On Speeding-up Parallel Jacobi Iterations for SVDs

Soumitra Pal   Sudipta Pathak   Sanguthevar Rajasekaran

Computer Science and Engineering, University of Connecticut
371 Fairfield Road, Storrs, CT 06269, USA
{mitra@,sudipta.pathak@,rajasek@engr.}uconn.edu

Abstract—We live in an era of big data and the analysis of these data is becoming a bottleneck in many domains including biology and the internet. To make these analyses feasible in practice, we need efficient data reduction algorithms. The Singular Value Decomposition (SVD) is a data reduction technique that has been used in many different applications. For example, SVDs have been extensively used in text analysis. Several sequential algorithms have been developed for the computation of SVDs. The best known sequential algorithms take cubic time which may not be acceptable in practice. As a result, many parallel algorithms have been proposed in the literature. There are two kinds of algorithms for SVD, namely, QR decomposition and Jacobi iterations. Researchers have found out that even though QR is sequentially faster than Jacobi iterations, QR is difficult to parallelize. As a result, most of the parallel algorithms in the literature are based on Jacobi iterations. JRS is an algorithm that has been shown to be very effective in parallel. JRS is a relaxation of the classical Jacobi algorithm. In this paper we propose a novel variant of the classical Jacobi algorithm that is more efficient than the JRS algorithm. Our experimental results confirm this assertion. We also provide a convergence proof for our new algorithm. We show how to efficiently implement our algorithm on such parallel models as the PRAM and the mesh.

Keywords-SVD; Jacobi iterations; JRS; parallel algorithms

I. Introduction

Singular Value Decomposition (SVD) is a fundamental computational problem in linear algebra and it has application in various computational science and engineering areas. For example, it is widely used in areas such as statistics where it is directly related to principal component analysis, in signal processing and pattern recognition as an essential filtering tool, and in control systems. Recently, it is used as one of the fundamental steps in many machine learning applications such as least square regressions, information retrieval and so on. With the advent of BigData, it has become essential to process data matrices with thousands of rows and columns in real time. The SVD is one of the data reduction techniques. Hence, there is a strong need for efficient sequential and parallel algorithms for the SVD.

SVD takes as input a matrix $A \in \mathbb{F}^{m \times n}$ where $\mathbb{F}$ could be the field of real ($\mathbb{R}$) or complex ($\mathbb{C}$) numbers and outputs three matrices U, S, V such that $A = USV^T$, where $U \in \mathbb{F}^{m \times m}$, $V \in \mathbb{F}^{n \times n}$ are orthogonal matrices (i.e. $U^T U = I_m$, $V^T V = I_n$) and $S \in \mathbb{R}^{m \times n}$ is a diagonal matrix. If $S = diag(\sigma_1, \sigma_2, \ldots \sigma_{\min\{m,n\}})$, then the diagonal elements σ_is are called the singular values of A. The columns of U and V are referred to as the left and right singular vectors respectively. Without loss of generality, we assume that $m \geq n$. We also assume that the input matrices are real, the algorithms in this paper can be easily extended to complex matrices.

There are various methods of computing the SVD [1]. The most commonly used algorithms for dense matrices, which we consider in this paper, can be classified as *QR-based* and *Jacobi-based*. The QR-based algorithms work in two stages. In the first stage the input matrix is converted to a band matrix (bidiagonal, tridiagonal and so on) using factorizations such as Cholesky, LU and QR. In the final stage the band matrix is converted to a diagonal form to obtain the singular values. The singular vectors are also computed accordingly. In the sequential setting, the QR based methods are more frequently used as they are faster than the sequential Jacobi based methods. However, Jacobi-based methods are known to be more accurate [2] and also have a higher degree of potential parallelism. Though there are parallel implementations of QR based algorithms in out-of-core setting [3], [4] as well as in homogeneous multi-core setting [5], our focus is on designing faster parallel implementations of the Jacobi based algorithms.

There are two different variations of the Jacobi based algorithms, *one-sided* and *two-sided*. The two sided Jacobi algorithms are applicable only when A is symmetric and $m = n$. The basic idea of a two sided Jacobi algorithm is as follows. Using a series of plane rotation matrices $U_1, U_2, \ldots, U_t$, the symmetric matrix A is converted a diagonal matrix $S = U_t \ldots U_2 U_1 A U_1^T U_2^T \ldots U_t^T$. The left and right singular vectors are given by $U = V = U_1 U_2, \ldots, U_t$. The basic idea of the one-sided Jacobi algorithms proposed by Hestenes [6] is as follows. Using a series of plane rotation matrices $V_1, V_2, \ldots, V_t$ the input matrix A is first converted to a matrix $B = A V_1 V_2 \ldots V_t$ such that the columns of B are orthogonal. The decomposition $B = US$ gives us the required left singular vectors and the singular values respectively, where U is obtained by normalizing the columns of B (keeping null columns unchanged) and the norm of ith column of B gives S_{ii} of the diagonal matrix S. The right singular vectors are the columns of $V = V_1 V_2, \ldots, V_t$. As the rotation matrices V_is are orthogonal, V is also orthogonal and hence $AV = B = US$ implies $A = USV^T$.

Each of the plane rotation matrices V_i corresponds to a pair of columns (j, k) where the rotation is on the plane going through the jth and the kth axes. We assume $j < k$ and call such a pair (j, k) a *pivot*. In the traditional Jacobi algorithms the rotations are divided among *sweeps*, in each sweep all possible $\binom{n}{2}$ pivots are used. For the one-sided Jacobi, the rotation using pivot (j, k) annihilates the off diagonal entries a_{jk}, a_{kj} to 0. However, either of a_{jk}, a_{kj} may again become non-orthogonal due to some later pivot involving j, k in the same sweep. Hence multiple sweeps are required. As a test for convergence, we could check if the Frobenius norm of the off-diagonal entries of A fell below a certain tolerance. For the two-sided Jacobi, pivoting (j, k) ensures that the columns a_j, a_k become mutually orthogonal after the rotation. Here too, multiple sweeps may be required as the columns may again become non-orthogonal due to some later pivot. For convergence, we could count how many times in any sweep the dot-product $a_j^T a_k$ fell below a certain tolerance and the algorithm is terminated when the count reached $n(n-1)/2$. It is believed that the number S of sweeps needed for the convergence of the sequential one-sided and two-sided Jacobi algorithm is $O(\log n)$ [1].

It turns out that order in which the pivots are applied in a sweep has a significant effect on the number of sweeps required for convergence. Mainly two different orders are used. In the classic Jacobi algorithm [1], each rotation chooses the pivot (j, k) with maximum absolute value of a_{jk} in the two-sided Jacobi and the maximum value of the dot-product $a_j^T a_k$ in the one-sided Jacobi. However, searching for this element is computationally expensive. Cyclic Jacobi algorithms trade off this computation with slower convergence and uses the pivots in the order $(1, 2), (1, 3), \ldots, (1, n), (2, 3), (2, 4), \ldots, (2, n), \ldots (n-1, n)$. It can be shown that after each sweep in one-sided Jacobi the Frobenius norm of off-diagonal entries reduces monotonically and hence the algorithm converges. For the two-sided Jacobi the sum of dot-products of all pairs of columns reduces in each sweep.

Many pivots in a sweep are independent of each other and hence the corresponding rotations can be applied in parallel. In fact the sweeps can be divided into $(n-1)$ *subsweeps* each containing $n/2$ independent pivots and all rotations in a subsweep can be applied in parallel. There are many ways of dividing the pivots in a subsweep. A simple round-robin based ordering is given in [1]. However, it turns out that more clever pivot ordering improves performance by reducing the total number of sweeps required for convergence [7].

A. Contributions

In this paper we introduce a novel algorithm for parallel computation of SVDs. This algorithm uses the idea of picking the pivots in the order of maximum absolute value of a_{jk} in two-sided Jacobi and of $a_j^T a_k$ in one-sided Jacobi. However, to reduce the high computational cost of searching

for the maximum before each rotation, the algorithm uses two relaxations. 1) Instead of searching for the maximum before each iteration, the pivots are sorted in descending values of a_{jk} or $a_j^T a_k$ at the beginning of a sweep and the pivots are applied in this order. 2) To reduce the quadratic sorting time, only the top $1/k$th of the pivots are selected and they are applied in the sorted order. We call this JPS (Jacobi Pivot Sorting) scheme hereafter. Our contributions are as follows. Firstly we show using a simulation based technique [8] that the JPS scheme significantly reduces the number of sweeps. Secondly we give a 'real' implementation of our ideas in a one-sided Jacobi algorithm based on the code from Gnu Scientific Library [9]. We also give a parallel implementation in the shared memory multi-core setting. Finally we discuss the implementation of JPS scheme on various models of computing such as the mesh, the hypercube, and the PRAM.

The remainder of the paper is organized as follows: In Section II, we survey the previous work on Jacobi-SVD algorithm. Section III describes our new JPS algorithm. In Section IV, we show experimental results. Section V discusses implementations of our new algorithms in different models of parallel computing. Finally, we provide some concluding remarks in Section VI.

II. A SURVEY OF BASIC IDEAS

Most existing parallel algorithms for SVD are based on the Jacobi iterations. At the heart of these iterations is the Jacobi rotation (also known as Givens rotation [1]). A rotation of angle θ in the plane going through the axes j, k can be represented as the following matrix:

$$
J(j, k, \theta) = \begin{pmatrix}
1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & & \vdots & & \vdots \\
0 & \cdots & c & \cdots & s & \cdots & 0 \\
\vdots & & \vdots & \ddots & \vdots & & \vdots \\
0 & \cdots & -s & \cdots & c & \cdots & 0 \\
\vdots & & \vdots & & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & \cdots & 0 & \cdots & 1
\end{pmatrix}
\begin{matrix} \\ \\ j \\ \\ k \\ \\ \\ \end{matrix} \quad (1)
$$

where $c = \cos\theta$ and $s = \sin\theta$. It is easy to verify that $J^T J = I$, so Jacobi rotation is an orthogonal transformation.

A. Sequential algorithms

The two-sided Jacobi algorithms attempts to diagonalize the input matrix A by applying a series of Jacobi rotations of the form $B = J^T A J$. It can be easily verified that if A is symmetric then B is also symmetric. Suppose we need to annihilate the element a_{jk} (by symmetry a_{kj} too), then the

effects of the transformation on elements $b_{jj}, b_{jk}, b_{kj}, b_{kk}$ can be written in the matrix form as follows:

$$\begin{pmatrix} b_{jj} & b_{jk} \\ b_{kj} & b_{kk} \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a_{jj} & a_{jk} \\ a_{kj} & a_{kk} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}. \quad (2)$$

Solving for $b_{jk} = b_{kj} = 0$ and taking the smaller root [1], gives

$$c = \frac{1}{\sqrt{1+t^2}}, \qquad s = tc, \qquad \text{where}$$

$$t = \frac{sign(\gamma)}{|\gamma| + \sqrt{\gamma^2 + 1}}, \qquad \text{and} \quad \gamma = \frac{a_{kk} - a_{jj}}{2a_{jk}}. \quad (3)$$

The one-sided Jacobi algorithm, also called Hestenes-Jacobi algorithm [6], converts the input matrix A to a orthogonal matrix B using a series of Jacobi rotations. For a pivot (j, k) the rotation affects only the columns j, k and can be written in the following matrix notation:

$$\begin{pmatrix} b_j & b_k \end{pmatrix} = \begin{pmatrix} a_j & a_k \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}. \quad (4)$$

The two columns would be orthogonalized if $b_j^T b_k = 0$. Thus solving for $b_j^T b_k = 0$ gives the solution

$$c = \frac{1}{\sqrt{1+t^2}}, \qquad s = tc, \qquad \text{where}$$

$$t = \frac{sign(\gamma)}{|\gamma| + \sqrt{\gamma^2 + 1}}, \quad \text{and} \quad \gamma = \frac{a_k^T a_k - a_j^T a_j}{2a_j^T a_k}. \quad (5)$$

As we could see, there is a close similarity between the one-sided and two-sided versions of the Jacobi algorithm, the one-sided algorithm could be thought of as applying two-sided algorithm on the symmetric matrix $A^T A$. In the rest of the paper we will refer to the one-sided algorithm only as it handles more general matrices and is mostly used in the literature.

B. Parallel algorithms

[10]–[12] introduced and implemented the idea of parallel one-sided block jacobi algorithm with dynamic ordering and variable blocking. Their approach, known as OSBJ method, accomplishes the parallel SVD in three stages namely, pre-processing, iteration and post processing. There are two types of pre-processing depending on the dimension of the matrix, namely QR-pre-processing and LQ pre-processing. During QR-pre-processing OSBJ decomposes input matrix $A = Q_1 R$ where $A \in \mathbb{R}^{m \times n}$, orthonormal matrix $Q_1 \in \mathbb{R}^{m \times m}$ and upper triangular matrix $R \in \mathbb{R}^{n \times n}$. On the contrary, LQ-pre-processing decomposes input matrix $A = LQ_2$ where $L \in \mathbb{R}^{n \times n}$ is a lower triangular matrix and $Q_2 \in \mathbb{R}^{\ltimes \times \ltimes}$ is an orthogonal matrix. The L or R matrix in pre-processing stage is considered as input A^0 for the iteration stage. A^0 is partitioned into column blocks. Let, $A_i^{(r)}$ and $A_j^{(r)}$ are one such column block pair where

$1 \leq i < j \leq l$ assuming we have l such column blocks. Weight $\hat{w}_{ij}^r = \frac{||A_i^{(r)T} A_j^{(r)} \mathbf{e}||_2}{||\mathbf{e}||_2}$, where $\mathbf{e} = (1, 1, ... 1)^T \in \mathbb{R}^{\frac{n}{l}}$. $\hat{w}_{ij}^r$ serves as an approximate measure of inclination between $Im(A_i^r)$ and $Im(A_j^r)$. The column block pairs are ordered based on their mutual inclination and the most mutually inclined pair is chosen to be orthogonalized and those columns are excluded from the set for next iteration. The process is repeated until all the column blocks are used up. For each column block pair a 2×2 Gram matrix $G_s \in \mathbb{R}^{\frac{2n}{l} \times \frac{2n}{l}}$ is constructed as is given by $G_s = [A_{i_{r,s}}^{(r)}, A_{j_{r,s}}^{(r)}]^T [A_{i_{r,s}}^{(r)}, A_{j_{r,s}}^{(r)}]$ where $1 \leq s \leq \frac{l}{2}$. This gram matrix is them diagonalized as $G_s = V_s D_s V_s^T$ where V_s is an orthogonal matrix and D_s is a diagonal matrix. The column blocks for next iteration $A^{(r+1)}$ is computed by $[A_{i_{r,s}}^{(r+1)}, A_{j_{r,s}}^{(r+1)}] = [A_{i_{r,s}}^{(r)}, A_{j_{r,s}}^{(r)}]V_s$. Iteration process continues until all the columns are mutually orthogonal.

The Jacobi Relaxation Scheme (JRS) algorithm by Rajasekaran and Song introduced the idea of improving parallelism in SVD computation by multiplying the off-diagonal element in each iteration by a very small number ϵ such that $0 < \epsilon < 1$ instead of setting the off-diagonal element to zero. [8] compares number of iterations taken by JRS algorithm to converge with that of Strumpen's Independent Jacobi algorithm [?] and shows that JRS takes much less number of iterations to converge than Independent Jacobi.

This paper has done same parallel implementation: [13].

This paper seems to do some kind of sorting for [14]. Since any rotation in the two-sided Jacobi algorithm changes only the corresponding (two) rows and (two) columns, and 771 one-sided Jacobi algorithm changes only the corresponding (two) rows, there exists inherent parallelism in the Jacobi iteration algorithms. For example, the n(n 1)/2 rotations in any sweep can be grouped into n 1 rotation sets each of which contains n/2 independent rotations. For instance, if n = 4, there are three rotation sets: (1,2),(3,4), (1,3),(2,4), (1,4),(2,3). Since each rotation can be performed in O(n) time on a single machine, we can perform all the rotations in O(n 2 S) time on a ring of n processors [6]. The idea here is to perform each set of rotations in parallel. We can think of the Jacobi algorithm as consisting of two phases. In the first phase we compute all the rotation matrices (there are O(n 2) of them). In the second phase we multiply them out to get U and V . Consider any rotation operation. The values of s and c can be computed in O(1) time sequentially. The algorithm of Strumpen et al. [11] performs all the n(n 1)/2 rotations of a sweep in parallel even though not all of these rotations are independent. Thus in their algorithm, all the rotation matrices can be constructed in O(1) time using n 2 CREW PRAM processors. This will complete the first phase of the Jacobi algorithm. The second phase has to be completed. This involves the multiplication of O(n 2) rotation matrices. Since two nn matrices can be multiplied in O(log n) time using n 3 CREW PRAM

processors (see e.g. [4,8]), a straightforward implementation of [11]s algorithm runs in time $O(S \log_2 n)$ using n^5 CREW PRAM processors. In [11] an implementation on an $n \times n$ mesh has been given that runs in $O(nS)$ time. However, as has been pointed out before, the value of S is much larger than the corresponding value for the sequential Jacobi iteration algorithm. Any parallel algorithm for SVD partitions the $n(n-1)/2$ rotations of a sweep into rotation sets where each rotation set consists of a number of rotations. All the rotations of a rotation set are performed in parallel. Most of the parallel SVD algorithms in the literature employ $(n-1)$ rotation sets each rotation set consisting of $n/2$ independent rotations. The algorithm of Strumpen et al. is an exception, where multiple processors compute the rotation matrices independently, all the processors employing the same original matrix. In the sequential case, if A is the input matrix, computations will proceed as follows: $B_1 = J_1^T A J_1$; $B_2 = J_2^T B_1 J_2$; $B_3 = J_3^T B_2 J_3$; and so on. On the other hand, in parallel, computations will proceed as follows: $B_1 = J_1^T A J_1$; $B_2 = J_2^T A J_2$; $B_3 = J_3^T A J_3$; etc. The number of B_is computed in parallel will be decided by the number of available processors. Once this parallel computation is complete, all of the computed transformations will be applied to A to obtain a new matrix B. After this, again a parallel computation of rotation matrices will be done all with respect to B; B will be updated with the computed transformations; and so on. In this paper we propose a fundamentally different algorithm for SVD. It is a specific relaxation of the Jacobi iteration algorithm called JRS iteration algorithm. Just like the Jacobi algorithm, there are two variants of the JRS iteration algorithm as well, namely, one-sided and two-sided. We provide details on these two variants in the next subsections.

3.1. Two-sided JRS iteration algorithm The main idea behind the original two-sided Jacobi SVD is to systematically reduce the norm of the off-diagonal elements of a symmetric matrix A: off$(A) = \sum_{i=1}^{n} \sum_{i=1, j=i}^{n} a_{ij}^2$. The convergence of the Jacobi algorithm is ensured by the fact that after each rotation, the norm of the off-diagonal elements decreases by twice the square of the element zeroed out in this rotation [6]. The JRS iteration algorithm also has sweeps and in each sweep we perform rotations (one corresponding to each off-diagonal element). The only difference is that in a given rotation we do not zero-out the targeted off-diagonal element but rather we decrease the value of this element by a fraction. Let the element targeted in a given rotation be the (i, j)th element. Then we let $b_{ij} = a_{ij}(c^2 - s^2) + (a_{ii} - a_{jj})cs = \omega a_{ij}$, where ω is in the interval $[0, 1)$. When $\omega = 0$, we obtain the original Jacobi iteration algorithm. We can solve for s and c as follows: If $a_{ij} = 0$, then set $c = 1$ and $s = 0$; Otherwise $\frac{a_{ii} - a_{jj}}{2a_{ij}} = \frac{c^2 - s^2}{2cs} - \frac{\omega}{2cs}$. Let $\theta = \frac{a_{jj} - a_{ii}}{2a_{ij}}$, $t = \frac{s}{c}$, then $(1 + \omega)t^2 + 2\theta t + \omega - 1 = 0$. According to [6], the smaller root should be chosen, so

sign$(\theta)(1 - \omega)$ $t = \frac{}{} = \frac{}{|\theta| + \sqrt{\theta^2 + (1 - \omega^2)}}$. Like in the regular Jacobi rotation, c and s can be computed as: $c = \frac{1}{\sqrt{1 + t^2}}$, $s = tc$. We call the above algorithm JRS iteration algorithm. Depending on whether it is implemented in sequence or in parallel, there are two variants: JRS sequential algorithm and JRS parallel algorithm. In sequential version, we compute another rotation after applying the rotation previously computed; In a parallel algorithm, we can compute all $n(n-1)/2$ rotations of a sweep independently and then apply these rotations altogether. They are described in the following pseudo codes: JRS sequential algorithm (A, ω, ϵ) //A: Input matrix, ω: relaxation parameter, ϵ: convergence scale. $\epsilon = \omega \|A\|^2$ $B := A$; Repeat for $i = 1$ to $n - 1$ do for $j = i + 1$ to n do Compute Jacobi rotation parameters (c,s) with respect to B_{ij} for B using the formulas in Section 3.1; Let J_{ij} be the corresponding rotation matrix; $B := J_{ij}^T B J_{ij}$; Compute the squared off-diagonal norm of B; Let it be f; until ($f < \epsilon$); JRS parallel algorithm (A, ω, ϵ) //A: Input symmetrical matrix, ω: relaxation parameter, ϵ: convergence scale. $\epsilon = \omega \|A\|^2$ $B := A$; Repeat for $i = 1$ to $n - 1$ do in parallel for $j = i + 1$ to n do in parallel Compute Jacobi rotation parameters (c,s) with respect to B_{ij} for B using the formulas in Section 3.1; Let J_{ij} be the corresponding rotation matrix; $B := J_{(n1)n}^T J_{(n2)n}^T \cdots J_{12}^T B J_{12} \cdots J_{(n2)n} J_{(n1)n}$; Compute the squared off-diagonal norm of B; Let it be f; until ($f < \epsilon$); We can prove the convergence of JRS sequential algorithm in a similar manner to that of the Jacobi algorithm. In particular, we can show that after each rotation the norm of the off-diagonal elements decreases by a constant multiple of the square of the targeted element. However, as in the regular Jacobi algorithm, it is difficult to estimate the number of sweeps needed for convergence. Also, it is hard to prove the convergence of JRS parallel algorithm in theory. Convergence: We can show that the sequential JRS algorithm converges in the limit in exactly the same way we prove the convergence of the sequential Jacobi algorithm. Let A be a given $n \times n$ symmetrical matrix and consider any rotation $J(i, j, \theta)$. As explained in Section 2.1, A gets modified as $B = J^T A J$ due to the rotation $J(i, j, \theta)$. Since the Frobenius norm is preserved under orthogonal transformations, it is the case that $a_{ii}^2 + a_{jj}^2 + 2a_{ij}^2 = b_{ii}^2 + b_{jj}^2 + 2b_{ij}^2 = b_{ii}^2 + b_{jj}^2 + 2\omega^2 a_{ij}^2$. Therefore, off$(B)^2 = \|B\|^2 - \sum_{i=1}^{n} b_{ii}^2 = \|A\|^2 - \sum_{i=1}^{n} a_{ii}^2 + (a_{ii}^2 + a_{jj}^2 - b_{ii}^2 - b_{jj}^2) = $ off$(A)^2 - 2(1 - \omega^2)a_{ij}^2$.

One-sided JRS algorithm is similar to the two-sided Jacobi algorithm. In each rotation, we let the norm of the corresponding two rows be reduced to a fraction of it. That is, $v_i^T v_j = \omega u_i^T u_j$. The solution is similar to the two-sided Jacobi $c = \frac{1}{\sqrt{1 + t^2}}$, $s = tc$. $\frac{1}{1 + t^2}$ where sign$(\theta)(1 - \omega)$, $t = \frac{}{} = \frac{}{|\theta| + \sqrt{\theta^2 + (1 - \omega^2)}}$ and $\theta = \frac{u_j^T u_j - u_i^T u_i}{2u_i^T u_j}$. 3.3. Group JRS iteration algorithm In this section we present a variant of the JRS algorithm. The JRS iteration algorithm is such that all the rotations in a sweep are performed independently and in parallel. The number

of sweeps for convergence is small. Thus JRS algorithm is amenable to maximum parallelism. For example, we can obtain a cost optimal algorithm using upto n 2 processors on a variety of models of computing. As described before, there are n(n 1)/2 rotations in a sweep. These rotations can be grouped into n 1 rotation sets each of which containing n/2 nonconflicting rotations. Given that all the rotations of a sweep can be performed in parallel in the case of the JRS algorithm, a variant of the JRS algorithm can be devised as follows: We partition the n 1 rotation sets into g groups (for some appropriate value of g), each group n n1 having n1 g sets. We compute all the (2 g) rotations in one group in parallel using the JRS algorithm. It is true that there are conflicts among the rotations of any group, but the relaxation scheme helps in reducing the convergence time. Thus all the rotations of a sweep can be completed in g parallel steps. We call this variant as the Group JRS Algorithm, which is described in the following pseudo code: Group JRS Parallel Algorithm (A, ,) //A: Input matrix, : relaxation parameter, : convergence scale. = A 2 F ; B := A; Repeat for each group G do for each element in G do Compute Jacobi rotation parameters (c, s) with respect to this element for B; Apply all the rotation matrices computed for G on B and update B; Compute the squared off-diagonal norm of B; Let it be f ; until (f ¡);

III. JPS ALGORITHM

Algorithm 1: One-sided Sequential JPS

Input: Matrix A, selection parameter τ,
      convergence scale ϵ
Output: SVD matrices U, S, V

1   $\delta \leftarrow \epsilon \times ||A||_F^2$; $B \leftarrow A$; $V \leftarrow I_n$;
2   **repeat**
3     $P \leftarrow$ empty array of pivot tuples $(j, k, b_j^T b_k)$;
4   **for** $j \leftarrow 1$ **to** $n - 1$ **do**
5   **for** $k \leftarrow j + 1$ **to** n **do**
6   insert $(j, k, b_j^T b_k)$ in P;
7   sort P in descending values of dot-products;
8   **if** *largest dot-product in P is $< \delta$* **then**
9   compute singular values S and singular vectors U from B as explained in Section I;
10   **return** U, S, V;
11   **for** $i \leftarrow 1$ **to** $n(n-1)/(2\tau)$ **do**
12      $(j, k, d) \leftarrow P[i]$;
13   compute Jacobi rotation parameters (c, s) with respect to B_{jk} for B using formulas in Section II;
14   let J_{jk} be corresponding rotation matrix;
15      $B \leftarrow B J_{jk}$; $V \leftarrow V J_{jk}$;
16 **until** *maximum sweeps is reached*;

Algorithm 2: One-sided Simulated Parallel JPS

Input: Matrix A, selection parameter τ,
      convergence scale ϵ
Output: SVD matrices U, S, V

1   $\delta \leftarrow \epsilon \times ||A||_F^2$; $B \leftarrow A$; $V \leftarrow I_n$;
2   **repeat**
3     $P \leftarrow$ empty array of pivot tuples $(j, k, b_j^T b_k)$;
4   **for** $j \leftarrow 1$ **to** $n - 1$ **do**
5   **for** $k \leftarrow j + 1$ **to** n **do**
6   insert $(j, k, b_j^T b_k)$ in P;
7   sort P in descending values of dot-products;
8   **if** *largest dot-product in P is $< \delta$* **then**
9   compute singular values S and singular vectors U from B as explained in Section I;
10   **return** U, S, V;
11     $Q \leftarrow$ empty queue of Jacobi rotation matrices;
12   **for** $i \leftarrow 1$ **to** $n(n-1)/(2\tau)$ **do**
13      $(j, k, d) \leftarrow P[i]$;
14   compute Jacobi rotation parameters (c, s) with respect to B_{jk} for B using formulas in Section II;
15   let J_{jk} be corresponding rotation matrix;
16   push J_{jk} in Q;
17   **while** *not empty Q* **do**
18   pop J from Q;
19      $B \leftarrow BJ$; $V \leftarrow VJ$;
20 **until** *maximum sweeps is reached*;

IV. EXPERIMENTAL RESULTS

We implemented our algorithms and tested them for convergence. We had two different implementations. Using the strategy in [8] we compared our algorithms with previous algorithms in terms of number of sweeps using a simulation on a serial computer. We also had an implementation based on the source code of the Jacobi based SVD algorithm in the GNU Scientific Library (GSL) [9].

A. Simulation on a sequential computer

We added a simulation of our algorithm in the list of algorithms simulated in [8] and compared the performances in terms of the number of sweeps. For the two-sided algorithms, we generated random symmetric matrices for different values of n: $500, 750, 1000, 1250, 1500, 1750, 2000$. For the one-sided algorithms we generated random matrices of sizes $500 \times 300, 750 \times 500, 1000 \times 700, 1250 \times 900, 1500 \times 1300, 1750 \times 1500, 2000 \times 1800$. The elements of the matrices were chosen uniformly randomly from the range $[1, 10]$. For each matrix sizes and the algorithms that we considered, we generated 5 random matrices and reported the average number of sweeps used by the algorithms. The convergence

Algorithm 3: One-sided Simulated Group JPS

Input: Matrix A, selection parameter τ,
      convergence scale ϵ

Output: SVD matrices U, S, V

1   $\delta \leftarrow \epsilon \times ||A||_F^2; \quad B \leftarrow A; \quad V \leftarrow I_n;$

2 **repeat**

3     $P \leftarrow$ empty array of pivot tuples $(j, k, b_j^T b_k)$;

4   **for** $j \leftarrow 1$ **to** $n - 1$ **do**

5   **for** $k \leftarrow j + 1$ **to** n **do**

6   insert $(j, k, b_j^T b_k)$ in P;

7   sort P in descending values of dot-products;

8   **if** *largest dot-product in P is $< \delta$* **then**

9   compute singular values S and singular vectors U from B as explained in Section I;

10   **return** U, S, V;

11     $G \leftarrow$ groups of independent pivots from top $n(n-1)/(2\tau)$ elements of P formed greedily;

12     $Q \leftarrow$ empty queue of Jacobi rotation matrices;

13   **foreach** $g \in G$ **do**

14   **for** $(j, k, d) \in g$ **do**

15   compute Jacobi rotation parameters (c, s) with respect to B_{jk} for B using formulas in Section II;

16   let J_{jk} be corresponding rotation matrix;

17   push J_{jk} in Q;

18   **while** *not empty Q* **do**

19   pop J from Q;

20        $B \leftarrow BJ; \quad V \leftarrow VJ;$

21 **until** *maximum sweeps is reached*;

---

condition employed was 10^{-15} times the squared Frobenius norm of the input matrix.

The sequential algorithms we experimented with are as follows. (1) *Cyclic*: a baseline Jacobi implementation using cyclic ordering of the pivots, (2) *JRS*: an implementation of the relaxation scheme used in [8], (3) *TopKFrac*: our algorithm, (4) *TopKFracJRS*: an implementation using the ideas of both JRS and our algorithm. We also experimented with the following parallel algorithms. (5) *ParJRS*, parallel implementation of JRS [8], (6) *GrpJRS*, an improved parallel implementation of JRS [8], (7) *ParTopKFrac*: a parallel implementation of our algorithm, (8) *ParTopKFracJRS*: a parallel implementation using the ideas of both JRS and our algorithm, (9) *GrpTopKFrac*: a parallel implementation of our algorithm using the grouping of independent pivots, (10) *GrpTopKFracJRS*: a parallel implementation using the ideas of both Group JRS and our algorithm.

For the JRS algorithms we used the value of the relxa-tion parameters as recommended in [8], i.e., for two-sided Jacobi $\lambda = 1 - 2.9267n^{-0.4284}$ and for one-sided Jacobi

$\lambda = 1 - 2.2919n^{-0.3382}$. For the 'group' variant of the parallel algorithms, we use the value of $g = (n-1)/\sqrt{n}$.

The results for one-sided and two-sided Jacobi algorithms are shown in Tables I and II respectively. Both the tables show that the number of sweeps taken by our algorithms is significantly less than the JRS algorithms.

B. Experimental recommendation for top-frac parameter k

For a given matrix size, it will be useful to find out the value of k that reduces the number of sweeps. To theoretically determine the best possible value of k seems to be hard. We varied $k = 1, 4, 8, 16, 32$ for the each the appropriate algorithm variants for the matrix sizes sown in Table III.

C. A real implementation on a multicore computer

We also implemented a 'real' version of our algorithms. We changed the basic implementation of one-sided Jacobi in GSL [9] to accommodate our ideas. Note that the GSL implementation is based on the algorithm given in [15].

V. Implementation in Different Parallel Computing Models

VI. Conclusion

In this paper, we have proposed a novel algorithm (called JRS Iteration Algorithm) for computing SVDs. This algorithm enables us to perform all the rotations in a sweep independently and in parallel without increasing the number of sweeps. Thus this algorithm can be implemented on a variety of parallel models of computing to obtain optimal speedups when the processor bound is O(n 2). This method significantly decreases the number of sweeps over independent Jacobi proposed in [11]. Therefore, our method can be used in their stream algorithm to achieve a run time of O(nS). Our algorithm can also be implemented on a CREW PRAM to have a run time of O(S log 2 n). In this paper we have also provided expressions for the relaxation parameter that will result in the minimum number of sweeps.

References

[1] G. H. Golub and C. F. Van Loan, *Matrix computations*. Johns Hopkins University Press, 2012, vol. 3.

[2] J. Demmel and K. Veselic, "Jacobi's method is more accurate than qr," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 4, pp. 1204–1245, 1992.

[3] R. Grimes, H. Krakauer, J. Lewis, H. Simon, and S.-H. Wei, "The solution of large dense generalized eigenvalue problems on the Cray X-MP/24 with SSD," *Journal of Computational Physics*, vol. 69, no. 2, pp. 471–481, 1987.

[4] R. G. Grimes and H. D. Simon, "Solution of large, dense symmetric generalized eigenvalue problems using secondary storage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 14, no. 3, pp. 241–256, 1988.

Table I
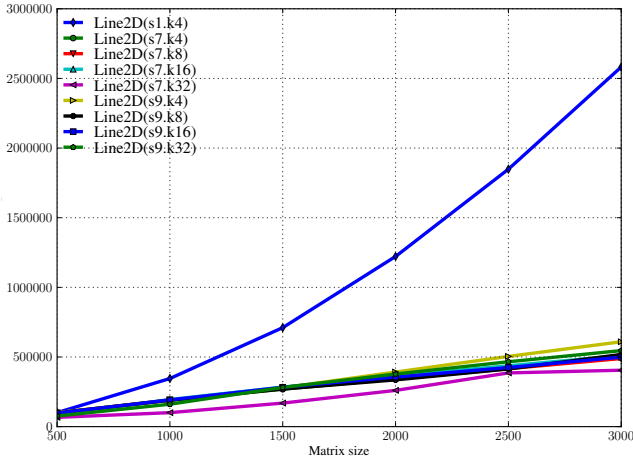NUMBER OF SWEEPS FOR DIFFERENT ALGORITHMS FOR TWO-SIDED JACOBI

Matrix size	Cyclic	TopKFrac	TopKFracJRS	ParJRS	GrpJRS	ParTopKFrac	ParTopKFracJRS	GrpTopKFrac	GrpTopKFracJRS
500×500									
750×750									
1000×1000									
1250×1250									
1500×1500									
1750×1750									
2000×2000									

Table II
NUMBER OF SWEEPS FOR DIFFERENT ALGORITHMS FOR ONE-SIDED JACOBI

Matrix size	Cyclic	TopKFrac	TopKFracJRS	ParJRS	GrpJRS	ParTopKFrac	ParTopKFracJRS	GrpTopKFrac	GrpTopKFracJRS
500×200									
750×500									
1000×700									
1250×900									
1500×1300									
1750×1500									
2000×1800									

Table III
NUMBER OF SWEEPS FOR DIFFERENT ALGORITHMS FOR ONE-SIDED JACOBI

Matrix size	CTopKFrac	TopKFracJRS	ParTopKFrac	ParTopKFracJRS	GrpTopKFrac	GrpTopKFracJRS
500×500						
500×200						
1000×1000						
1000×700						
1500×1300						
1500×1500						
2000×1800						
2000×2000						

(a) Number of updates
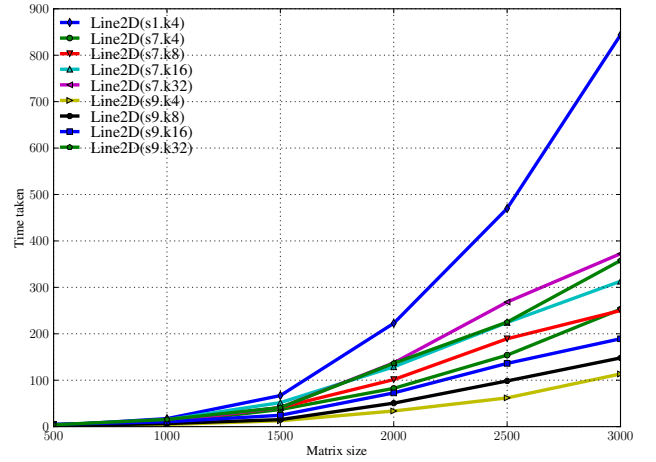
(b) Time taken

Figure 1.   Performance of our parallel implementation

[5] A. Haidar, J. Kurzak, and P. Luszczek, "An improved parallel singular value algorithm and its implementation for multicore hardware," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*.   ACM, 2013, p. 90.

[6] M. R. Hestenes, "Inversion of matrices by biorthogonalization

and related results," *Journal of the Society for Industrial and Applied Mathematics*, vol. 6, no. 1, pp. 51–90, 1958.

[7] M. Becka, G. Oksa, and M. Vajtersic, "Dynamic ordering for a parallel block-jacobi svd algorithm," *Parallel Comput.*, vol. 28, no. 2, pp. 243–262, 2002. [Online]. Available: http://dx.doi.org/10.1016/S0167-8191(01)00138-7

[8] S. Rajasekaran and M. Song, "A relaxation scheme for increasing the parallelism in Jacobi-SVD," *Journal of Parallel and Distributed Computing*, vol. 68, no. 6, pp. 769–777, 2008.

[9] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi, "Gnu scientific library," http://www.gnu.org/software/gsl/, 1996.

[10] M. Bečka and G. Okša, "Parallel one–sided jacobi svd algorithm with variable blocking factor," in *Parallel Processing and Applied Mathematics*. Springer, 2013, pp. 57–66.

[11] S. Kudo, Y. Yamamoto, M. Becka, and M. Vajteršic, "Parallel one-sided block jacobi svd algorithm with dynamic ordering and variable blocking: Performance analysis and optimization," 2016.

[12] M. Becka, G. Okša, and M. Vajteršic, "Parallel Code for One-sided Jacobi-Method," 2015.

[13] M. I. Soliman, "Memory Hierarchy Exploration for Accelerating the Parallel Computation of SVDs," *Neural, Parallel Sci. Comput.*, vol. 16, no. 4, pp. 543–561, Dec. 2008.

[14] B. B. Zhou and R. P. Brent, "On parallel implementation of the one-sided jacobi algorithm for singular value decompositions," in *Parallel and Distributed Processing, 1995. Proceedings. Euromicro Workshop on*. IEEE, 1995, pp. 401–408.

[15] J. C. Nash, "A one-sided transformation method for the singular value decomposition and algebraic eigenproblem," *The Computer Journal*, vol. 18, no. 1, pp. 74–76, 1975.