarded as the fastest algorithm among all SVD methods. This algorithm can be used to solve least square problems much faster than QR iteration algorithm [14]. The third one is Jacobi method. In terms of numeric calculation speed, Jacobi method is the slowest, but it has higher precision than any other methods [11].

Lots of research work has been devoted to improve SVD calculation efficiency. Graphics processing unit, which has strong computational efficiency, was used to improve computation speed of SVD [15–17]. However, their work focus on how to speed up SVD computation in stand-alone context and cannot deal with SVD of large-scale matrices.

Mahout [3], an open source toolkit, includes SVD algorithm based on MapReduce that employs Lanczos method to do SVD computation. In each iteration, one singular value and corresponding singular vectors are calculated. However, for full rank SVD applications, a large number of iterations are required. As mentioned earlier, this drawback produces severely adverse effects upon the performance of algorithm and makes the computation intractable when the dimension of matrix is large. In addition, Mahout SVD is only designed for sparse matrix.

Liu *et al*. [18] proposed a novel MapReduce-based distributed latent semantic indexing. However, this work did not include the approach to obtain the original documents-term matrix in distributed environment. Yeh *et al*. [19] proposed an iterative divide-and-conquer-based estimator for solving large-scale Least Square Estimation (LSE) problems. Iteratively, LSE problem to be solved is transformed to equivalent but smaller LSE problems. Actually, this method did not aim to solve SVD problem.

In other related works, Constantine *et al*. [8, 20] presented a method to compute QR factorization in MapReduce. They design an efficient way to get the result of R matrix whose dimension is small and then directly compute the SVD of small matrix. Their work is improved by Benson *et al*. [9] to obtain a stable tall-and-skinny QR factorization method based on MapReduce architecture. However, their methods are only designed for tall and skinny matrices. To solve the SVD of tall and fat matrices, Bayramli [4] proposed a solution based on MapReduce framework. It adopts a random projection method to transform the original large-scale matrix to a small matrix and then compute the SVD of small matrix. Obviously, their work mainly deals with rank k approximation of SVD. When k is large, their implementation will not work. Ding *et al*. [21] introduced a solution, which used PARPACK toolkit in MapReduce framework to obtain the singular values and singular vectors of the large-scale matrix. Nevertheless, it still aims to deal with low rank SVD computation and numerous of iterations are required to obtain a full rank SVD.

To the best of our knowledge, there is no effective implementation about full rank SVD computation in MapReduce framework.

3. DIVIDE-AND-CONQUER SVD ALGORITHM

In this section, we give an overview of the process of the divide-and-conquer SVD algorithm. More detailed information about this algorithm can be found in [13].

Generally, given a matrix $A \in R^{m\times n}(m > n)$, we can always find an equation

$$A = U \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix} V^T \tag{1}$$

where $U = [u_1, u_2, \ldots, u_m] \in R^{m\times m}$ and $V = [v_1, v_2, \ldots, v_n] \in R^{n\times n}$ are both orthogonal matrices, $\begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix}$ is a $m \times n$ matrix and Σ_r with nonzero diagonal entries satisfying $w_1 \geqslant w_2 \geqslant \ldots w_r > 0$. Equation (1) is what we call SVD. Normally, the input matrix of an SVD algorithm

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

is bi-diagonal matrix. If not, we should first convert it into bi-diagonal matrix by a Householder transformation. Here and in this paper, we assume to have a $(N+1)\times N$ lower bi-diagonal matrix B as the input of our SVD algorithm. Divide-and-conquer algorithm first recursively divides matrix B into the following form:

$$B = \begin{pmatrix} B_1 & \alpha_k e_k & 0 \\ 0 & \beta_k e_1 & B_2 \end{pmatrix} \tag{2}$$

where B_1 and B_2 are also $K\times(K-1)$ and $(N-K+1)\times(N-K)$ bi-diagonal matrices, respectively, and $K = \lfloor K/2 \rfloor$. The algorithm will solve the SVD of B_1 and B_2 recursively. Assume that the SVD of B_i is $\begin{pmatrix} Q_i & q_i \end{pmatrix}\begin{pmatrix} D_i \\ 0 \end{pmatrix} W_i^T$. We obtain, then, we permute the right hand of Equation (2) to the following form:

$$\begin{aligned} B &= QMW^T \\ &= \begin{pmatrix} c_0 q_1 & Q_1 & 0 & s_0 q_1 \\ s_0 q_2 & 0 & Q_2 & c_0 q_2 \end{pmatrix} \begin{pmatrix} r_0 & 0 & 0 \\ \alpha_k l_1 & D_1 & 0 \\ \beta_k f_2 & 0 & D_2 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & W_1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & W_2 \end{pmatrix} \end{aligned} \tag{3}$$

where $r_0 = \sqrt{(\alpha_k\lambda_1)^2 + (\beta_k\phi_2)^2}$ and variables l_1, f_2^T are last row and first row of Q_1 and Q_2, respectively, λ_1 and ϕ_2 are last component and first component of q_1 and q_2, respectively. The next step is to solve the singular values of middle matrix

$$\begin{pmatrix} z_1 & & & \\ z_2 & d_2 & & \\ \vdots & & \ddots & \\ z_n & & & d_n \end{pmatrix}$$

where $0 \equiv d_1 \leqslant d_2 \leqslant \cdots \leqslant d_n$. The singular values of this matrix are the roots of the secular equation $f(w) = 1 + \sum_{i=1}^{n} \frac{z_k^2}{d_k^2 - w^2} = 0$ and also satisfy the property $0 = d_1 < w_1 < d_2 < \ldots < d_n < w_n < d_n + \|z\|_2$.

After solving all the singular values, we calculate the singular vectors of matrix M using the following equation:

$$u_i = \frac{\left(\frac{z_1}{d_1^2 - w_i^2}, \ldots, \frac{z_n}{d_n^2 - w_i^2}\right)^T}{\sqrt{\sum_{k=1}^{n} \frac{z_k^2}{(d_k^2 - w_i^2)^2}}}, v_i = \frac{\left(-1, \frac{d_2 z_2}{d_2^2 - w_i^2}, \ldots, \frac{d_n z_n}{d_n^2 - w_i^2}\right)^T}{\sqrt{1 + \sum_{k=2}^{n} \frac{(d_k z_k)^2}{(d_k^2 - w_i^2)^2}}} \tag{4}$$

However, in all cases, for a specific singular value w_i, we can only calculate an approximation $\hat{w}_i$ using whatever approaches to solve the secular equation. In reality, even if the variable $\hat{w}_i$ is very close to w_i, algebraic expression $\frac{z_i}{d_i^2 - \hat{w_i}^2}$ and $\frac{d_i z_i}{d_i^2 - \hat{w_i}^2}$ are also very different from the true numeric values $\frac{z_i}{d_i^2 - w_i^2}$. As a result, the accuracy and orthogonality of all singular vectors are not ensured.

To solve this problem, Gu and Eisenstat [13] presented an efficient way that after solving all the singular values $\hat{w}_i$, all the elements $\hat{z}_i$ in the first column of matrix M are recalculated using the following equation:

$$|\hat{z}_i| = \sqrt{\left(\hat{w_n}^2 - d_i^2\right) \prod_{k=1}^{i-1} \frac{\hat{w}_k^2 - d_i^2}{d_k^2 - d_i^2} \prod_{k=i}^{n-1} \frac{\hat{w}_k^2 - d_i^2}{d_{k+1}^2 - d_i^2}} \tag{5}$$

This leads to a high accuracy of left and right singular values of matrix M. After the SVD of matrix M is computed as the product $U\Sigma V^T$, we obtain the SVD of matrix B as $(QUq)\Sigma(WV)^T =$

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

$X\Sigma Y^T$ through two matrix multiplications.

We summarize the divide-and-conquer SVD algorithm as Algorithm 1.

Algorithm 1 Divide-and-conquer SVD algorithm

1: **Input:** bi-diagonal matrix $\boldsymbol{B}$
2: **Output:** the SVD result of bi-diagonal matrix $\boldsymbol{B}$.
3: Define the minimal size of the sub-problem, divide the bi-diagonal matrix $\boldsymbol{B}$ recursively to obtain a binary tree.
4: For all the leaf sub-problems in the binary tree, use QR iteration-based SVD algorithm to solve them.
5: **for** each non-leaf problems in the tree **do**
6: Permute the SVD results of its two sub-problems to construct the middle matrix $\boldsymbol{M}$.
7: Solve the secular equation to get all singular values and then revise the z vector.
8: Calculate the singular vectors of matrix $\boldsymbol{M}$.
9: Calculate the singular vectors of original matrix $\boldsymbol{B}$ by matrix multiplications.
10: **end for**

4. ARCHITECTURE

4.1. Two-stage merge task scheduling

After we split the original bi-diagonal matrix recursively using the specific division strategy, the SVD solving process of the original problem can be organized as a binary tree, as shown in Figure 1. 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. The most direct way of solving this problem is based on a post-order traversal fashion. That is to say, we divide the original problem recursively until we reach one leaf problem, 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. However, in a distributed running environment, we need to record the global information of the whole binary tree, which is significantly memory demanding. To overcome this problem, given a fixed size of matrix, we first fully split the original matrix to a set of leaf sub-problems using a specific division algorithm. Second, we solve all the leaf nodes and then, execute the merge tasks level by level.

In Hadoop runtime environment, a straightforward way of parallelization is to let one MapReduce job execute one level of merge tasks, and each merge task of one level is assigned to one map or reduce task in this job as shown in Figure 1. However, there is one main drawback about this implementation. 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

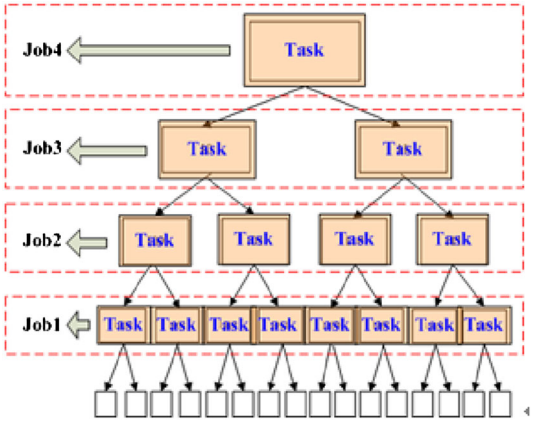

Figure 1. Level by level iteration.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

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 this problem, another way of parallelization is to let one or more MapReduce jobs execute one merge task of the whole tree as illustrated in Figure 2. In this way, Hadoop framework will dynamically distribute the computation work to all nodes in distributed systems. 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 MapReduce jobs to be launched for the

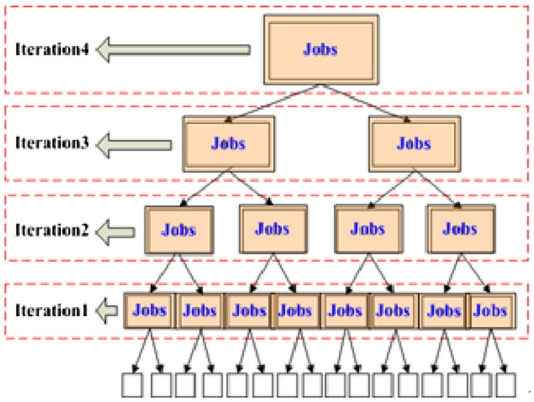

Figure 2. Multi-job task scheduling.

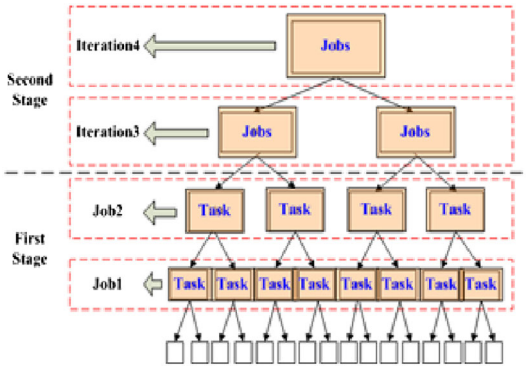

Figure 3. Two-stage task scheduling strategy.

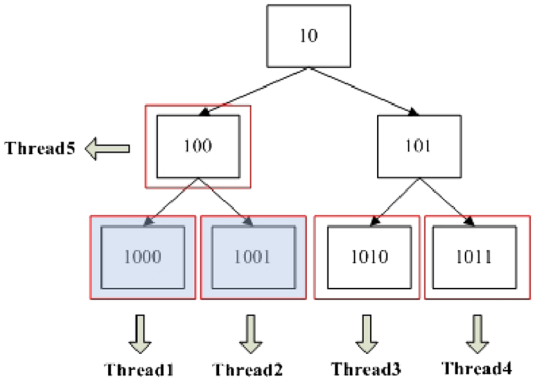

Figure 4. Pipelined task scheduling.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

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, in this paper, we propose a two-stage task scheduling algorithm. Namely, at the bottom level of the tree, if the size of matrix is less than a specific threshold s_1, we use the first way of parallelization. We call it the first stage. As the program moves to a specific level when the size of matrix is equal or larger than s_1, we begin to use the second way of parallelization. We call it the second stage. Thus, we take advantage of both approaches to significantly improve the performance of the algorithm running in distributed environment. The architecture is shown in Figure 3.

4.2. Pipelined task scheduling

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. We use one thread to control the scheduling of jobs for each step in each merge task. To speed up the execution of the algorithm, we propose a pipelined task scheduling strategy.

Figure 4 shows the procedure of how pipelined task scheduling works. Assuming at time t_1, there are four threads scheduling task 1000,1001,1010, and 1011. And then at time t_2, 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. On the contrary, if we adopt level by level scheduling strategy in the same way as the first stage when all the tasks in a level should be finished before its upper level of merge tasks start scheduling, there will be two extreme situations occurring in the second stage. There must be a time when there is only one MapReduce job belonging to the last merge task in a level running in the cluster, which may only occupy several computation nodes in the cluster while other nodes are idle. On the other hand, there also must be a time when all the merge tasks belonging to one level start being scheduled at the same time so that they have to madly compete for the relatively limited computation resource in the cluster. Therefore, by adopting the pipelined task scheduling strategy, at any time, we can make full use of the computation resources in distributed environment.

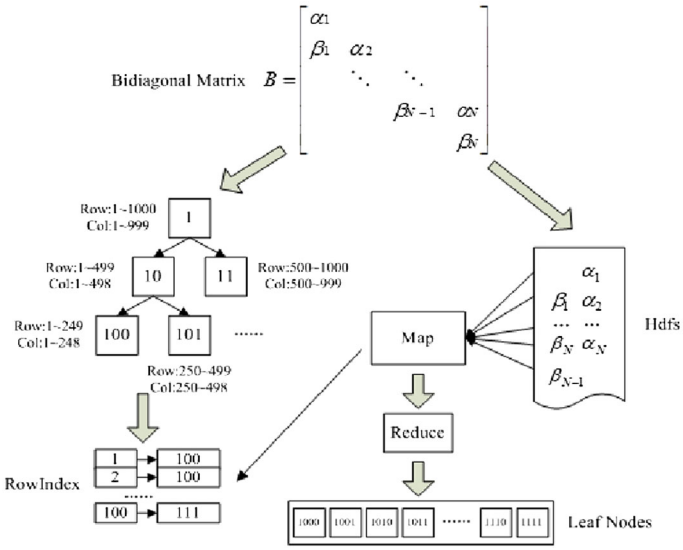

Figure 5. Divide algorithm in MapReduce.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

4.3. Divide algorithm

In a distributed runtime environment, for a large-scale lower bi-diagonal matrix, even if we 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 is how to efficiently extract all leaf problems from the original matrix, which exists in distributed file system. In this paper, we design a row-index-based approach for quick matrix division. Figure 5 shows the whole procedure of our approach. Given the dimensionality of a matrix and the specific division method, we can use an in-memory program to generate a binary tree with each node recording the positions of all sub-problems in the original matrix. Consequently, we can get the whole mappings from each row of the matrix to a specific sub-problem. We call this kind of information row-index. On the other hand, the original bi-diagonal matrix is stored in a distributed file system row by row, and each row of the matrix is stored as a key-value pair. Thus, we launch a MapReduce job, for each row read from Hadoop distributed file system (HDFS) in map task, look up its corresponding row-index information in memory to find which sub-problem it belongs to. For an example, when map task read the second row of matrix in Figure 5, it will emit the key-value pair $< 10, (100, 2, \beta_1, \alpha_2) >$ to reduce task. This means that leaf nodes that share the same parent node will be received as one key-value list pair by reduce task, and we can resemble all the data belonging to one leaf node very easily in reduce task. Then, we can solve all the leaf nodes using QR iteration SVD algorithm and put the SVD result of sibling nodes in one key-value pair. In this way, we get preparation for the subsequent iterations.

5. PARALLELIZATION IMPLEMENTATION IN MAPREDUCE

5.1. Level by level merge implementation

If the size of leaf problems is small, we will merge the sub-problems level by level. All the merge tasks in a level are controlled by one MapReduce job. The output of division algorithm will be the input of the first stage. As shown in Figure 6, all the merge tasks in a level are executed in map tasks. For example, when the sub-problems 1000 and 1001 are merged in a map task and produce the SVD result of 100, the map task will emit a key-value pair $< 10, [100, SVD(100)] >$ to reduce task. Thus, in one reduce task, sibling sub-problems 100 and 101, which share the same parent node 10, will be received. A reduce task merges the SVD results of the sibling nodes as one key-value pair and outputs it to HDFS. This output will be the input of next MapReduce job, which performs the same operation until the size of sub-problems in current level reaches a specific threshold.

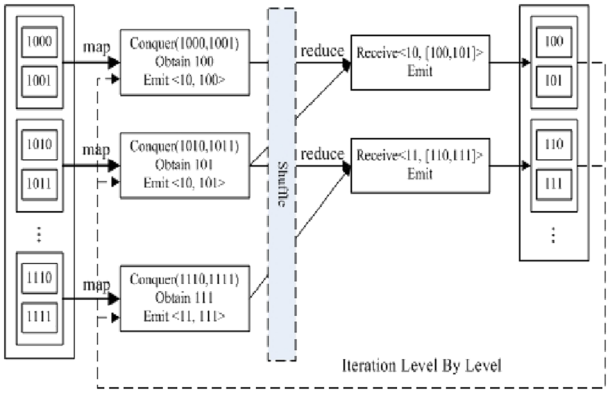

Figure 6. Level by level merge implementation.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

5.2. Multi-job-based pipelined task scheduling

When the matrix grows to a big size, it is incapable for a single node to accommodate a whole matrix in memory. Therefore, it is time to seek ways to parallelize the merge procedure using MapReduce to make it possible for large-scale matrix processing in a distributed environment.

When the algorithm enters the second stage, the layout of matrix in HDFS is a critical point. A reasonable layout of matrix is conducive to the subsequent steps of a merge task. We can see from Algorithm 1 that the whole merge procedure of SVD algorithm is mainly composed of four steps. Because of the mathematical complexity of SVD algorithm, we use separate files to store different parts of matrix involved in the merge procedure of SVD algorithm. For example, we use four different files to store four different parts of the matrix Q, namely $Q1, Q2, (c_0q_1, s_0q_2)^T$, and $(-s_0q_1, c_0q_2)^T$. Also, we use five files to store five different parts of matrix M, namely $r_0, \alpha_k l_1, \beta_k f_2, D_1$, and D_2, and two files to store matrix W. Different parts of different matrices will be the input of different steps of merge procedure that we will describe later on. In addition, we apply a row major or column major storage for all matrices in this paper depending on different situations.

5.2.1. Middle matrix M permutation. Observing three matrices generated in middle matrix permutation step, we discover that six matrices Q_1, Q_2, D_1, D_2, W_1, and W_2 can be obtained from the result of two sub-problems through simple transformation, and this operation fits well in MapReduce, while we need additional consideration for the computation of $(c_0q_1, s_0q_2)^T, (-s_0q_1, c_0q_2)^T, r_0, \alpha_k l_1$, and $\beta_k f_2$. First, we focus on how to obtain $(c_0q_1, s_0q_2)^T, (-s_0q_1, c_0q_2)^T$, and r_0. According to divide-and-conquer algorithm, computation of these three parts only relies on variables q_1, q_2, α_k, and β_k. Hence, we only need to read q_1, q_2, α_k, and β_k from HDFS to calculate them. This step has small amount of computations. Thus, putting this step in the driver module of MapReduce program is quite suitable. Six files that are used to store matrices Q_1, Q_2, W_1, W_2, D_1, and D_2 will be the input of one MapReduce job. This MapReduce job will convert the format of the matrix and output the six matrices Q_1, Q_2, W_1, W_2, D_1, and D_2 to separate files organized line by line. At the same time, when map tasks read the last row of matrix Q_1 or the first row of matrix Q_2, they will output $\alpha_k l_1$ and $\beta_k f_2$ directly to HDFS. This process is shown in Figure 7.

5.2.2. Secular equation solving and z vector revising. This step is to solve the singular values of middle matrix M. The singular values of matrix M satisfy secular equation. There are various approaches available to solve this equation including Newton method, Middle Way method, and bisection method. No matter which method is adopted, if a root is not found in one iteration, another iteration is required to find the next approximate point. Among all these approaches, Newton method and Middle Way method take less iterations than bisection method to get the final result [5, 22, 23]. However, for these methods, doing one numerical iteration is much more

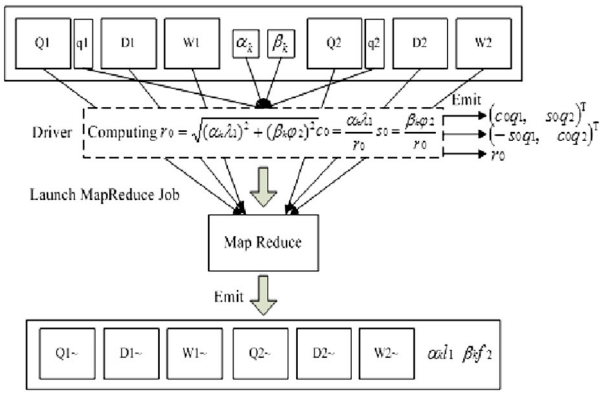

Figure 7. Middle matrix permutation.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

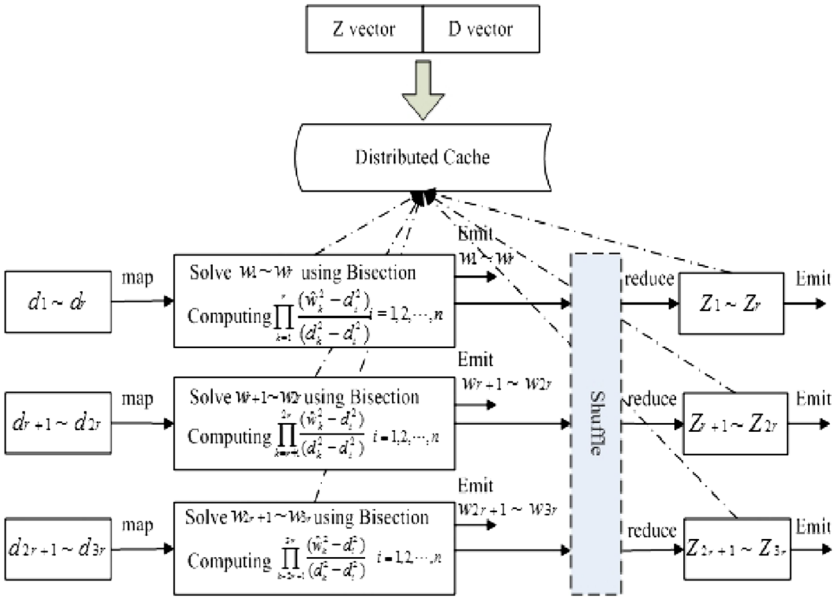

Figure 8. Secular equation solving and z vector revising.

expensive than bisection method. What's more, Newton method and Middle Way method actually do not ensure numerical convergence [17]. Compared with Newton method, bisection method has the slowest arithmetic speed [24]. However, this method is quite simple for every numerical iteration. That is to say, only one floating point operation is needed to find the middle point of upper and lower bounds. Besides, bisection method guarantees numerical convergence perfectly and is easier for parallelization. Therefore, we use bisection method to solve secular equation in distributed environment.

Note that we require all elements of z vector and d vector before we solve one root of secular equation. In this paper, we assume every single node of distributed environment is capable to store two vectors of matrix M in memory. Thus, we can directly import z vector and d vector into memory to make the computation of different singular values work in parallel in distributed environment. We see that the solving process of different singular values can be done independently. Thus, interval segments of singular values are the input of a MapReduce job. The z vector and d vector are added to distributed cache before a MapReduce job is launched. We solve the singular values in map tasks and emit segments of singular values directly into HDFS.

At the same time, we observe Equation (5) and find that the computation of $\hat{z}_i$ relies on d vector and all singular values $\{w_i\}_{i=1}^{n}$. Besides, the computation of $\hat{z}_i$ is the multiplication of multiple factors with the same form and extract a square root at last. Therefore, in map task, after solving a segmentation of singular values $w_1 \sim w_r$, we can calculate a continued product $\prod_{k=1}^{r} \frac{\hat{w}_k{}^2 - d_i^2}{d_k^2 - d_i^2}$ as partial result of $\hat{z}_i$. And then, we send all partial results belonging to $\hat{z}_i$ to one reduce task during shuffle phase. Reduce tasks will multiply all these partial results and perform a square root extraction. At last, reduce tasks emit the final result of $\hat{z}_i$ into HDFS. The process is shown in Figure 8.

5.2.3. UV vector computation. The next step is to compute all the singular vectors of matrix M using Equation (4). As shown in Figure 9, the computations of vector u_i and v_i rely on d vector, revised $\hat{z}_i$ vector and their corresponding singular value w_i. d vector and revised $\hat{z}_i$ vector are shared by the program globally, and the computations of u_i and v_i can be performed independently. As a result, we import the d vector and revised $\hat{z}_i$ vector into Hadoop distributed cache. All segments of singular values output by previous step will be the input of this step. In map task, as one singular value w_i is read from input data, we calculate its corresponding u_i and v_i vectors using

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

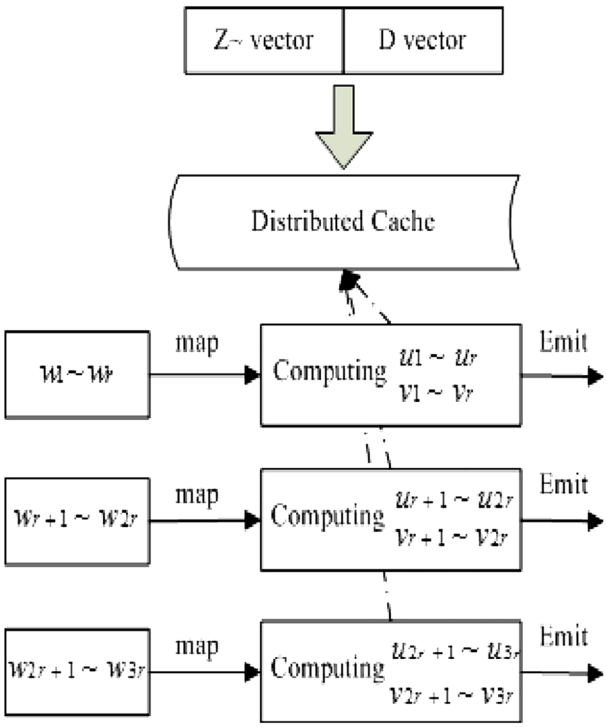

Figure 9. UV vector computation.

Equation (4). The computation result of map task is output into HDFS directly with a format of column major storage.

5.2.4. Matrix multiplication. The final step of merge procedure is matrix multiplication to obtain the singular matrices of matrix B. There are many researches available concerning how to perform large-scale matrix multiplication in MapReduce framework. The most frequently used method is to perform dot production between one row of first matrix and one column of second matrix [25–27]. This method is simple and intuitive when employed in MapReduce framework, while, on the other hand, this method can cause severe network traffic during the shuffle phase [26]. To address this problem, block matrix multiplication is presented [28]. In distributed running environment, to achieve high parallelization, we have to distribute different parts of calculations of result matrix to different nodes, so all the elements of operational matrices need replicated multiple times, which causes additional cost of I/O and network transfer. For general dense matrices, block matrix multiplication requires the least times of matrix element replication of all the available approaches. In this paper, we will adopt a block matrix multiplication approach.

The principal of block matrix multiplication is illustrated as Figure 10. Assume that there is a matrix multiplication as the Equation (6)

$$A \times B = C \tag{6}$$

where matrix A is divided into *NIB* row blocks and *NJB* column blocks, matrix B is divided into *NJB* row blocks and *NKB* column blocks. Figure 10 shows the computation process of first row and first column block of result matrix C, where each block in first row block of matrix A has to be multiplied by another block in first column block of matrix B. However, we discover that there are large amount of continuous position zero elements in matrix Q and matrix W as shown in Equation (3), which occupy approximately half of the total elements in these matrices. Based on the basic block matrix multiplication, there will be a lot of ineffective block replications and block computations.

To avoid this problem, we use a matrix block mapping mechanism. We take the case of matrix Q multiplying matrix U as an example. Recall that we use three separate files to store three parts

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

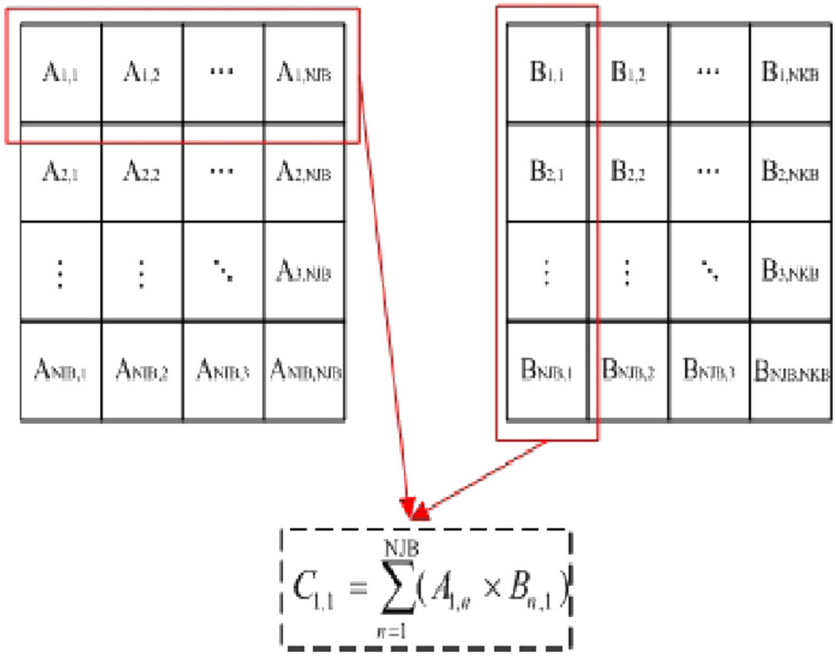

Figure 10. Basic block matrix multiplication.

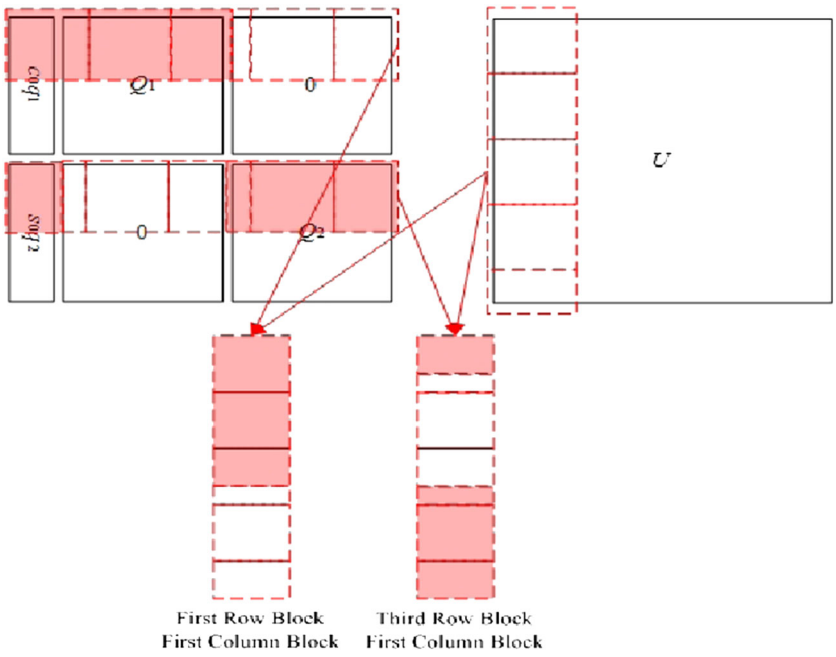

Figure 11. Revised block matrix multiplication.

of matrix Q, namely $(c_0q_1, s_0q_2)^T$, Q_1 and Q_2. Thus, we use these three files instead of the whole matrix Q as input of matrix multiplication algorithm in this paper. Given a specific matrix partitioning strategy, as we read input data from matrix Q row by row, we can obtain the row and column offset of the first element of each row in the original matrix and then map all the elements in a row to a specific block belonging to the matrix. After replication, different parts of operating matrices are distributed to different nodes in the network.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Now, we consider the case of matrix U. According to block matrix multiplication algorithm, we know that each column block of matrix U requires *NIB* times of replication and the ith copy of the block has to be multiplied by the ith row block of matrix Q. However, we find that there exist zero blocks in each row block of matrix Q, which causes such situations where a lot of blocks in matrix U are multiplied by zero blocks in matrix Q. To avoid ineffective block replication and transfer of matrix U as well, we only select partial blocks in each column block of matrix U to be replicated as we perform the ith copy of the whole column block. For example, assume that matrix Q is divided into *NIB* row blocks and *NJB* column blocks where $NIB > 2$ and $NJB > 2$, respectively. Obviously, $U_{1,NJB}$ is a zero block, as we perform the first copy of each column block in matrix U, the last row block will not get replicated. This process is shown in Figure 11.

Therefore, we can improve the performance of matrix multiplication in distributed environment by approximately 50% in theory. The pseudo code of revised block matrix multiplication in MapReduce framework is shown as Algorithm 2. Let $I \times J$ matrix and $J \times K$ matrix be one block of matrices Q and U, respectively, and we assume that matrices Q and U are divided into $NIB \times NJB$ and $NJB \times NKB$ blocks for each row and column, respectively.

Algorithm 2 QU matrix multiplication in MapReduce

1: **Map Input:** Matrix file: $(c_0q_1, s_0q_2)^T, Q_1, Q_2, U$
2: $I_1 \leftarrow$ *number of Row Blocks of Q1*
3: $I_2 \leftarrow$ *number of Row Blocks of Q2*
4: $J_1 \leftarrow$ *number of Column Blocks of* (c_0q_1, Q_1)
5: $J_2 \leftarrow$ *number of Column Blocks of* Q_2
6: **if** from file U **then**
7: **for** $i \leftarrow 0$ *to* I_1 **do**
8: **Output** the first J_1 Row Blocks of Value Row.
9: **end for**
10: **for** $i \leftarrow 0$ *to* I_2 **do**
11: **Output** the first element and the last J_2 Row Blocks of Value Row.
12: **end for**
13: **else**
14: **for** each element v in Value **do**
15: Calculate its Block Location in Matrix Q;
16: **Output** the element with the block location.
17: **end for**
18: **end if**
19: **Reduce Input:** Row Blocks of Matrix Q and Column Blocks of Matrix U
20: Multiply each block of one Row Blocks in Matrix Q by corresponding block of one Column Blocks in Matrix U, Obtain one result Block Matrix.
21: **Output** result Block Matrix.

6. EXPERIMENT

In this section, we demonstrate the effectiveness of our proposed divide-and-conquer SVD algorithm based on MapReduce. We conduct all our experiments in a cluster that contains 32 physical nodes. Each node has two 8-core processors (2.60GHz) and 64GB of random access memory. We use JAVA 1.7 as the programming language and deploy distributed computation framework Hadoop1.0.3 on this cluster with each node configured to run eight map tasks and eight reduce tasks simultaneously. We set the block size of HDFS to 192MB. Notice in this section again, when we mention the size of matrix is N, we mean to have a $N \times (N-1)$ matrix in our experiment.

6.1. Overall performance

Figure 12 shows the overall performance of MapReduce-based divide-and-conquer SVD algorithm. The settings of mappers and reducers for different steps in this experiment are shown in Table I, in which 'Div', 'Iter', 'Per', 'Sec', 'UV', and 'Matr' denote the step of divide, first stage iteration, matrix permutation, secular equation solving, z vector revising, uv vector computation, and matrix multiplication, respectively. The threshold s_1 when the first stage transmits to the second stage is

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

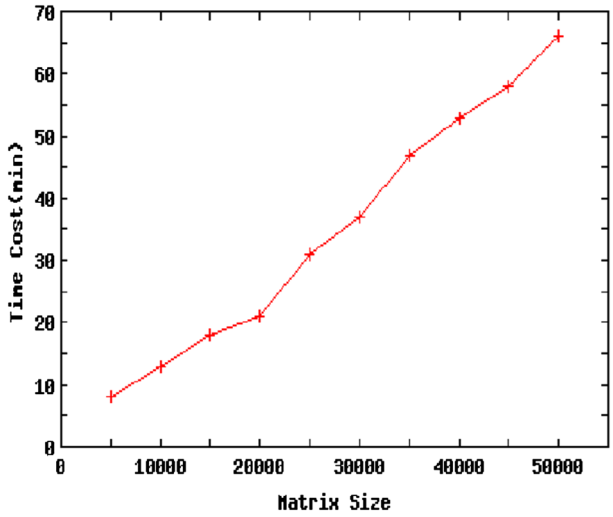

Figure 12. Overall performance.

Table I. Settings of number of mappers and reducers for each step.

Step	Mapper	Reducer
Div	20	10
Iter	20	10
Per	—	10
Sec	20	10
UV	20	—
Matr	—	70

set to 2500, and leaf matrix size is set to 300 for all matrix size. We can see from Figure 12 that as the size of matrix increases, the computation time of our algorithm increases gracefully. However, compared with $O(N^3)$ time complexity of SVD algorithm, our implementation in MapReduce framework only shows approximately linear growing trend in computation time, which proves that when dealing with large-scale SVD matrix processing, our algorithm can successfully distribute the workload to more machines and take advantage of parallelism. Another important reason, which contributes to this growing trend, is that given a fixed leaf matrix size as the threshold for our divide algorithm, the larger matrix size produces higher binary tree. Hence, the advantage of pipelined multi-job-based task scheduling for the second stage is more enhanced.

Figure 13 shows the execution time of different steps in SVD algorithm compared with the total execution time. It is clear that the merging procedure of SVD algorithm dominates the total execution time. This percentage is 99% when the matrix size reaches 50,000. This phenomenon is mainly due to the fact that when the merging procedure moves toward the upper level of the binary tree, the dimension of matrix doubles, which results in four times increase of memory space for matrix storage and eight times increase of computation time, respectively. Even though the number of merge tasks is halved at the upper level of the binary tree, the computation time of larger matrix significantly increases compared with smaller matrix in the binary tree. Therefore, when the matrix grows to a large size, the height of the binary tree is high, and more computation time for merging tasks is required. Another obvious trend we notice from this figure is the computation time for matrix multiplication step. When the matrix size is 5000, the matrix multiplication operation takes only slightly more than 20% of the total time. However, when the matrix size grows to 50,000, this

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

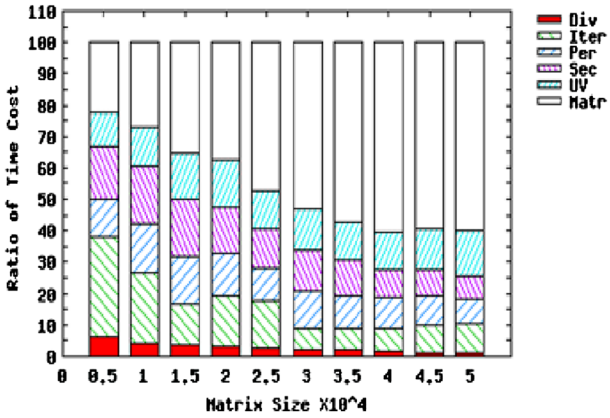

Figure 13. Ratio of time cost for each step.

Table II. Comparison on performance, in seconds, of different types of operations according to different number of mappers (m) and reducers (r).

	Matrix Size						
Step	10,000	20,000	30,000	40,000	50,000	m	r
Div	34	37	39	45	53	10	10
	34	37	40	42	48	20	10
	35	35	41	43	46	30	10
Iter	37	38	38	40	40	10	10
	39	38	38	40	39	20	10
	38	34	37	41	34	30	10
Per	54	56	87	125	143	—	10
	58	59	70	96	131	—	20
	51	51	59	82	115	—	30
Sec	44	45	55	70	85	10	10
	44	44	50	56	68	20	10
	38	39	45	51	58	30	10
UV	37	73	141	231	354	10	10
	40	72	130	214	316	20	10
	40	76	126	188	198	30	10

fraction is 60%. This trend mainly results from the highest time complexity $O(N^3)$ of matrix multiplication operation among all steps. In addition, the characteristics of block matrix multiplication in MapReduce framework determine that large numbers of block matrix replications are required when the matrix size is large enough given a fixed block matrix dimension, which causes large amount of native I/O and network transformation. Other steps including first stage iteration, matrix permutation, secular equation solving, z vector revising, and uv vector computation all achieve high performance, which occupy only small fraction of total execution time.

Table II shows the comparison result on execution time of all steps except matrix multiplication for different settings of mappers or reducers, in which 'm' and 'r' denote the numbers of mappers or reducers we set to run these steps in MapReduce. The input of matrix permutation step is SVD result of two sub-problems, so the size of input file for this step is much larger than any other steps. Consequently, the number of mappers in this step is basically determined by the size of input file

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

based on the running mechanism of MapReduce framework. Therefore, we measure how different settings of reducers affect the performance for this step. For other steps, because the size of input file is much smaller and most of computation work is done in map tasks, we only measure how different setting of mappers affect the performance of these steps. The number of reducers is set to 10 for these steps. From this table, we can clearly see that most steps achieve satisfactory performance. It takes less than 6 min to finish all these steps. Also, we discover that adding more mappers or reducers for these steps does not contribute to the improvement of performance significantly. For matrix size 30,000, the computation time of divide step is quite stable for all settings of mappers, and it does not take much less time for *uv* vector step by adding more mappers. It is because the computation of each task for these steps is quick and adding more map tasks or reduce tasks does not provide more effective computation power. In addition, launching more tasks causes additional overhead including task startup cost and network transfer of intermediate data during the shuffle phase.

6.2. *Adaptability analysis*

In this section, we analyze two important parameters related to our MapReduce-based divide-and-conquer SVD algorithm. First, we measure how different settings of s_1 affect the overall performance of divide-and-conquer SVD algorithm. In this experiment, the setting of mappers and reducers for each step is the same with the setting in Section 6.1. The minimal size of leaf problem is set to 50. Figure 14 shows the results of performances for matrix ranging from 10,000 to 30,000. For all cases, the performances first improve to a considerable extent when s_1 increases and then decline dramatically as s_1 grows to a large size. Given a fixed large matrix size and a small leaf matrix size, when s_1 is small, the number of iterations for first stage is also small, which leads to a situation where a large number of MapReduce jobs are launched to merge small sub-problems when the second stage just begins. When SVD results of small sub-problems are taken as input for a MapReduce job in the second stage, all map tasks and reduce tasks execute very quickly, so the costs of task startup and I/O operation during the job execution dominate the overall performance for one particular job. Thus, these accumulative overhead slow down the performance of overall algorithm. When s_1 is large, the last iteration of first stage parallelization has to deal with the merge task of large sub-problems. Obviously, running the merge process for large sub-problems in a single task of a MapReduce job leads to extremely low parallelization of overall algorithm, which definitely harms the overall performance. Finding the best s_1 for each matrix size is difficult, because the overall performance is influenced by many other parameters, such as number of mappers and reducers. From the experimental results, well balance can be achieved when s_1 is between 1000 and 3000.

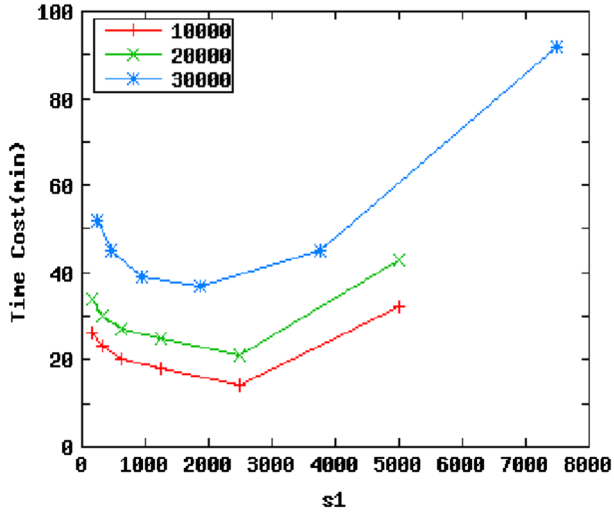

Figure 14. Performance result of different settings of s1.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

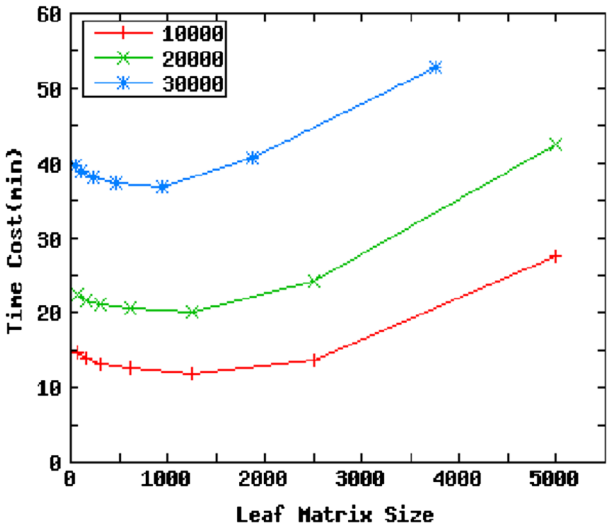

Figure 15. Experiment result of different settings of leaf matrix size.

Figure 15 shows how different settings of leaf matrix size affect the performance of overall divide-and-conquer SVD algorithm. We set s_1 to 2500, and the matrix size we measure here ranges from 10,000 to 30,000. For a fixed size of matrix, different settings of leaf matrix size directly determine the height of the binary tree and, more precisely, the number of levels for the first stage iteration. For matrix size 30,000, the number of iterations required for the first stage is five and one when we set the leaf matrix size to 58 and 938, respectively. We can see from this figure that in different cases, when the leaf matrix size is less than 1000, the gap of performances for these settings is quite small. It is because when the matrix size is below 1000, one merge task for two sub-problems runs quite fast in the first stage. It generally takes less than 1 min for each iteration in the first stage. Even though more iterations consumes slightly more running time compared with total time cost of overall algorithm, the execution time of level by level iterations only occupies a small portion, and the overall performance is not sensitive to the settings of leaf matrix size. However, as the leaf matrix size grows greater than 1500 when all the parallelization work is done in second stage, we notice a sharp increase of total execution time for all matrix size. This phenomenon can be explained by several reasons. First, if leaf matrix is set to a large size, the number of sub-problems in the binary tree is small, leading to a low parallelization of overall algorithm. Second, the computation time of QR iteration is quite sensitive to the size of the matrix. Obviously, when the leaf matrix is large, the computation of QR iteration for leaf matrix becomes the bottleneck of overall algorithm.

6.3. Matrix multiplication performance

In this section, we evaluate the efficiency of our proposed revised block matrix multiplication in MapReduce framework. We first look into how the number of reducers and block size affect the performance of block matrix multiplication in MapReduce framework. Then, we compare the performance of our revised block matrix multiplication with basic block matrix multiplication. Notice that we have to perform two matrix multiplications for the last step of divide-and-conquer SVD algorithm. Therefore, we run two jobs of matrix multiplication simultaneously to measure actual performance of matrix multiplication step of divide-and-conquer SVD algorithm. Figure 16 shows the relationship between the performance and different settings of number of reducers for different size of matrices. The block size is set to 1200. The number of mappers is controlled by MapReduce framework automatically according to the size of input file. The computation time first decreases greatly as the number of reducers increases, because we can achieve more parallelism when we increase the number of reducers. However, as the number of reducers grows bigger, the overall computation time no longer decreases or even increases at certain points. For example, when the number of reducers is set to 100, 200 map tasks will be launched in our environment. Large amount of

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

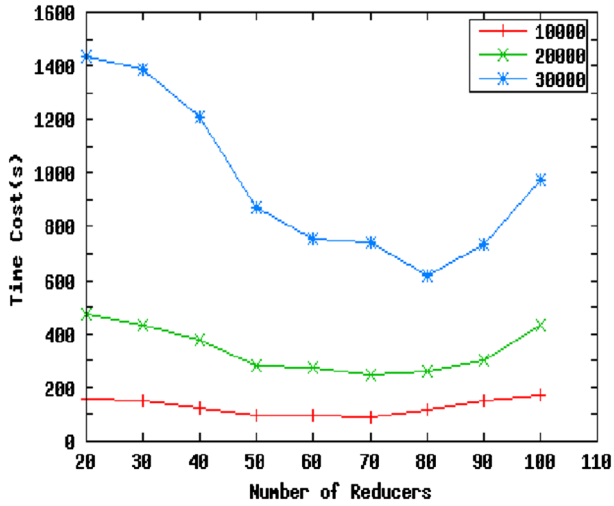

Figure 16. Performance comparison between basic block matrix multiplication and revised block matrix multiplication.

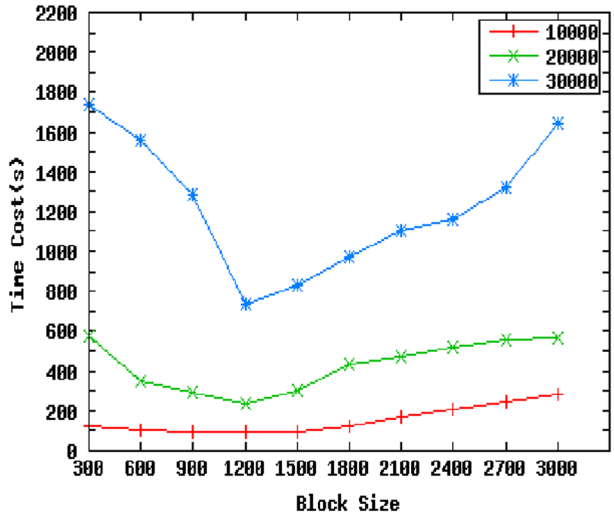

Figure 17. Experiment result for different settings of reducers of revised block matrix multiplication.

reducers being launched means less work assigned to each reducer task but still with fixed startup cost for each reduce task. In addition, large number of reducers running simultaneously causes more network traffic during the shuffle phase and more native I/O competition when the reduce tasks running on the same physical node read their intermediate results from native physical nodes. Figure 17 shows the effect of various size of block matrix on the performance of matrices of different size. The number of reducers is set to 70 for matrices in this experiment. We can see from this figure that the overall computation time decreases first when the block size grows bigger. According to Figure 10, when the block matrix size is relatively small, the number of replication for each block is relatively big. Then, map tasks have to do very heavy replication work for each block and then send the intermediate results to reducers. Hence, native I/O in the physical nodes running map tasks and network transfer has great impact on the performance. Unfortunately, as the block size further increases, we can see that the cost of time start to increase. For a fixed size of matrix, increasing block size results in less parallelism for matrix multiplication and each reducer task has to do more computation work. Therefore, we can conclude that there is a tradeoff between the overhead of map tasks for matrix replication and the parallelism of block matrix computation.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

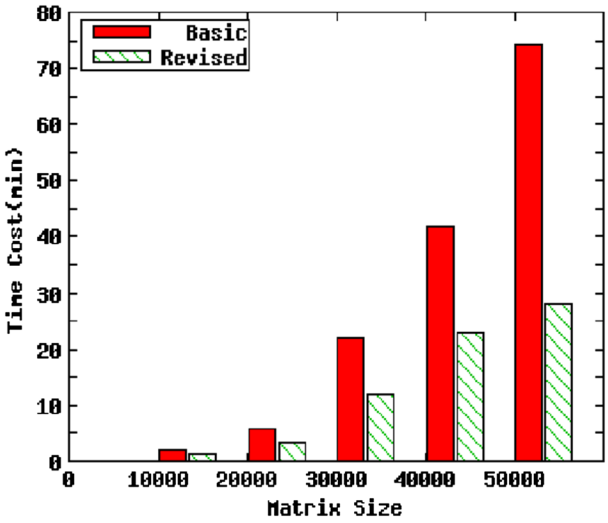

Figure 18. Experiment result for different settings of block size of revised block matrix multiplication.

Figure 18 shows the comparison result of performance between basic block matrix multiplication and our proposed revised block matrix multiplication for different size of matrix. The number of reducers is set to 70, and the size of block matrix is set to 1200. It is clear that our proposed method significantly outperforms basic block matrix multiplication in MapReduce framework. When the matrix size is relatively small, the time cost of our revised block matrix multiplication algorithm is approximately half of the basic block matrix multiplication algorithm. Moreover, for large matrix such as 50,000, basic block matrix multiplication gradually shows its bottleneck in our experiment environment. Recall that for our revised block matrix multiplication, we only take non-zero elements as input of our algorithm. When matrix size is 50,000, 388 map tasks will be launched for two basic block matrix multiplication in MapReduce, which greatly exceeds the total map slots of our experiment environment and causes network congestion, while running our revised block matrix multiplication algorithm calls for 290 map tasks.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a new MapReduce-based approach to solve divide-and-conquer SVD algorithm. Compared with previous implementations of SVD in MapReduce model, our implementation aims to solve full rank SVD computation for general matrices in MapReduce. Our implementation takes advantage of characteristics of divide-and-conquer SVD algorithm and achieves excellent parallelism in MapReduce framework.

Although we adopt Hadoop1.0 as our experiment environment in this paper, our implementation can be easily transplanted to any distributed computing platforms that support MapReduce programming model like Yarn and Spark. Compared with Hadoop1.0 platform, these big data processing platforms have made a lot of improvement to significantly reduce I/O operation during the execution of MapReduce jobs. In addition, for a heterogeneous running environment, it is really interesting to explore a mathematical model to predict the performance of our algorithm according to the diversity of hardware condition of experiment cluster.

ACKNOWLEDGEMENTS

The authors would like to thank anonymous reviewers for their insightful comments, which have helped to improve the manuscript significantly. This work is supported by the National Natural Science Foundation of China under grant nos. 61173170, 61300222, and 61433006, the National High Technology Research and Development Program of China under grant no. 2007AA01Z403, and the Innovation Fund of Huazhong University of Science and Technology under grant nos. 2013QN120, 2012TS052, and 2012TS053.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

REFERENCES

1. Singular value decomposition. (Available from: http://en.wikipedia.org/wiki/Singular_value_decomposition) [Accessed on 20 May 2014].
2. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Operating Systems Design and Implementation*, San Francisco, California, 2004; 137–150.
3. Mahout. (Available from: http://mahout.apache.org) [Accessed on 29 May 2014].
4. Bayramli B. SVD Factorization for tall-and-fat matrices on Map/Reduce architectures, 2013. arXiv preprint arXiv:1310.4664.
5. Melman A. Numerical solution of a secular equation. *Numerische Mathematik* 1995; **69**(4):483–493.
6. Chung J, Knepper S, Nagy JG. Large-scale inverse problems in imaging . *Handbook of Mathematical Methods in Imaging* 2011:43–86.
7. Song F, You J, Zhang D, Xu Y. Impact of full rank principal component analysis on classification algorithms for face recognition . *International Journal of Pattern Recognition and Artificial Intelligence* 2012; **26**(3):1256005.
8. Constantine PG, Gleich DF. Tall and skinny QR factorizations in MapReduce architectures . In *Proceedings of the second international workshop on MapReduce and its applications*. ACM: San Jose, CA, USA, 2011; 43–50.
9. Benson AR, Gleich DF, Demmel J. Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures. *Proceedings of 10th 2013 IEEE International Conference on Big Data*, Santa Clara, CA, USA, 2013; 264–272.
10. Golub G, Van Loan C. *Matrix Computations* (2nd edn). Johns Hopkins University Press: Baltimore, 1989.
11. Drmac Z, Veselic K. New fast and accurate Jacobi SVD algorithm. I. *SIAM Journal on Matrix Analysis and Applications* 2008; **29**(4):1322–1342.
12. Cuppen JJM. A divide-and-conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik* 1981; **36**(2):177–195.
13. Gu M, Eisenstat S. A divide-and-conquer algorithm for the bidiagonal SVD. *Technical Report YALEU/DCS/RR-933*, Yale University, 1992.
14. Gu M, Demmel J, Dhillon I. Efficient computation of the singular value decomposition with applications to least squares problems. *Technical Report CS-94-257*, Department of Computer Science, University of Tennessee, 1994.
15. Anderson M, Ballard G, Demmel J, Keutzer K. Communication-avoiding QR decomposition for GPUs. *International Parallel and Distributed Processing Symposium*, 2011; 48–58.
16. Novakovi V, Singer S. A GPU-based hyperbolic SVD algorithm. *BIT Numerical Mathematics* 2011; **51**:1009–1030.
17. Liu D, Li R, Lilja DJ, Xiao W. A divide-and-conquer approach for solving singular value decomposition on a heterogeneous system . *Proceedings of the ACM International Conference on Computing Frontiers*, Ischia, Italy, 2013; 36.
18. Liu Y, Li M, Hammoud S, Alham NK, Ponraj M. A MapReduce based distributed LSI. *Proceedings of the 7th IEEE International Conference on Fuzzy Systems and Knowledge Discovery* 2010; **6**(2):2978–2982.
19. Yeh C, Peng Y, Lee S. An iterative divide-and-merge based approach for solving large-scale least squares problems. *IEEE Transactions on Parallel and Distributed Systems* 2013; **24**(3):428–438.
20. Constantine PG, Gleich DF. Distinguishing signal from noise in an SVD of simulation data. *Proceedings of 9th IEEE International 1 Conference on Acoustics, Speech and Signal Processing*, Kyoto, Japan, 2012; 5333–5336.
21. Ding Y, Zhu G, Cui C, Zhou J, Tao L. A parallel implementation of Singular Value Decomposition based on Map-Reduce and PARPACK. In *International Conference on Computer Science and Network Technology*, Vol. 2, IEEE: Harbin, Heilongjiang, China, 2011; 739–741.
22. Li RC. Solving secular equations stably and Efficiently. *Technical Report UCB/CSD-94-851*, EECS Department, University of California: Berkeley, 1994.
23. Melman A. A numerical comparison of methods for solving secular equations. *Journal of Computational and Applied Mathematics* 1997; **86**(1):237–249.
24. Bisection. (Available from: http://en.wikipedia.org/wiki/Bisection_method) [Accessed on 17 May 2014].
25. Qian Z, Chen X, Kang N, Chen M, Yu Y, Moscibroda T, Zhang Z. MadLINQ: large-scale distributed matrix computation for the cloud. *Proceedings of the 7th ACM European Conference on Computer Systems*, Bern, Switzerland, 2012; 197–210.
26. A mapreduce algorithm for matrix multiplication. (Available from: http://www.norstad.org/matrix-multiply) [Accessed on 2 June 2014].
27. He B, Fang W, Luo Q, Govindaraju NK, Wang T. Mars: a MapReduce framework on graphics processors. *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Toronto, Canada, 2008; 260–269.
28. Seo S, Yoon EJ, Kim J, Jin S, Kim J-S, Maeng S. Hama: an efficient matrix computation with the mapreduce framework. *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science*, Indianapolis, USA, 2010; 721–726.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe