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 factorizations such as Cholesky, LU and QR. In the final stage the band matrix is converted to a diagonal form to obtain the singular values. The singular vectors are also computed accordingly. In the sequential setting, the QR based methods are more frequently used as they are faster than the sequential Jacobi based methods. However, Jacobi-based methods are known to be more accurate [2] and also have a higher degree of potential parallelism. Though there are parallel implementations of QR based algorithms in out-of-core setting [3], [4] as well as in homogeneous multi-core setting [5], our focus is on designing faster parallel implementations of the Jacobi based algorithms.

There are two different variations of the Jacobi based algorithms, *one-sided* and *two-sided*. The two sided Jacobi algorithms are applicable only when the input matrix is symmetric and are also computationally more intensive. Hence the one-sided Jacobi algorithms are used more frequently. The basic idea of the one-sided Jacobi algorithms proposed by Hestenes [6] is as follows. Using a series of plane rotation matrices $V_1, V_2, \ldots, V_k$ the input matrix A is first converted to a matrix $B = AV_1V_2\ldots V_k$ 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 $V = V_1V_2, \ldots, V_k$. 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 matrix 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. The rotation using pivot (j,k) ensures that the columns j,k become mutually orthogonal after the rotation. However, the columns may again become non-orthogonal due to some later pivot involving one of the columns j,k in the same sweep. Hence multiple sweeps are required. As a test 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 Jacobi algorithm is $O(\log n)$ [1].

It turns out that order in which the pivots are applied in a sweep has significant effect on the number of sweeps used. Researchers have have used mainly two different orders. In the classic Jacobi algorithm [1], each rotation chooses the pivot (j,k) with maximum absolute value of $a_j^T a_k$. However, searching for this element is computationally expensive. Cyclic Jacobi algorithms trade off this computation with lesser convergence rate 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 the sum of dot-products of all pair of columns reduces and hence the algorithm converges.

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-diagonal elements decreases monotonically. Thus the Jacobi algorithms converge

A. Contributions

In this paper we introduce a novel algorithm for parallel computation of SVDs. This algorithm is a specific relaxation of the Jacobi iteration. We call this JRS (Jacobi Relaxation Scheme) iteration hereafter. This algorithm is nicely parallelizable, since it enables the computation of all the rotations of a sweep in parallel and the number of sweeps is reasonable. For the example of a 100100 matrix, JRS iteration takes 47 sweeps. A variant of JRS takes only 12 sweeps. 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 an n 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

II. Previous Work

[7]–[9] introduced and implemented the idea of parallel one-sided block jacobi algorithm with dynamic ordering and variable blocking. Their approach, known as OSBJ method, accomplishes the parallel SVD in three stages namely, pre-processing, iteration and post processing. There are two types of pre-processing depending on the dimension of the matrix, namely QR-pre-processing and LQ pre-processing. During QR-pre-processing OSBJ decomposes input matrix $A = Q_1 R$ where $A \in \mathbb{R}^{m\times n}$, orthonormal matrix $Q_1 \in \mathbb{R}^{m\times m}$ and upper triangular matrix $R \in \mathbb{R}^{n\times n}$. On the contrary, LQ-pre-processing decomposes input matrix $A = LQ_2$ where $L \in \mathbb{R}^{n\times n}$ is a lower triangular matrix and $Q_2 \in \mathbb{R}^{\Bbbk\times\Bbbk}$ is an orthogonal matrix. The L or R matrix in pre-processing stage is considered as input A^0 for the iteration stage. A^0 is partitioned into column blocks. Let, $A_i^{(r)}$ and $A_j^{(r)}$ are one such column block pair where $1 \leq i < j \leq l$ assuming we have l such column blocks. Weight $\hat{w}_{ij}^r = \frac{||A_i^{(r)T} A_j^{(r)}\mathbf{e}||_2}{||\mathbf{e}||_2}$, where $\mathbf{e} = (1,1,....1)^T \in \mathbb{R}^{\frac{n}{l}}$. $\hat{w}_{ij}^r$ serves as an approximate measure of inclination between $Im(A_i^r)$ and $Im(A_j^r)$. The column block pairs are ordered based on their mutual inclination and the most mutually inclined pair is chosen to be orthogonalized and those columns are excluded from the set for next iteration. The process is repeated until all the column blocks are used up. For each column block pair a 2×2 Gram matrix $G_s \in \mathbb{R}^{\frac{2n}{l}\times\frac{2n}{l}}$ is constructed as is given by $G_s = [A_{i_{r,s}}^{(r)}, A_{j_{r,s}}^{(r)}]^T [A_{i_{r,s}}^{(r)}, A_{j_{r,s}}^{(r)}]$ where $1 \leq s \leq \frac{l}{2}$. This gram matrix is them diagonalized as $G_s = V_s D_s V_s^T$ where V_s is an orthogonal matrix and D_s is a diagonal matrix. The column blocks for next iteration $A^{(r+1)}$ is computed by $[A_{i_{r,s}}^{(r+1)}, A_{j_{r,s}}^{(r+1)}] = [A_{i_{r,s}}^{(r)}, A_{j_{r,s}}^{(r)}]V_s$. Iteration process continues until all the columns are mutually orthogonal.

The Jacobi Relaxation Scheme (JRS) algorithm by Rajasekaran and Song introduced the idea of improving parallelism in SVD computation by multiplying the off-diagonal element in each iteration by a very small number ϵ such that $0 < \epsilon < 1$ instead of setting the off-diagonal element to zero. [10] compares number of iterations taken by JRS algorithm to converge with that of Strumpen's Independent Jacobi algorithm [**?**] and shows that JRS takes much less number of iterations to converge than Independent Jacobi.

This paper has done same parallel implementation: [11].

This paper seems to do some kind of sorting for [12].

III. Quick Pivoting

Since any rotation in the two-sided Jacobi algorithm changes only the corresponding (two) rows and (two) columns, and 771 one-sided Jacobi algorithm changes only the corresponding (two) rows, there exists inherent parallelism in the Jacobi iteration algorithms. For example, the n(n 1)/2 rotations in any sweep can be grouped into n 1 rotation sets each of which contains n/2 independent rotations. For instance, if n = 4, there are three rotation sets: (1,2),(3,4), (1,3),(2,4), (1,4),(2,3). Since each rotation can be performed in O(n) time on a single machine, we can perform all the rotations in O(n 2 S) time on a ring of n processors [6]. The idea here is to perform each set of rotations in parallel. We can think of the Jacobi algorithm as consisting of two phases. In the first phase we compute all the rotation matrices (there are O(n 2) of them). In the second phase we multiply them out to get U and V . Consider any rotation operation. The values of s and c can be computed in O(1) time sequentially. The algorithm of Strumpen et al. [11] performs all the n(n 1)/2 rotations of a sweep in parallel even though not all of these rotations are independent. Thus in their algorithm, all the rotation matrices can be constructed in O(1) time using n 2 CREW PRAM processors. This will complete the first phase of the Jacobi algorithm. The second phase has to be completed. This involves the multiplication of O(n 2) rotation matrices. Since two nn matrices can be multiplied in O(log n) time using n 3 CREW PRAM processors (see e.g. [4,8]), a straightforward implementation of [11]s algorithm runs in time O(S log 2 n) using n 5 CREW PRAM processors. In [11] an implementation on an n n mesh has been given that runs in O(nS) time. However, as has been pointed out before, the value of S is much larger than the corresponding value for the sequential Jacobi iteration algorithm. Any parallel algorithm for SVD partitions the n(n 1)/2 rotations of a sweep into rotation sets where each rotation set consists of a number of rotations. All the rotations of a rotation set are performed in parallel. Most of the parallel SVD algorithms in the literature employ (n 1) rotation sets each rotation set consisting of n/2 independent rotations. The algorithm of Strumpen et al. is an exception, where multiple processors compute the rotation matrices independently, all the processors employing the same original matrix. In the sequential case, if A is the input matrix, computations will proceed as follows: B 1 = J 1 T A J 1 ; B 2 = J 2 T B 1 J 2 ; B 3 = J 3 T B 2 J 3 ; and so on. On the other hand, in parallel, computations will proceed as follows: B 1 = J 1 T A J 1 ; B 2 = J 2 T A J 2 ; B 3 = J 3 T A J 3 ; etc. The number of B i s computed in parallel will be decided by the number of available processors. Once this parallel computation is complete, all of the computed transformations will be applied to A to obtain a new matrix B. After this, again a parallel computation of rotation matrices will be done all with respect to B; B will be updated with the computed transformations; and so on. In this paper we propose a fundamentally different algorithm for SVD. It is a specific relaxation of the Jacobi iteration algorithm called JRS iteration algorithm. Just like the Jacobi algorithm, there are two variants of the JRS iteration algorithm as well, namely, one-sided and two-sided. We provide details on these two variants in the next subsections.

IV. Parallel Implementation

Wherever Times is specified, Times Roman or Times New Roman may be used. If neither is available on your system, please use the font closest in appearance to Times. Avoid using bit-mapped fonts if possible. True-Type 1 or Open Type fonts are preferred. Please embed symbol fonts, as well, for math, etc.

V. Experimental Results

We implemented our algorithms and tested them for convergence. We had two different implementations. Using the strategy in [10] 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) [13].

A. Simulation on a sequential computer

We added a simulation of our algorithm in the list of algorithms simulated in [10] and compared the performances in terms of the number of sweeps. For the two-sided algorithms, we generated random symmetric matrices for different values of n: $500, 750, 1000, 1250, 1500, 1750, 2000$. For the one-sided algorithms we generated random matrices of sizes $500 \times 300, 750 \times 500, 1000 \times 700, 1250 \times 900, 1500 \times 1300, 1750 \times 1500, 2000 \times 1800$. The elements of the matrices were chosen uniformly randomly from the range $[1, 10]$. For each matrix sizes and the algorithms that we considered, we generated 5 random matrices and reported the average number of sweeps used by the algorithms. The convergence condition employed was 10^{-15} times the squared Frobenius norm of the input matrix.

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

For the JRS algorithms we used the value of the relxation parameters as recommended in [10], i.e., for two-sided Jacobi $\lambda = 1 - 2.9267n^{-0.4284}$ and for one-sided Jacobi $\lambda = 1 - 2.2919n^{-0.3382}$. For the 'group' variant of the parallel algorithms, we use the value of $g = (n-1)/\sqrt{n}$.

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

B. Experimental recommendation for top-frac parameter k

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

C. A real implementation on a multicore computer

We also implemented a 'real' version of our algorithms. We changed the basic implementation of one-sided Jacobi in GSL [13] to accommodate our ideas. Note that the GSL implementation is based on the algorithm given in [14].

VI. Conclusion

In this paper, we have proposed a novel algorithm (called JRS Iteration Algorithm) for computing SVDs. This algorithm enables us to perform all the rotations in a sweep independently and in parallel without increasing the number of sweeps 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 this paper we have also provided expressions for the relaxation parameter that will result in the minimum number of sweeps.

References

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

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

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

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

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

[8] 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.

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

[10] 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.

[11] 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.

[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. 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.

[14] 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.

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

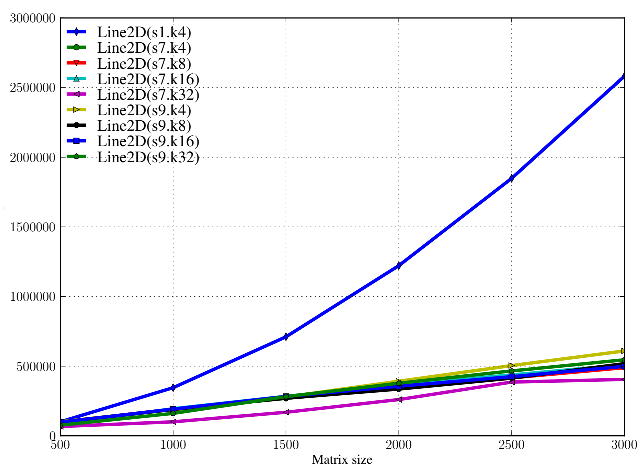
Matrix size	Cyclic	TopKFrac	TopKFracJRS	ParJRS	GrpJRS	ParTopKFrac	ParTopKFracJRS	GrpTopKFrac	GrpTopKFracJRS
500×500									
750×750									
1000×1000									
1250×1250									
1500×1500									
1750×1750									
2000×2000									

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

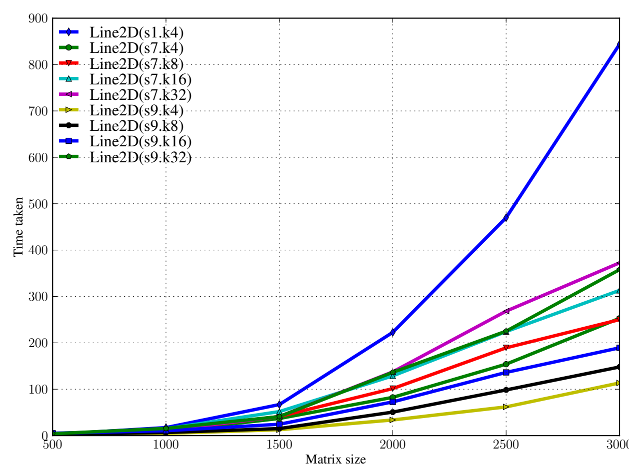
Matrix size	Cyclic	TopKFrac	TopKFracJRS	ParJRS	GrpJRS	ParTopKFrac	ParTopKFracJRS	GrpTopKFrac	GrpTopKFracJRS
500×200									
750×500									
1000×700									
1250×900									
1500×1300									
1750×1500									
2000×1800									

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

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



(a) Number of updates



(b) Time taken

Figure 1. Performance of our parallel implementation