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. $U^TU = I_m$, $V^TV = I_n$) and $S \in \mathbb{R}^{m \times n}$ is a diagonal matrix. If $S = diag(\sigma_1, \sigma_2, ...\sigma_{\min\{m,n\}})$, 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 factorization 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 $B = AV_1V_2 \ldots V_t$ 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 $V = V_1V_2, \ldots, V_t$. 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 the 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 the 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, and 2) to reduce the quadratic sorting time, only the top $\frac{1}{\tau}$ fraction of the pivots are selected and they are applied in the sorted order. We call this JTS (Jacobi Target Sorting) scheme hereafter. Our contributions are as follows. Firstly we show using a 'simulation' based technique [8] that the JTS scheme significantly reduces the number of sweeps required to reach convergence. Secondly we give a 'real' implementation of our ideas in a one-sided Jacobi algorithm based on the source code from Gnu Scientific Library [9]. We also give a parallel version of this real implementation in the shared memory multi-core setting. Finally we discuss the implementation of JTS scheme on various models of parallel 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 JTS 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:

$$J(j,k,\theta) = \begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \end{array} \\ \begin{array}{ccccccc} \vdots & \ddots & \vdots & & \vdots & & \vdots \end{array} \\ \begin{array}{ccccccc} 0 & \cdots & c & \cdots & s & \cdots & 0 \end{array} \\ \begin{array}{ccccccc} \vdots & & \vdots & \ddots & \vdots & & \vdots \end{array} \\ \begin{array}{ccccccc} 0 & \cdots & -s & \cdots & c & \cdots & 0 \end{array} \\ \begin{array}{ccccccc} \vdots & & \vdots & & \vdots & \ddots & \vdots \end{array} \\ \begin{array}{ccccccc} 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{array} \\ \begin{array}{ccccccc} & & j & & k & & \end{array} \end{array} \begin{array}{l} \\ \\ j \\ \\ k \\ \\ \\ \end{array} \tag{1}$$

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

A. Sequential Jacobi 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 us

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

As we could see, there is a close similarity between the one-sided and the two-sided versions of the Jacobi algorithm, the one-sided algorithm could be thought of as applying the two-sided algorithm on the symmetric matrix $C = A^T A$. 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.

The convergence of the one-sided Jacobi algorithm can be shown by the fact that after each iteration the Frobenius norm of the off-diagonal elements of C reduces a positive quantity and eventually C becomes diagonal which further implies that the columns of A become orthogonal. The original algorithm by Hestenes [6] also maintain the condition that

$$||a_j|| \geq ||a_k|| \qquad \text{for } j < k. \quad (6)$$

When the cyclic pivot ordering is used, this ensures that in the final output the singular values will be sorted in a non-increasing order. Nash [10] gave a faster implementation with two modifications: (i) a rotation is applied only if in addition to (6) the following condition too holds

$$||b_j||^2 - ||a_j||^2 = ||a_k||^2 - ||b_k||^2 \geq 0 \quad (7)$$

and (ii) the algorithm is stopped if for all possible pivots (j,k) in a sweep, the quantity $\frac{a_j^T a_k}{(a_j^T a_j)(a_k^T a_k)}$ falls below a tolerance level.

B. Parallel Jacobi algorithms

The Jacobi algorithm has an inherent parallelism in the sense that many of the $\binom{n}{2}$ pivots are independent of each other and hence can be applied in parallel. More precisely the pivots can be divided into $(n-1)$ subsweeps and in each subsweep $n/2$ independent pivots can be applied in parallel. This applies to both two-sided and one-sided Jacobi algorithms. In the original Hestenes algorithm the $n/2$ pivots in a subsweep is found using a simple round robin ordering [1]. Brent and Luk [11] gave a parallel one-sided algorithm using $O(mnS)$ algorithm on a linear array of n processors, where S is the number of sweeps. Their two-sided algorithm takes $O(nS)$ time on an array of n^2 processors. If the round-robin ordering is used, the conditions (6) and (7) used by the algorithm of Nash [10], do not hold. Zhou and Brent [12] gave two parallel pivot orderings for which for the conditions (6) and (7) hold.

There have been other efforts to improve time taken by the Jacobi algorithms, mainly by (1) reducing time taken by an individual rotation and hence by a sweep, and (2) reducing the number of sweeps required for convergence. To reduce time taken by a sweep, Bečka and Vajteršic [13], [14] divided the input matrix into multiple blocks of columns, establishing orthogonality of columns within the blocks first and then across the blocks. Their block two-sided Jacobi algorithm on hypercubes and rings takes $O(n^2 S)$ time using $O(n)$ processors [13] and $O(nS)$ time algorithm on meshes with n^2 processors [14]. However, the communication complexity is $O(n^2 S)$. Okša and Vajteršic [15] reduced communication by designing a systolic two-sided block-Jacobi algorithm with a run time of $O(nS)$ using n^2 processors. To reduce the number of sweeps required for a parallel two-sided block-Jacobi SVD, Bečka et al. [7] proposed a dynamic ordering of pivots based on the values of the off-diagonal elements. This algorithm takes $O(n^2 S)$ time using $O(n)$ processors. It turns out that first converting the input matrix to a band matrix (bidiagonal, tridiagonal, triangular etc.) using QR and LQ decompositions and then applying Jacobi rotations with suitable post-processing helps in improving the number of sweeps required. Combining the ideas of blocking, dynamic ordering and preprocessing, Vajteršic and his collaborators gave several practical implementations [16]–[18] of the one-sided Jacobi algorithm.

Though the ordering of pivots really affect the number of sweeps required, Strumpen et al. [19] experimented with an idea that is totally on the other extreme. Assuming that there are enough processors ($n \times n$ mesh), they showed that each sweep can be parallelized on these n^2 processors by computing multiple Jacobi rotations independently and applying all the transformations thereafter. Their stream algorithm for one-sided Jacobi that has a run time of $O(n^3 S/p^2)$, where p is the number of processors (p being $O(\sqrt{n})$). Unfortunately their experimental results show that the value of S is much

larger than what the sequential algorithm takes. The large number of sweeps is due to the fact that when independent rotations are applied, each rotation greedily annihilates its own pivot without considering about the effects on other pivots. When these rotations are combined at the end, the global improvements becomes very small.

Rajasekaran and Song [8] presented a novel parallel Jacobi algorithm (JRS) that is a specific relaxation of the parallel Jacobi iterations. Instead of greedily annihilating an off-diagonal element of the matrix in the two-sided Jacobi, JRS reduces the element by a factor λ where $\lambda \in [0, 1)$. This local sacrifice in each of the parallel pivot computations gives huge improvement globally when rotations of all pivots are combined together. For example, when A is of size 100×100, the sequential Jacobi algorithm takes seven sweeps and the algorithm by Strumpen et al. [19] takes more than 300,000 sweeps. On the other hand, JRS iteration takes only 47 sweeps. A variant of the JRS algorithm, called the Group JRS, divides the $n-1$ independent rotation sets (a set is equivalent to a subsweep) into g groups (for some appropriate value of g), each group having $\frac{n-1}{g}$ sets, and computes the $\frac{n}{2} \times \frac{n-1}{g}$ rotations in one group in parallel using the JRS algorithm. The rotations in a group are not totally independent, still the relaxation scheme helps in reducing the convergence time. For the 100×100 example, Group JRS takes only 12 sweeps for $g = \sqrt{n}$.

In this paper we propose another relaxation of parallel Jacobi iterations, which we all Jacobi Target Sorting (in short, JTS). Similar to the dynamic ordering [7], our algorithm also applies Jacobi rotations dynamically based on the values of matrix. However, our algorithm is a relaxation in the sense that it does not apply all the rotations in a sweep, in fact, only the rotations for top $\frac{1}{\tau}$ of all the pivots are applied in the sorted order for some given τ. Our algorithm achieves speedup by avoiding the rotations which comes around the end in the sorted order and do not improve the solution much. Though we used the simulation approach of [8], our idea of relaxation (JTS) is different and performs better than the relaxation (JRS) in [8]. We experimented with the idea of combining JTS and JRS, it does not seem to work. We also give a real implementation of JTS which is not available for JRS.

III. JTS Algorithms

The sequential one-sided Jacobi algorithm using JTS relaxation is shown in Algorithm 1. The inputs to the algorithm are (1) the input matrix A, (2) a selection parameter τ denoting that only the top $\frac{1}{\tau}$ fraction of the pivots are used in a sweep, and (3) a convergence scale parameter ϵ for checking convergence. The algorithm runs multiple sweeps until either the convergence criteria is met or the maximum limit on the number of sweeps is reached. In each sweep, first the top $\frac{1}{\tau}$ fraction of pivots (j, k) are selected depending on the values of the dot-products $b_j^T b_k$. If the dot-product for the topmost pivot is less than ϵ times the Frobenius norm of the input matrix, then all the columns in the current matrix B (same as A at the beginning) are considered to be orthogonal to each other, hence Jacobi iterations are stopped and the output matrices U, V, S are computed accordingly, as explained in Section I. Otherwise, the rotations for selected pivots are applied in the sorted order before starting the next sweep.

Algorithm 1: One-sided Sequential JTS

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

$\delta \leftarrow \epsilon \times ||A||_F^2$; $B \leftarrow A$; $V \leftarrow I_n$;
repeat
  $P' \leftarrow$ empty array of pivot tuples $(j, k, b_j^T b_k)$;
  for $j \leftarrow 1$ ***to*** $n-1$ **do**
    for $k \leftarrow j+1$ ***to*** n **do**
      insert $(j, k, b_j^T b_k)$ in P';
  $P \leftarrow$ top $\frac{1}{\tau}$ fraction of elements of P';
  sort P in descending values of dot-products;
  if *largest dot-product in* P *is* $< \delta$ **then**
    compute singular values S and singular vectors U from B as explained in Section I;
    return U, S, V;
  for *each pivot* $(j, k, d) \in P$ *in sorted order* **do**
    compute Jacobi rotation parameters (c, s) with respect to B_{jk} for B using (5) and let J_{jk} be corresponding rotation matrix;
    $B \leftarrow BJ_{jk}$; $V \leftarrow VJ_{jk}$;
until *maximum sweeps is reached*;

The sequential JTS algorithm can be thought of as consisting of three phases. In the first phase we compute the dot-products $b_j^T b_k$ for all the $\binom{n}{2}$ pivots (j, k) and get the top $\frac{1}{\tau}$ fraction of them in the descending order. In the second phase we compute all the rotation matrices (there are $O(\frac{1}{\tau}n^2)$ of them). In the final phase we multiply all the rotation matrices to get the final value of the orthogonal matrix B. The second phase clearly takes tome $O(n^2)$. Although each rotation in third phase involves multiplication two matrices of size $m \times n$ and $n \times n$ respectively, the rotation changes two columns only and hence take $O(m)$ time. Hence overall time taken in the third phase is $O(mn^2)$. The selection in first phase takes $O(n^2)$ time as it can be done using a selection algorithm on (n^2) elements. The most costly computation in the first phase is sorting the selected pivots which can be done in $O((n^2/\tau)^2 \log(n^2/\tau))$ time. If we select τ such that $\tau = \sqrt{\frac{n^2}{m} \log n}$ then the overall time taken by the first phase as well as the whole sweep would be $O(n^3)$ which is asymptotically no worse than the time taken by a sweep in the cyclic Jacobi algorithm. Moreover, this relaxation reduces the number of sweeps required for convergence, as

we will see in Section IV, and hence reduces the overall time across all the sweeps.

Most of the parallel Jacobi algorithms in the literature partitions the $n(n-1)/2$ rotations of a sweep into $(n-1)$ rotation sets where each rotation set consists of $n/2$ independent rotations. The algorithm of [8], [19] are exceptions, where multiple processors compute the rotation parameters c and s independently, but using the values of the matrix as it was at the beginning of the sweep. In the sequential case, if A is the input matrix, computations will proceed as follows: $B_1 = AJ_1$, $B_2 = B_1J_2$, $B_3 = B_2J_3$, and so on. On the other hand, in parallel computations employed in [8], [19] would proceed as follows: $B_1 = AJ_1$, $B_2 = AJ_2$, $B_3 = AJ_3$, and so on. The number of B_is computed in parallel is decided by the number of available processors. The final effect of all these rotations are computed as $B = AJ_{i_1}J_{i_2}J_{i_3}\dots$ where $i_1i_2i_3\dots$ is some permutation of $123\dots$. If this permutation were exactly the same as the one used in the sequential algorithm, the results would have been the same. However, in general, these two orders are not same, affecting the convergence rate. Nevertheless, the relaxation used in [8] helps in achieving faster convergence than [19]. The same idea of parallelism can applied to sequential JTS algorithm too, as shown in the simulated Algorithm 2 where each of the three phases can be executed in parallel on some parallel computing model. For example, phase 1 consisting of lines 3-8 can be executed using parallel selection and sorting, phase 2 consisting of lines 12-16 has a simple parallel implementation and phase 3 consisting of lines 17-19 can be executed using a binary tree of parallel matrix computations.

As shown in [8] the parallel implementation can be further improved by dividing the rotations for the selected pivots into g rotation-sets each containing $\frac{n(n-1)}{2\tau g}$ rotations and applying these rotations in parallel as done in the parallel JTS algorithm. The details of this Group JTS algorithm are shown in Algorithm 3.

Algorithm 2: One-sided Simulated Parallel JTS

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 $P \leftarrow$ top $\frac{1}{\tau}$ fraction of elements of P';
8 sort P in descending values of dot-products;
9 **if** *largest dot-product in* P *is* $< \delta$ **then**
10 compute singular values S and singular vectors U from B as explained in Section I;
11 **return** U, S, V;
12 $Q \leftarrow$ empty queue of Jacobi rotation matrices;
13 **for** *each pivot* $(j, k, d) \in P$ *in sorted order* **do**
14 compute Jacobi rotation parameters (c, s) with respect to B_{jk} for B using (5) and let J_{jk} be corresponding rotation matrix;
15 $B \leftarrow BJ_{jk}$; $V \leftarrow VJ_{jk}$;
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 implemented our JTS algorithm in the simulation framework used 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: $10, 50, 100, 200, 500, 1000$. For the one-sided algorithms we generated random matrices of sizes $20\times10, 50\times30, 200\times100, 500\times200, 700\times400, 1000\times500$. 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 that was required for convergence. If $\tau > 1$ our JTS algorithms apply less than $\binom{n}{2}$ rotations in a sweep. For better comparison, we normalize the number of sweeps required by an algorithm is total number of rotations required for convergence divided by $\binom{n}{2}$. The convergence scale employed was $\epsilon = 10^{-15}$ and the limit on maximum number of sweeps was set to 3,000.

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) *JTS*: our algorithm, (4) *JRTS*: an implementation using the ideas of both JRS and our algorithm. For JRTS, we computed rotation parameters c and s as given in Section 3.2 in [8] instead of using (5). We also experimented with the following parallel algorithms. (5) *ParallelJRS*, parallel implementation of JRS [8], (6) *GroupJRS*, an improved parallel implementation of JRS [8], (7) *ParallelJTS*: a parallel implementation of our algorithm, (8) *ParallelJRTS*: a parallel implementation using the ideas

Algorithm 3: One-sided Simulated Group JTS

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

$\delta \leftarrow \epsilon \times ||A||_F^2$; $B \leftarrow A$; $V \leftarrow I_n$;
repeat
  $P' \leftarrow$ empty array of pivot tuples $(j, k, b_j^T b_k)$;
  for $j \leftarrow 1$ ***to*** $n-1$ **do**
    for $k \leftarrow j+1$ ***to*** n **do**
      insert $(j, k, b_j^T b_k)$ in P';
  $P \leftarrow$ top $\frac{1}{\tau}$ fraction of elements of P';
  sort P in descending values of dot-products;
  if *largest dot-product in* P *is* $< \delta$ **then**
    compute singular values S and singular vectors U from B as explained in Section I;
    return U, S, V;
  $G \leftarrow$ groups of pivots from P;
  foreach $g \in G$ **do**
    $Q \leftarrow$ empty queue of Jacobi rotation matrices;
    for $(j, k, d) \in g$ **do**
      compute Jacobi rotation parameters (c, s) with respect to B_{jk} for B using (5) and let J_{jk} be corresponding rotation matrix;
      push J_{jk} in Q;
    while *not empty* Q **do**
      pop J from Q;
      $B \leftarrow BJ$; $V \leftarrow VJ$;
until *maximum sweeps is reached*;

of both JRS and our algorithm, (9) *GroupJTS*: a parallel implementation of our algorithm using the grouping of independent pivots, (10) *GroupJRTS*: a parallel implementation using the ideas of both Group JRS and our algorithm.

For the JRS algorithms we used the value of the relaxation 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, except for ParallelJTS, are shown in Tables I and II respectively. ParallelJTS has the same drawback as the algorithm in [19] as both methods apply pivot rotations greedily. In our experiments, except for a few smaller sizes, ParallelJTS did not converge in 3,000 sweeps. However, the results clearly show that sequential JTS outperforms Cyclic, Group JTS almost matches the performance of sequential JTS and outperforms GroupJRS. Results for three JRTS algorithms suggest that both the relaxations JRS and JTS do not work well together. The reason could be that JTS thrives on larger values of the top pivots, however, JRS with $\lambda > 0$ somewhat reduces that advantage.

B. Experimental recommendation for selection parameter τ

For a given matrix size, it will be useful to find out the value of τ that reduces the number of sweeps. To theoretically determine the best possible value of τ seems to be hard. We varied $\tau = 1, 2, 4, 8, 16, 32, 64$ and experimented with JTS and GroupJTS versions as they seem to be the best sequential and parallel algorithms, respectively. The results on varying τ are shown in Tables III and IV. In both the sequential and parallel versions we see that increasing τ initially improves performance as only the best pivots are applied. However, for very large value of τ, JTS misses many 'good' pivots. It seems that $\tau = 32$ is a good choice for the matrix sizes we experimented, and possibly larger for bigger matrices. However, it should also be noted that as we increase τ we need to more frequently do selection and sorting of top pivots, which will increase run-time.

C. A real implementation on a multi-core computer

We also implemented a 'real' version of our algorithms. We changed the basic implementation of one-sided Jacobi in GSL [9] (based on the algorithm in [10]) to accommodate JTS relaxation and experimented on a Dell Precisions Workstation T7910 running RHEL 7.0 on two sockets each containing 8 Dual Intel Xeon Processors E5-2667 (8C HT, 20MB Cache, 3.2GHz) and 256GB RAM. For our experiments we used only one of the two sockets. In this implementation, the groups of rotation sets are identified by using a greedy matching algorithm on the top selected pivots: in the order descending pivots, go on taking pivots which are independent of the pivots already picked. When no more pivots cannot be picked, we apply them in parallel using the available $p = 16$ cores. The experimental results on random matrices of sizes $n \times n$, $n = 500, 1000, 1500, 2000$ are shown in Figure 1.

V. Implementation on Different Parallel Computing Models

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero,

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

Matrix	Cyclic	JRS	JTS	JRTS	ParallelJRS	ParallelJRTS	GroupJRS	GroupJTS	GroupJRTS
10x10	5.0	9.0	4.0	8.0	18.5	8.0	10.0	4.0	8.0
50x50	7.0	23.0	4.0	20.0	29.5	20.0	23.0	5.0	20.0
100x100	7.5	34.0	5.0	29.0	43.0	29.0	34.0	5.0	29.0
200x200	8.0	49.0	5.0	42.0	58.5	42.0	49.0	6.0	42.0
500x500	9.0	77.0	6.0	64.0	89.5	64.0	76.0	6.0	64.0
1000x1000	9.0	106.0	6.0	88.0	123.0	88.0	106.0	7.0	88.0

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

Matrix	Cyclic	JRS	JTS	JRTS	ParallelJRS	ParallelJRTS	GroupJRS	GroupJTS	GroupJRTS
20x10	7.5	23.0	4.0	15.0	204.0	15.0	27.0	4.0	15.0
50x30	9.0	39.5	4.0	26.0	86.5	27.0	40.0	5.0	27.0
200x100	10.0	76.0	4.0	40.5	78.5	41.0	75.0	5.0	41.0
500x200	10.0	109.5	4.0	47.0	110.5	48.0	109.5	5.0	47.0
700x400	12.0	124.0	5.0	65.0	119.5	66.0	124.5	7.0	65.0
1000x500	11.5	142.5	5.0	67.0	141.5	69.0	142.0	6.0	67.0

Table III
NUMBER OF SWEEPS FOR DIFFERENT τ FOR TWO-SIDED JTS

Matrix	JTS							GroupJTS						
	τ=1	τ=2	τ=4	τ=8	τ=16	τ=32	τ=64	τ=1	τ=2	τ=4	τ=8	τ=16	τ=32	τ=64
10x10	4.5	4.0	4.0	4.0	5.0	5.0	47.0	6.8	5.0	4.0	4.0	5.0	5.0	47.0
50x50	5.5	5.0	4.0	4.0	4.0	4.0	4.0	8.5	6.0	5.0	4.0	4.0	4.0	4.0
100x100	6.0	5.0	5.0	4.0	4.0	4.0	4.0	10.0	7.0	5.0	4.8	4.0	4.0	4.0
200x200	7.0	6.0	5.0	5.0	4.0	4.0	4.0	11.2	7.0	6.0	5.0	4.0	4.0	4.0
500x500	7.2	6.0	6.0	5.0	5.0	4.0	4.0	2265.5	8.0	6.0	5.0	5.0	4.0	4.0
1000x1000	8.0	7.0	6.0	5.0	5.0	4.0	4.0	3000.0	9.0	7.0	6.0	5.0	4.0	4.0

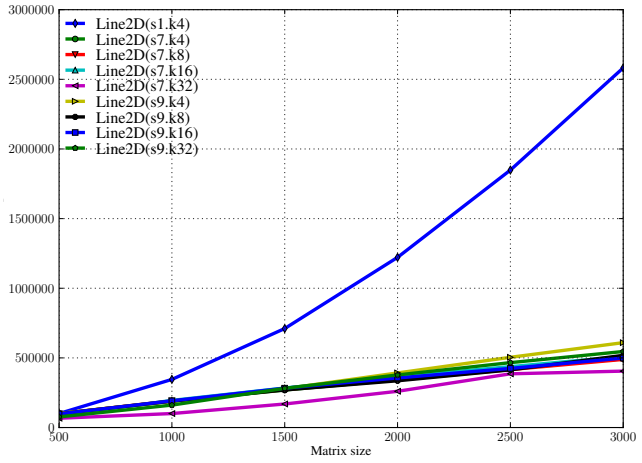
Table IV
NUMBER OF SWEEPS FOR DIFFERENT τ FOR ONE-SIDED JTS

Matrix	JTS							GroupJTS						
	τ=1	τ=2	τ=4	τ=8	τ=16	τ=32	τ=64	τ=1	τ=2	τ=4	τ=8	τ=16	τ=32	τ=64
20x10	6.0	4.0	4.0	4.0	4.0	4.0	5.0	18.8	6.0	4.2	4.0	4.0	4.0	5.0
50x30	6.5	5.0	4.0	4.0	4.0	4.0	4.0	26.0	7.5	5.0	4.8	4.0	4.0	4.0
200x100	7.0	5.0	4.0	4.0	4.0	4.0	4.0	33.5	14.0	5.0	4.0	4.0	4.0	4.0
500x200	8.0	5.0	4.0	4.0	3.0	3.0	3.0	3000.0	16.5	5.0	4.0	4.0	3.0	3.0
700x400	8.5	6.0	5.0	5.0	4.0	4.0	4.0	3000.0	19.2	7.0	5.0	5.0	4.0	4.0
1000x500	9.0	6.0	5.0	4.0	4.0	4.0	4.0	3000.0	169.0	6.0	5.0	4.0	4.0	4.0

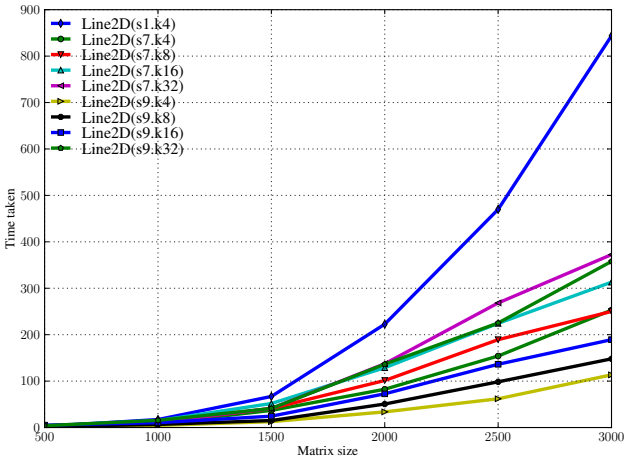
pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

VI. CONCLUSION

In this paper, we have proposed and implemented a novel algorithm (called JTS Algorithm) for computing SVDs in parallel. This algorithm improved number of sweeps required by applying only a top $\frac{1}{\tau}$ fraction of possible Jacobi rotations in a sweep. We showed that our algorithm works better than known algorithms on both a simulated framework as used in [8] as well as on a real implementation based on the code in GNU Scientific Library [9]. We also gave theoretical results on several parallel computing models. During our experimentation on the real implementation we found that the cache miss plays a big role as the selection pivots and application of rotation requires computing dot-product of each column with all other. We would like to explore ideas from [20] and [5] for a better cache efficient implementation.



(a) Number of updates



(b) Time taken

Figure 1. Performance of our parallel implementation

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.

[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. Bečka, G. Okša, and M. Vajteršic, "Dynamic Ordering for a Parallel block-Jacobi SVD Algorithm," *Parallel Computing*, vol. 28, no. 2, pp. 243–262, 2002.

[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] 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.

[11] R. P. Brent and F. T. Luk, "The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays," *SIAM Journal on Scientific and Statistical Computing*, vol. 6, no. 1, pp. 69–84, 1985.

[12] 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.

[13] M. Bečka and M. Vajteršic, "Block-Jacobi SVD Algorithms for Distributed Memory Systems I: Hypercubes and Rings," *Parallel Algorithms and Application*, vol. 13, no. 3, pp. 265–287, 1999.

[14] ——, "Block-Jacobi SVD Algorithms for Distributed Memory Systems II: Meshes," *Parallel Algorithms and Application*, vol. 14, no. 1, pp. 37–56, 1999.

[15] G. Okša and M. Vajteršic, "A systolic block-Jacobi SVD solver for processor meshes," *Parallel Algorithms and Applications*, vol. 18, no. 1-2, pp. 49–70, 2003.

[16] 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.

[17] 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.

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

[19] V. Strumpen, H. Hoffmann, and A. Agarwal, "A Stream Algorithm for the SVD," Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, East Lansing, Michigan, Tech. Rep. Technical Memo 641, October 2003.

[20] 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.