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. The best known sequential algorithms for the computation of SVDs 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. For example, the Jacobi Relaxation Scheme (JRS) of the classical Jacobi algorithm has been shown to be very effective in parallel. 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. The key idea behind our algorithm is to select the pivot elements for each sweep appropriately. We also 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 applications 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 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 an out-of-core setting [3], [4] as well as in homogeneous multi-core setting [5], our focus in this paper 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 $S = U_t \ldots U_2U_1AU_1^TU_2^T \ldots U_t^T$. The left and right singular vectors are given by $U = V = U_1U_2, \ldots, U_t$. 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 two-sided Jacobi, the rotation using the 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 fall below a certain tolerance. For the one-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 algorithms 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 the 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 use 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 two-sided Jacobi the Frobenius norm of the off-diagonal entries reduces monotonically and hence the algorithm converges. For the one-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. Our algorithm is faster than the classical Jacobi algorithm in sequence. 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) in each sweep 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 Selection) 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 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:

$$J(j,k,\theta) = \begin{array}{cc}
\left(\begin{array}{ccccccc} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{array}\right) & \begin{array}{c} \\ \\ j \\ \\ k \\ \\ \\ \end{array} \\
\begin{array}{ccccccc} & & j & & k & & \end{array} &
\end{array} \qquad \begin{array}{c} (1) \\ \\ (2) \end{array}$$

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 attempt to diagonalize the input matrix A by applying a series of Jacobi rotations of the form $B{=}J^T AJ$. It can be easily verified that if A

is symmetric then B is also symmetric. The effect of the rotation on $b_{jj}, b_{jk}, b_{kj}, b_{kk}$ can be written as follows:

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

Suppose we need to annihilate the element b_{jk} (by symmetry b_{kj} too). Solving for $b_{jk}{=}b_{kj}{=}0$ and taking the smaller root [1], gives

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

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 of the form $B = AJ$. The rotation affects only the columns j, k and can be written in the following matrix notation:

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

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

$$c = \frac{1}{\sqrt{1+t^2}}, \qquad s = tc, \qquad \text{where}$$
$$t = \frac{sign(\gamma)}{|\gamma| + \sqrt{\gamma^2+1}}, \quad \text{and} \quad \gamma = \frac{a_k^T a_k - a_j^T a_j}{2a_j^T a_k}. \qquad (3)$$

There is a close similarity between the one-sided and the two-sided versions as 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 by a positive quantity and eventually C becomes diagonal. This implies that the columns of A have become orthogonal. The original algorithm by Hestenes [6] also maintains the condition that

$$||a_j|| \geq ||a_k|| \qquad \text{for } j < k. \qquad (4)$$

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 (4) the following condition too holds

$$||b_j||^2 - ||a_j||^2 = ||a_k||^2 - ||b_k||^2 \geq 0 \qquad (5)$$

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 the rotations for them 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 targeted for rotation 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 are found using a simple round robin ordering [1]. Brent and Luk [11] gave a parallel one-sided algorithm that runs in $O(mnS)$ time 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. The conditions (4) and (5) used by the algorithm of Nash [10] do not hold if the round-robin ordering is used. To circumvent this problem Zhou and Brent [12] gave two parallel pivot orderings.

There have been other efforts to speed up Jacobi algorithms, mainly by (1) reducing the time taken by a single sweep, and (2) reducing the number of sweeps required for convergence. To reduce the 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 two-sided block Jacobi algorithm on hypercubes and rings takes $O(n^2S)$ time using $O(n)$ processors [13] and $O(nS)$ time on meshes with n^2 processors [14]. However, the communication complexity is $O(n^2S)$. Okša and Vajteršic [15] reduced the 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 $O(n^2S)$ 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 reduces the number of sweeps. Combining the ideas of blocking, dynamic ordering and preprocessing, Vajteršic and his collaborators gave several practical implementations [16], [17].

Though the ordering of pivots really affects the number of sweeps required, Strumpen et al. [18] 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 has a run time of $O(n^3S/p^2)$, where p^2 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 the other pivots. When these rotations are combined at the end, the global improvement become 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's value by a factor λ where $\lambda \in [0, 1)$. This local sacrifice in each of the parallel pivot computations gives a huge improvement globally when the rotations of all the pivots are combined together. For example, when A has size 100×100, the sequential Jacobi algorithm takes 7 sweeps and the algorithm by Strumpen et al. [18] 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, takes only 12 sweeps.

In this paper we propose another relaxation of parallel Jacobi iterations, which we call Jacobi Target Selection (in short, JTS). Similar to the dynamic ordering [7], our algorithm also applies Jacobi rotations dynamically based on the values in the matrix. However, our algorithm is a relaxation in the sense that it does not apply rotations for all the pivots in a sweep. In any sweep only the top $\frac{1}{\tau}$ fraction of all the pivots are targeted for rotation, for some appropriate value of τ. Our algorithm achieves speedups by avoiding the rotations which come around the end in the sorted order and do not improve the solution much. Our idea of relaxation (JTS) is different from and performs better than the relaxation (JRS) in [8]. We have experimented with the idea of combining JTS and JRS, but 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) a matrix A, (2) a selection parameter τ such that only the top $\frac{1}{\tau}$ fraction of possible pivots are used in a sweep, and (3) a convergence scale parameter ϵ. The algorithm runs multiple sweeps until either the convergence criterion is met or the maximum limit on the number of sweeps is reached. In each sweep, first the top $\frac{1}{\tau}$ fraction of all possible 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 (the 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 the selected pivots are applied in the sorted order before starting the next sweep.

Algorithm 1: One-sided Sequential JTS

```
Input: Matrix $A$, selection parameter $\tau$,
       convergence scale $\epsilon$
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 $S, U$ 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         $(c, s) \leftarrow$ rotation parameters for pivot $(j, k)$
           using (3) and push $J(j, k, \theta)$ in $Q$;
15     while not empty $Q$ do
16         pop $J$ from $Q$;   $B \leftarrow BJ$;   $V \leftarrow VJ$;
17 until maximum sweeps is reached;
```

The sequential JTS algorithm can be thought of as consisting of three phases. In phase 1 (lines 3-8) 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 descending order. In phase 2 (lines 12-14) we compute all the rotation matrices (there are $O(\frac{1}{\tau}n^2)$ of them). In phase 3 (lines 15-16) we multiply all the rotation matrices to get the final value of the orthogonal matrix B. Phase 2 clearly takes time $O(n^2)$. Although each rotation in phase 3 involves the multiplication of two matrices of sizes $m \times n$ and $n \times n$, respectively, the rotation changes two columns only and hence takes $O(m)$ time. Hence the overall time taken in phase 3 is $O(mn^2)$. The selection in phase 1 takes $O(n^2)$ time as it can be done using a selection algorithm on $O(n^2)$ elements. The most costly computation in phase 1 is sorting the selected pivots which can be done in $O((n^2/\tau)\log(n^2/\tau))$ time. The overall time taken by a single sweep is $O(mn^2)$ (under the assumption that $m = \omega(\log n)$) which is asymptotically no worse than the time taken by a sweep in the cyclic Jacobi algorithm. Moreover, JTS reduces the number of sweeps, and hence overall time, as we will see in Section IV.

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], [18] are exceptions, where multiple processors compute the rotation parameters c, 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], [18] 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 is $B = AJ_{i_1}J_{i_2}J_{i_3}\ldots$ where $i_1i_2i_3\ldots$ is some permutation of $123\ldots$. 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 the same, affecting the convergence rate. Nevertheless, the relaxation used in [8] helps in achieving faster convergence than [18]. We will see in Section IV that the same idea of parallelism when applied to the sequential JTS algorithm, i.e., when phase 1 is executed using parallel selection and sorting, phase 2 using a simple parallel implementation and phase 3 using a parallel matrix multiplications, does not work.

However, like the 'group' variant of JRS, the group variant of JTS seem to work well in practice. In this variant, the selected top pivots are divided into g rotation sets each containing $\frac{n(n-1)}{2\tau g}$ rotations and applying these rotations in parallel similar to JRS. The details of this group JTS are shown in Algorithm 2. There could be multiple ways of forming the rotation sets in line 12. One simple way, as done in group JRS, could be to include in a rotation set all the $(n-1)/g$ sets of $n/2$ independent pivots found using the round-robin algorithm in [1] provided the pivots are in the top $\frac{1}{\tau}$ fraction of pivots. The pivots in a group are not totally independent, still JTS significantly reduces number of sweeps. It seems hard to prove convergence of group JTS.

Algorithm 2: One-sided Parallel Group JTS

```
   Input: Matrix $A$, selection parameter $\tau$,
          convergence scale $\epsilon$
   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 $S, U$ from $B$ as explained in Section I;
11         return $U, S, V$;
12     $G \leftarrow$ groups of pivots from $P$;
13     foreach $g \in G$ do
14         $Q \leftarrow$ empty queue of Jacobi rotation matrices;
15         for $(j, k, d) \in g$ do
16             $(c, s) \leftarrow$ rotation parameters for pivot $(j, k)$
               using (3) and push $J(j, k, \theta)$ 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 the 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

Following [8] we simulated our algorithms and compared the performances in terms of number of sweeps. The parallel Jacobi rotations of all pivots were simulated on a serial computer by storing the rotation matrices separately finally multiplying those matrices in cyclic order. 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 $\tau{>}1$ the JTS algorithm and its variants apply less than $\binom{n}{2}$ rotations in a sweep. For a better comparison, we normalize the number of sweeps required by an algorithm as the total number of rotations required for convergence divided by $\binom{n}{2}$. Convergence scale $\epsilon = 10^{-15}$ and a maximum of 3,000 sweeps were used.

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 JTS. For JRTS, we computed rotation parameters c and s as given in Section 3.2 in [8] instead of using (3). We also experimented with the following parallel algorithms. (5) *ParallelJRS*: a parallel implementation of JRS [8], (6) *ParallelJTS*: a parallel implementation of JTS as discussed in Section III, (7) *ParallelJRTS*: a parallel implementation using both the ideas JRS and JTS, (8) *GroupJRS*, (9) *GroupJTS* and (10) *GroupJRTS*, where the last three are 'group' parallel implementations of JRS, JTS, JRTS, respectively.

For the JRS relaxation based algorithms we used the value of the relaxation parameter λ as recommended in [8], i.e., $1 - 2.9267n^{-0.4284}$ for two-sided Jacobi and $1-2.2919n^{-0.3382}$ for one-sided Jacobi. For the 'group' variant of the parallel algorithms, we use the value of $g = (n-1)/\sqrt{n}$. For JTS variants we used the value of the selection parameter $\tau = 4$.

Table I shows the results for the experimented algorithms, except ParallelJTS. The square matrices were run using two-sided Jacobi and the rest using one-sided Jacobi. ParallelJTS has the same drawback as the algorithm in [18] 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; GroupJTS almost matches the performance of sequential JTS and outperforms GroupJRS. Results for three JTS variants suggest 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. *Empirical recommendation for the 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 τ=$1, 2, 4, 8, 16, 32, 64$ 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 Table II. In both the sequential and parallel versions, increasing τ initially improves the performance as only the best pivots are applied. However, for very large value of τ, JTS misses many 'good' pivots. For τ=1, GroupJTS has no relaxation, and like the algorithm in [18] takes a large number of sweeps. It seems τ=32 is a good choice for the matrix sizes we used, 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 the top pivots, which will increase the run-time.

C. *A real implementation on a multi-core computer*

We also implemented 'real' versions of our algorithms. We changed the implementation of one-sided Jacobi in GSL [9] (based on the algorithm in [10]) to accommodate JTS relaxation and experimented on one of the two sockets each containing 8 Dual Intel Xeon Processors E5-2667 (20MB Cache, 3.2GHz) and 256GB RAM in a Dell Precisions Workstation T7910 running RHEL 7.0. In this implementation, the rotation sets are identified using a greedy matching algorithm on the selected top pivots: in the descending order of top pivots, go on picking pivots which are independent of the pivots already picked. When no more pivot can be picked in a group, we apply the grouped pivots in parallel using the available p cores. Subsequent groups are formed from the remaining top pivots selected for the current sweep. The experimental results on $n \times n$ random matrices, $n = 500, 1000, \ldots, 3000$ are shown in Figure 1. The GSL implementation skips the rotations which do not satisfy conditions (4) and (5). In Figures 1(a) and 1(b) we report the number of rotations that were actually applied and the time taken, respectively, by the following three implementations: (1) the baseline GSL, (2) our modification for JTS, and (3) a shared memory multi-core implementation of JTS using p=16 OpenMP threads. On the same plots we show the results of the JTS implementations for different values of τ=$1, 2, 4, 8, 16$. Clearly the JTS implementations outperform the baseline GSL. Both the sequential and the multi-core implementations take the least time for τ=4. The scaling factor of our multi-core implementation (time taken using 1 thread divided by time taken using p threads) for different n and $\tau = 4$ are shown in Figure 1(c). For n=2000 our algorithm achieves scaling of almost 8 using 16 threads. *Remark*: In the 'real' implementation all pivots in a rotation set are independent of each other. Thus, unlike GroupJRS, the convergence of real implementation can be easily proved.

V. Implementation on the PRAM and the Mesh

In this section we implement JTS on various parallel models of computing including the PRAM and the mesh. A Parallel Random Access Machine (PRAM) is a collection of RAMs working in synchrony where communication takes place with the help of a common block of shared memory [19], [20]. For instance if processor i wants to communicate with processor j it can do so by writing a message in memory cell j which can then be read by processor j. More than one processor may want to either write into or read from the same cell at the same time. Depending on how these conflicts are handled, a PRAM can further be classified into three: Exclusive Read and Exclusive Write (EREW) PRAM, Concurrent Read and Exclusive Write (CREW) PRAM, and Concurrent Read and Concurrent Write (CRCW) PRAM.

An $n \times n$ mesh can be represented as a directed $n \times n$ grid-graph whose nodes correspond to processing elements and whose edges correspond to bidirectional communication links [20], [21]. If two processors are connected by an edge, they can communicate in a unit step. Otherwise, they communicate by sending a message along a connecting path.

A. *Some Fundamental Parallel Algorithms*

In this section we describe some basic algorithms that will be employed in the parallel implementation of JTS. These algorithms can be found in relevant texts (such as [19]–[21]).

Let $\oplus$ be any associative unit-time computable binary operator defined in some domain Σ. Given a sequence of n elements $k_1, k_2, \ldots, k_n$ from Σ, the problem of *prefix computation* is to compute $k_1, k_1 \oplus k_2, \ldots, k_1 \oplus k_2 \oplus \cdots \oplus k_n$.

Lemma 1: Prefix computation on a sequence of n elements can be performed in $O(\log n)$ time using n CREW PRAM processors.

Lemma 2: Prefix computation on a sequence of n^2 elements can be performed in $O(n)$ time on a $n \times n$ mesh.

Lemma 3: Any PRAM algorithm that runs in time T using P processors can be simulated on a P'-processor machine in time $O\left(\frac{PT}{P'}\right)$, for any $P' \leq P$. $\square$

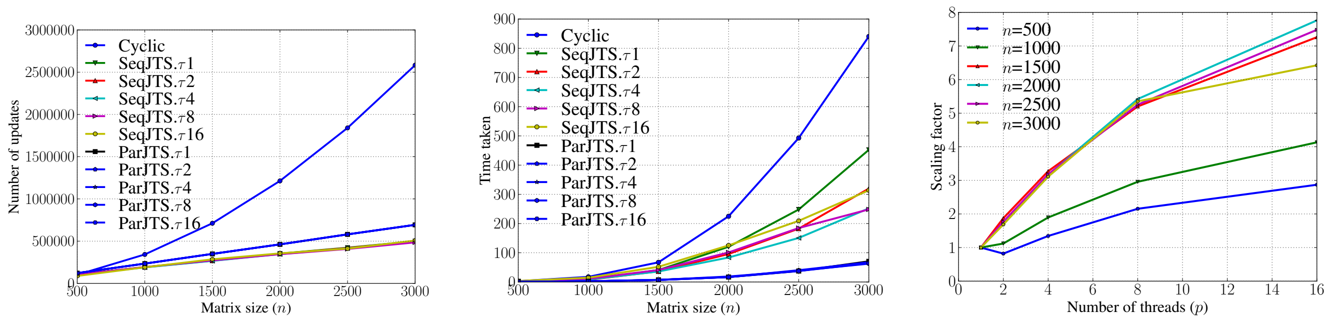
Lemma 4: Any sequence of arbitrary elements of length n can be sorted in $O(\log n)$ time using n EREW PRAM processors. $\square$

Table I
NUMBER OF SWEEPS FOR DIFFERENT JACOBI ALGORITHMS

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
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 II
NUMBER OF SWEEPS FOR JTS WITH DIFFERENT τ

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
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



(a) Number of actual rotations (b) Time taken (c) Scaling

Figure 1. Performance of real implementation of JTS

Lemma 5: Any sequence of arbitrary elements of length n can be sorted in $O(n)$ time using on a $n \times n$ mesh. $\square$

B. Implementation on the PRAM

For simplicity let the input matrix A have size $n \times n$. The details of the parallel implementation of each of the three phases of a single sweep in Algorithm 1 are given below.

Phase 1. For a single pair of columns, the dot product can be computed in $O(\log n)$ time using $\frac{n}{\log n}$ CREW PRAM processors. For all pairs, it takes $O(\log n)$ time using a total of $\frac{n^3}{\log n}$ CREW PRAM processors. Sorting the $\binom{n}{2}$ dot product values and selecting the top $\binom{n}{2}/\tau$ values takes $O(\log n)$ time using n^2 CREW PRAM processors.

Phase 2. For each pivot element we have to compute a rotation matrix as in Equation (2) which can be done in $O(1)$ time. Thus phase 2 takes $O(1)$ time using n^2 processors.

Phase 3. After having computed all the $\binom{n}{2}/\tau$ rotation matrices we have to multiply all of them. One possible way of doing this is to multiply each one of these matrices at a time with a cumulative product. Though the rotation matrix has only four nonzero entries, the cumulative product matrix can be possibly full. This multiplication can be done in $O(n)$ time sequentially. If we have a PRAM, this can be done in $O(1)$ time using n processors. Thus phase 3 can be completed in $O(n^2)$ time using n CREW PRAM processors. Alternatively, we can multiply all of these matrices like in a prefix computation. In this case each matrix multiplication can be between two possibly full matrices. Thus phase 3 can be completed in $O(\log^2 n)$ time using $\frac{n^5}{\log n}$ processors.

Combining all three phases, we get the following result:

Theorem 1: The JTS algorithm has a parallel run-time of $O(S \log^2 n)$ using $\frac{n^5}{\log n}$ CREW PRAM processors, and $O(Sn^2)$ using n CREW PRAM processors. Here S is the number of sweeps.

C. Implementation on the Mesh

We can run the JTS algorithm on the mesh as well. Phase 1 including sorting can be completed in $O(n)$ time on an $n \times n$ mesh. Phase 2 can be completed in $O(1)$ time, and phase 3 in $O(n^2)$ time. Thus, we get the following result.

Theorem 2: The JTS algorithm takes $O(Sn^2)$ time on an $n \times n$ mesh. Here S is the number of sweeps.

VI. Conclusions

In this paper, we have proposed and implemented a novel algorithm (called JTS) for computing SVDs in parallel. JTS improves the number of sweeps required by applying only the top $\frac{1}{\tau}$ fraction of possible Jacobi rotations in a sweep (for some suitable value of τ). We showed that our algorithm works better than known algorithms on both a simulated framework [8] and a real implementation based on the code in GSL [9]. We have also given theoretical results on the PRAM and the mesh. While experimenting on the real implementation we found that the cache misses play a big role as the selection of pivots and the application of rotations require computing dot-product of each column with all the others. In our future work, we would like to explore ideas from [5] and [22] for a cache efficient implementation.

Acknowledgment

This work has been supported in part by the NIH grant R01-LM010101 and NSF grant 1447711.

References

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

[2] J. Demmel and K. Veselić, "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," *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 *Proceedings of Euromicro Workshop on Parallel and Distributed Processing*. 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. Bečka, and M. Vajteršic, "Parallel One-Sided Block Jacobi SVD Algorithm with Dynamic Ordering and Variable Blocking: Performance Analysis and Optimization," Computer Science, University of Salzburg, Salzburg, Austria, Tech. Rep. 2016-02, March 2016.

[18] V. Strumpen, H. Hoffmann, and A. Agarwal, "A Stream Algorithm for the SVD," Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, Tech. Rep. 641, October 2003.

[19] J. JáJá, *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992, vol. 1.

[20] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*. Silicon Press, 2008, vol. 2.

[21] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Elsevier, 2014.

[22] M. I. Soliman, "Memory Hierarchy Exploration for Accelerating the Parallel Computation of SVDs," *Neural, Parallel and Scientific Computations*, vol. 16, no. 4, pp. 543–562, 2008.