A Novel Scheme for the Parallel Computation of SVDs

Sanguthevar Rajasekaran and Mingjun Song

Computer Science and Engineering
University of Connecticut
Storrs CT 06269, USA
{rajasek, mjsong}@engr.uconn.edu

Abstract. The Singular Value Decomposition (SVD) is a vital problem that finds a place in numerous application domains in science and engineering. As an example, SVDs are used in processing voluminous data sets. Many sequential and parallel algorithms have been proposed to compute SVDs. The best known sequential algorithms take cubic time. This amount of time may not be acceptable especially when the data size is large. Thus parallel algorithms are desirable. In this paper, we present a novel technique for the parallel computation of SVDs. This technique yields impressive speedups.

We discuss implementation of our technique on parallel models of computing such as the mesh and the PRAM. We also present an experimental evaluation of our technique.

Keywords: Singular Value Decomposition, Two-sided Jacobi, One-sided Jacobi.

1 Introduction

The Singular Value Decomposition (SVD) is a vital problem with applications in many a domain. An important application of SVD is to reduce dimensionality in data mining and information retrieval fields. The well-known sequential bidiagonalization-based Golub-Kahan-Reinsch SVD algorithm [6] takes $O(mn^2)$ time (on an $m \times n$ matrix). For large values of m and n, this time could be prohibitive. With the advent of the internet and the subsequent data explosion, parallel techniques for computing SVDs have become increasingly more important.

The bidiagonalization-based SVD algorithm has been found to be difficult to parallelize and hence works on parallel SVD focus on Jacobi-based techniques. Both two-sided Jacobi and one-sided Jacobi techniques have been studied in this context. Brent and Luk [5] presented a parallel one-sided SVD algorithm using a linear array of $O(n)$ processors, with a run time of $O(mnS)$, where S is the number of sweeps. They also presented an $O(nS)$ time algorithm to compute the singular values of a symmetric matrix using an array of n^2 processors. Zhou and

Brent [12] described an efficient parallel ring ordering algorithm for one-sided Jacobi.

Bečka and Vajteršic [2] presented a parallel block two-sided Jacobi algorithm on hypercubes and rings with a run time of $O(n^2S)$. They also gave an $O(nS)$ time algorithm on meshes [3]. Bečka *et al.* [1] proposed a dynamic ordering algorithm for a parallel two-sided block-Jacobi SVD with a run time of $O(nS)$. Okša and Vajteršic [10] designed a systolic two-sided block-Jacobi algorithm with a run time of $O(nS)$. Strumpen *et al.* [11] presented a stream algorithm for one-sided Jacobi that has a run time of $O(\frac{n^3}{p^2}S)$, where p is the number of processors (p being $O(\sqrt{n})$). They created parallelism by computing multiple Jacobi rotations independently and applying all the transformations thereafter. They show that each sweep of the Jacobi iteration algorithm can be parallelized on an $n \times n$ mesh in $O(nS)$ time. Clearly this algorithm is asymptotically optimal. Unfortunately their experimental results show that the value of S is much larger than what the sequential algorithm takes. In this paper, we employ their idea of separating rotation computations and transformations. We propose a novel algorithm for computing SVDs. This algorithm is fundamentally different from all the algorithms that have been proposed for SVD. It employs a specific "relaxation" of the Jacobi iteration. We call this *JRS iteration.* This algorithm is nicely parallelizable. For example, it enables the computation of all the rotations of a sweep in parallel such that the number of sweeps is reasonable.

We discuss the implementation of JRS iteration on various models of computing such as the mesh, the hypercube, and the PRAM. For example, on the CREW PRAM our algorithm has a run time of $O(S\log^2 n)$ (for a symmetrix $n \times n$ matrix).

The remainder of the paper is organized as follows. In Section 2, we introduce the sequential Jacobi-SVD algorithm. Section 3 describes our new JRS iteration algorithm. In Section 4, we show experimental results. Section 5 discusses parallel implementations of our new algorithms. Finally, we provide some concluding remarks in Section 6.

2 A Survey of the Basic Ideas

The SVD problem takes as input any $m \times n$ matrix A ($m \geq n$) and computes three matrices U, Σ, and V such that:

$$A = U\Sigma V^T,$$

where U is a $m \times n$ orthogonal matrix (i.e., $U^TU = I$), V is an $n \times n$ orthogonal matrix ($V^TV = I$), and Σ is an $n \times n$ diagonal matrix. If $\Sigma = diag(\sigma_1, \sigma_2, \ldots, \sigma_n)$, then these diagonal elements are the singular values of A. The column vectors of U are the left singular vectors of A, and the column vectors of V are the right singular vectors of A.

All the existing parallel algorithms use the Jacobi iteration as the basis. Jacobi iteration algorithm attempts to diagnolize the input matrix A by a series of *Jacobi* *rotations* where each rotation tries to zero-out an off-diagonal element. In particular, each Jacobi rotation involves premultiplying A by an orthogonal matrix and postmultiplying by another orthogonal matrix. We perform $(n^2 - n)/2$ rotations (in the case of a symmetric matrix) attempting to zero-out all the off-diagonal elements. These $(n^2 - n)/2$ transformations constitute a *sweep*. It can be shown that after each sweep the norm of the off-diagonal elements decreases and hence the algorithm converges. It is believed that the number S of sweeps needed for convergence of the sequential Jacobi iteration algorithm is $O(\log n)$ [6].

There are two varaints of the Jacobi iteration algorithm, namely, one-sided and two sided. Accordingly, there are two versions of our JRS iteration algorithm as well. Both the versions of JRS perform well in parallel.

2.1 Two-sided Jacobi SVD

The two-sided Jacobi iteration algorithm [9] transforms a symmetric matrix A into a diagonal matrix Σ by a sequence of Jacobi rotations (J):

$$\Sigma = \cdots J_3^T(J_2^T(J_1^T A J_1)J_2)J_3 \cdots = (J_1 J_2 J_3 \cdots)^T A (J_1 J_2 J_3 \cdots)$$

Each transform attempts to zero-out a given off-diagonal element of A. The Jacobi rotation, also called the Givens rotation[6], is defined as follows:

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

where (i,j) is an index pair of A to be zeroed, $c = \cos\theta$, $s = \sin\theta$, θ being called the rotation angle. It can be easily verified that $J^T J = I$, so the Jacobi rotation is an orthogonal transformation. The values of c and s are computed as follows. Consider one of the transformations: $B = J^T A J$. We choose c and s such that

$$\begin{pmatrix} b_{ii} & b_{ij} \\ b_{ji} & b_{jj} \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

is diagonal, *i.e.*, $b_{ij} = b_{ji} = 0$. By solving this equation and taking the smaller root [6], c and s are obtained by:

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

where

$$t = \frac{sign(\tau)}{|\tau| + \sqrt{\tau^2 + 1}},$$

and

$$\tau = \frac{a_{ii} - a_{jj}}{2a_{ij}}.$$

Depending on the order of choosing the element to be zeroed, there are classic Jacobi and cyclic Jacobi algorithms. In the classic Jacobi iteration algorithm, each transformation chooses the off-diagonal element of the largest absolute value. However, searching for this element requires expensive computations. Cyclic Jacobi algorithm sacrifices the convergence behavior and steps through all the off-diagonal elements in a row-by-row fashion. For example, if $n = 3$, the sequence of elements is $(1, 2), (1, 3), (2, 3), (1, 2), \ldots$. The computation is organized in sweeps such that in each sweep every off-diagonal element is zeroed once. Note that when an off-diagonal element is zeroed it may not continue to be zero when another off-diagonal element is zeroed. After each sweep, it can be shown that, the norm of the off-diagoal elements decreases monotonically. Thus the Jacobi algorithms converge.

2.2 One-sided Jacobi SVD

One-sided Jacobi algorithm, also called Hestenes-Jacobi algorithm [7][11], first produces a matrix B whose rows are orthogonal by premultiplying A with an orthogonal matrix U:

$$UA = B,$$

where rows of B satisfy:

$$b_i^T b_j = 0 \text{ for } i \neq j.$$

Followed by this B is normalized by:

$$V = S^{-1}B,$$

where $S = diag(s_1, s_2, \ldots, s_m)$, and $s_i = b_i^T b_i$. It can be easily shown that $A = U^T SV$, which is equivalent to the definition of SVD.

One-sided Jacobi is also realized by a series of Jacobi rotations, but on one side. For a given i and j, rows i and j are orthogonalized by $B = J^T A$ where $J = J(i, j, \theta)$ is the same matrix as in the two-sided Jacobi and:

$$\begin{pmatrix} b_i^T \\ b_j^T \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a_i^T \\ a_j^T \end{pmatrix}.$$

Here c and s of J are chosen such that $b_i^T b_j = 0$. The solution of them is:

$$c = \frac{1}{\sqrt{1+t^2}}, \quad s = tc.$$

where

$$t = \frac{sign(\tau)}{|\tau| + \sqrt{\tau^2 + 1}},$$

and

$$\tau = \frac{a_j^T a_j - a_i^T a_i}{2a_i^T a_j}.$$

As we could see, there is a close similarity between the one-sided and two-sided versions of the Jacobi algorithm.

3 JRS Iteration Algorithm

Since any rotation in the two-sided Jacobi algorithm changes only the corresponding (two) rows and (two) columns, and 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 $n-1$ non-conflicting rotation sets each of which containing $n/2$ rotations. For instance, if $n = 4$, there are 3 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 $O(n^2S)$ 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. Followed by this the second phase has to be completed. This involves the multiplication of $O(n^2)$ rotation matrices. Since two $n \times n$ matrices can be multiplied in $O(\log n)$ time using n^3 CREW PRAM processors (see e.g., [4], [8]), a straight forward 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 some 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. We let multiple processors compute the rotation matrices of a rotation set (one matrix per processor), 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 that wel call the *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:

$$off(A) = \sqrt{\sum_{i=1}^{n} \sum_{i=1, j \neq i}^{n} a_{ij}^2}.$$

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

$$b_{ij} = a_{ij}(c^2 - s^2) + (a_{ii} - a_{jj})cs = \lambda a_{ij},$$

where λ is in the interval $[0, 1)$. When $\lambda = 0$, we get 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

$$\frac{a_{ii} - a_{jj}}{2a_{ij}} = \frac{c}{2s} - \frac{s}{2c} - \frac{\lambda}{2cs}.$$

Let $\tau = \frac{a_{ii} - a_{jj}}{2a_{ij}}$, $t = \frac{s}{c}$, then

$$(1 + \lambda)t^2 + 2\tau t + \lambda - 1 = 0.$$

According to [6], the smaller root should be chosen, so

$$t = \frac{sign(\tau)(1 - \lambda)}{|\tau| + \sqrt{\tau^2 + (1 - \lambda^2)}}.$$

Like in the regular Jacobi rotation, c and s can be computed as:

$$c = \frac{1}{\sqrt{1 + t^2}}, \quad s = tc.$$

We call the above algorithm JRS iteration algorithm.

3.2 One-sided JRS Iteration Algorithm

One-sided JRS algorithm is similar to the one-sided Jacobi algorithm. In each rotation, we let the norm of the corresponding two rows be reduced to a fraction of it. That is,

$$v_i^T v_j = \lambda u_i^T u_j.$$

The solution is similar to the two-sided Jacobi:

$$c = \frac{1}{\sqrt{1+t^2}}, \quad s = tc.$$

where

$$t = \frac{sign(\tau)(1-\lambda)}{|\tau| + \sqrt{\tau^2 + (1-\lambda^2)}},$$

and

$$\tau = \frac{u_j^T u_j - u_i^T u_i}{2u_i^T u_j}.$$

4 Experimental Results

We have implemented our JRS algorithms and tested them for convergence as well as performance. They have been compared with the regular Jacobi algorithms as well as the algorithms of [11]. We provide the experimental results in this subsection.

In this experiment, we have compared the number of sweeps taken by the different Jacobi approaches. We generated randomly several matrices of different sizes, including $10 \times 10, 50 \times 50, 100 \times 100, 200 \times 200, 500 \times 500$, and 1000×1000. The elements of the matrices were generated randomly to have a value in the interval $[1, 10]$. For each matrix size, we generated 10 matrices and for each algorithm we took the average number of sweeps. The convergence condition employed was on the norm of the off-diagonal elements. We used a norm value of 10^{-15}.

The results are shown in tables 1 and 2 for two-sided Jacobi and one-sided Jacobi algorithms, respectively. For two-sided Jacobi, we used symmetric matrices; for one-sided Jacobi, we generated unsymmetric matrices. In these tables, Independent Jacobi refers to the Jacobi algorithm where all the rotations in a sweep are done independently and in parallel. This is one of the algorithms employed in [11]. The values of the parameter λ used in JRS algorithm for matrices of different sizes are chosen experimentally, which are: 0.25, 0.5, 0.5, 0.75, 0.8, 0.85 respectively.

From tables 1 and 2, we see that the number of sweeps taken by the JRS is significantly less than that of Independent Jacobi of [11]. Also the number of sweeps taken by the JRS based algorithm is within a reasonable multiple of that of the sequential cyclic Jacobi algorithm.

Table 1. Experimental results for two-sided SVD

Matrix size	Cyclic Jacobi	Independent Jacobi [11]	JRS
10×10	5	11	16
50×50	7	6692	31
100×100	7	>300000	47
200×200	8	>300000	66
500×500	9	>300000	90
1000×1000	9	>300000	125

Table 2. Experimental results for one-sided SVD

Matrix size	Cyclic Jacobi	Independent Jacobi [11]	JRS
10×10	7	14	28
50×50	10	133	52
100×100	11	1037	62
200×200	12	193107	114
500×500	14	>300000	145
1000×1000	15	>300000	197

5 Parallelism

As our experimental results show, even when all the rotations in a sweep are done in parallel, JRS based algorithms converge fast. In particular, the number of sweeps is no more than a reasonable multiple of the number of sweeps taken by the sequential Jacobi algorithm. As a consequence, JRS based algorithms offer maximum parallelism. In fact most of the parallel algorithms that have been derived thus far (that employ Jacobi iterations) for SVDs can be readily translated into JRS based algorithms. We just mention a few of them below.

Based on the algorithms of [11] we get:

Theorem 1. *JRS algorithms run in time $O(nS)$ on an $n \times n$ mesh.*

The algorithm of [5] yields the following:

Theorem 2. *One-sided JRS algorithm can be implemented on a linear array of $O(n)$ processors to have a run time of $O(mnS)$.*

From our discussion in Section 3, we infer the following:

Theorem 3. *JRS algorithms can be implemented on a CREW PRAM to have a run time of $O(S \log^2 n)$.*

6 Conclusions

In this paper, we have proposed a novel algorithm (called JRS Iteration Algorithm) for computing SVDs. This algorithm enables one to perform all the rotations in a sweep independently and in parallel without increasing the number of sweeps significantly. 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 the full version of this paper we provide additional experimental data, a value for λ (as a function of n) that results in the minimum number of sweeps, a convergence proof for JRS, a variant of JRS called *Group JRS*, etc.

Acknowledgments. This research has been supported in part by the NSF Grant ITR-0326155.

References

1. Bečka, M., Okša, G., Vajteršic, M., Dynamic ordering for a parallel block-Jacobi SVD algorithm, *Parallel Comp.*, 28, 243-262, 2002.
2. Bečka, M. and Vajteršic, M., Block-Jacobi SVD algorithms for distributed memory systems I: Hypercubes and rings, *Parallel Algorithms Appl.*, 13, 265-287, 1999.
3. Bečka, M. and Vajteršic, M., Block-Jacobi SVD algorithms for distributed memory systems II: Meshes, *Parallel Algorithms Appl.*, 14, 37-56, 1999.
4. Bini, D., and Pan, V., Polynomial and Matrix Computations, Vol.1, Fundamental Algorithms, Birkhäuser, Boston, 1994.
5. Brent, R.P., and Luk, F.T., The Solution of Singular-Value and Symmetric Eigenvalue Problems on Multiprocessor Arrays. *SIAM Journal on Scientific and Statistical Computing*, 6(1):69-84, 1985.
6. Golub, G.H., and Van Loan, C.F., *Matrix Computations.* John Hopkins Univer sity Press, Baltimore and London, 2nd edition, 1993.
7. Hestenes, M.R., Inversion of Matrices by Biorthogonalization and Related Results. *Journal of the Society for Industrial and Applied Mathematics*, 6(1):51-90, 1958.
8. Horowitz, E., Sahni, S., and Rajasekaran, S., *Computer Algorithms*, W.H. Freeman Press, 1998.
9. Jacobi, C.G.J., Über eine neue Auflösungsart der bei der Methode der kleinsten Quadrate vorkommenden linearen Gleichungen. *Astronomische Nachrichten*, 22, 1845.
10. Okša, G., and Vajteršic, M., A Systolic Block-Jacobi SVD Solver for Processor Meshes, *Parallel Algorithms and Applications*, 18(1-2), 49-70, 2003.
11. Strumpen, V., Hoffmann, H., Agarwal, A., A Stream Algorithm for the SVD, *Technical Memo 641*, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, October 2003.
12. Zhou, B.B., and Brent, R.P., A Parallel Ring Ordering Algorithm for Efficient One-sided SVD Computations, *Journal of Parallel and Distributed Computing*, 42, 1-10, 1997.