On Parallel Implementation of the One-sided Jacobi Algorithm for Singular Value Decompositions

B. B. Zhou and R. P. Brent
Computer Sciences Laboratory
The Australian National University
Canberra, ACT 0200, Australia
{bing,rpb}@cslab.anu.edu.au

Abstract

In this paper we give evidence to show that in one-sided Jacobi SVD computation the sorting of column norms in each sweep is very important. Two parallel Jacobi orderings are described. These orderings can generate $n(n-1)/2$ different index pairs and sort column norms at the same time. The one-sided Jacobi SVD algorithm using these parallel orderings converges in about the same number of sweeps as the sequential cyclic Jacobi algorithm. Some experimental results on a Fujitsu AP1000 are presented. The issue of equivalence of orderings is also discussed.

1 Introduction

Let A be a real $m \times n$ matrix. Without loss of generality we assume that $m \geq n$. The singular value decomposition (SVD) of A is its factorization into a product of three matrices

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

where U is an $m \times n$ matrix with orthonormal columns, V is an $n \times n$ orthogonal matrix, and Σ is an $n \times n$ non-negative diagonal matrix, say $\Sigma = diag(\sigma_1, \cdots, \sigma_n)$.

There are various ways to compute the SVD [11]. Two of the most commonly used classes of algorithms are *QR-based* and *Jacobi-based*. In sequential computing the QR-based algorithms are usually preferred because they are faster than the Jacobi-based algorithms. However, the Jacobi-based algorithms may be more accurate [6]. The Jacobi-based algorithms have recently attracted a lot of attention as they have a higher degree of potential parallelism. There are two varieties of Jacobi-based algorithms, *one-sided* and *two-sided*. The two-sided Jacobi algorithms are computationally more expensive than the one-sided algorithms, and not so suitable for vector pipeline computing. Thus, to achieve efficient parallel SVD computation the best approach may be to adopt the Hestenes one-sided transformation method [13] as advocated in [4, 5].

The Hestenes method generates an orthogonal matrix V such that

$$AV = H,$$

where the columns of H are orthogonal. The nonzero columns $\tilde{H}$ of H are then normalised so that

$$\tilde{H} = U_r\Sigma_r$$

with $U_r^T U_r = I_r$, $\Sigma_r = diag(\sigma_1, \cdots, \sigma_r)$ and $r \leq n$ is the rank of A.

The matrix V can be generated as a product of plane rotations. As in the traditional Jacobi algorithm, the rotations are performed in a fixed sequence called a *sweep*, each sweep consisting of $n(n-1)/2$ rotations, and every column in the matrix is orthogonalised with every other column exactly once per sweep. The iterative procedure terminates if one complete sweep occurs in which all columns are orthogonal to working accuracy and no columns are interchanged. If the rotations in a sweep are chosen in a reasonable, systematic order, the convergence rate is ultimately quadratic [9, 11]. Exceptional cases in which cycling occurs are easily avoided by the use of a threshold strategy [23].

It is known that one Jacobi plane rotation operation only involves two columns. Therefore, there are disjoint operations which can be executed simultaneously. In a parallel implementation, we want to perform as many non-interacting operations as possible at each parallel time step. Many parallel orderings have been introduced in the literature [3, 4, 5, 7, 8, 10, 14, 16, 17, 18]. These orderings were mainly designed for parallel eigenvalue decompositions. Special

1066-6192/95 $4.00 © 1995 IEEE

care has to be taken in order to achieve high efficiency for parallel SVD computations.

In this paper we show that sorting the column norms in each sweep of the SVD computation is a very important issue. If a parallel ordering does not include a proper sorting procedure, it results in too many sweeps when adopted in a one-sided Jacobi SVD algorithm. We introduce two parallel Jacobi orderings and show that both orderings can generate $n(n-1)/2$ different index pairs, and also sort n elements into order, where n is the problem size. The first ordering is a novel ring ordering, while the second is a combination of the well-known odd-even index ordering and odd-even transposition sort. We have implemented one-sided Jacobi SVD using each ordering on a distributed memory MIMD machine, the Fujitsu AP1000. Our experimental results show that the new algorithms can achieve the same efficiency as the sequential cyclic Jacobi algorithm for SVDs, i.e. the same total number of sweeps to convergence.

The paper is organised as follows. §2 discusses sequential algorithms. Three different rotation algorithms for generating plane rotation parameters are described. In §3 we consider parallel implementation of one-sided Jacobi SVD using index sorting and show the efficiency obtained (in terms of the total number of sweeps) is not as good as that of the sequential cyclic Jacobi algorithm. Our parallel Jacobi orderings (with experimental results) are then described in §4 and §5. The issue on equivalence of orderings for one-sided Jacobi is also discussed in §6. Some conclusions are given in §7.

2 Sequential Algorithms

There are two important implementation details which determine the speed of convergence of the one-sided Jacobi method for computing the SVD. The first is the method of ordering, i.e., how to order the $n(n-1)/2$ rotations in one sweep of computation. Various orderings have been introduced in the literature. In sequential computation, the most commonly used is the cyclic Jacobi ordering (cyclic ordering by rows or by columns) [9, 12]. When discussing sequential Jacobi algorithms in this paper, we assume that the cyclic ordering by rows is applied.

The second important detail is the method for generating the plane rotation parameters c and s in each iteration. For the one-sided Jacobi method there are three main rotation algorithms, which we now describe.

Rotation Algorithm 1 This algorithm is derived from the standard two-sided Jacobi method for the eigenvalue decomposition of the matrix $B = A^T A$.

Suppose that after k sweeps we have the updated matrix $A^{(k)} = \left[\begin{array}{cccc} a_1^{(k)} & a_2^{(k)} & \cdots & a_n^{(k)} \end{array} \right]$. To annihilate the off-diagonal element $b_{ij}^{(k)}$ of $B^{(k)} = (A^{(k)})^T A^{(k)}$ in the $(k+1)^{th}$ sweep, we first need to compute $b_{ii}^{(k)}$, $b_{ij}^{(k)}$ and $b_{jj}^{(k)}$, that is,

$$b_{ii}^{(k)} = (a_i^{(k)})^T a_i^{(k)} = \|a_i^{(k)}\|^2,$$

$$b_{ij}^{(k)} = (a_i^{(k)})^T a_j^{(k)}$$

and

$$b_{jj}^{(k)} = (a_j^{(k)})^T a_j^{(k)} = \|a_j^{(k)}\|^2.$$

where $\|x\|$ is the 2-norm of the vector x. The plane rotation factors c and s, which are used to orthogonalise the corresponding two columns, are then generated based on the two-sided Jacobi method. It can be proved that the value of $b_{ii}^{(k)}$ is increased and the value of $b_{jj}^{(k)}$ is decreased after a plane rotation operation if $b_{ii}^{(k)} > b_{jj}^{(k)}$. Otherwise, $b_{ii}^{(k)}$ is decreased and $b_{jj}^{(k)}$ is increased.

Rotation Algorithm 2 The second algorithm, introduced by Hestenes [13], is the same as the Algorithm 1 except that the columns $a_i^{(k)}$ and $a_j^{(k)}$ are to be swapped if $\|a_i^{(k)}\| < \|a_j^{(k)}\|$ for $i < j$ before the orthogonalization of the two columns. Therefore, we always have $b_{ii}^{(k+1)} \geq b_{jj}^{(k+1)}$. When the cyclic ordering by rows is applied, the computed singular values will be sorted in a nonincreasing order.

Rotation Algorithm 3 The third algorithm was derived by Nash [19] and implemented on the ILLIAC IV by Luk [16]. To determine the rotation parameters c and s for orthogonalising two columns i and j, one extra condition has to be satisfied in this algorithm, that is,

$$\|a_i^{(k+1)}\|^2 - \|a_i^{(k)}\|^2 = \|a_j^{(k)}\|^2 - \|a_j^{(k+1)}\|^2 \geq 0.$$

With this extra condition the rotation parameters are chosen so that $\|a_i^{(k+1)}\|$ is greater than $\|a_j^{(k+1)}\|$ after the orthogonalization, without explicitly exchanging the two columns. As in Algorithm 2, the computed singular values will appear in a nonincreasing order if the cyclic ordering by rows is applied.

It is known from numerical experiments that an implementation which uses Rotation Algorithm 2 or 3 is

Size	Alg. 1	Alg. 2	Alg. 3
80	11	9	9
100	12	8	8
120	11	9	9
140	12	9	9
160	12	9	9
180	12	9	9
200	12	9	10

Table 1: Sweeps to convergence for the cyclic Jacobi ordering on a Sun workstation.

more efficient than the one using Rotation Algorithm 1 when the cyclic ordering is applied.

It is easy to verify that implicit in the cyclic ordering is a sorting procedure which can sort the values of n elements into nonincreasing (or nondecreasing) order in $n(n-1)/2$ steps. Since Rotation Algorithms 2 and 3 always increase $b_{ii}^{(k)}$ and decrease $b_{jj}^{(k)}$ for $i < j$ when orthogonalising the two columns, the column norms tend to be sorted after each sweep of computations. Therefore, the columns and their norms tend to be approximately determined after a few sweeps and only change by a small amount during each sweep. Since the column norms are not sorted during each sweep when using Rotation Algorithm 1, it is possible that the norm of column i may be increased when two columns i and j are orthogonalised in a sweep, but norm of column j will be increased when the two columns meet again in the next sweep. Thus there are oscillations in column norms and (empirically) it takes more sweeps for the same problem to converge. This effect was also noted in [6, 20]. It is probably the main reason why applying Rotation Algorithm 2 or 3 is more efficient than applying Rotation Algorithm 1.

In order to compare the performance in terms of the total number of sweeps with parallel implementations which are described in the following sections, we give in Table 1 some experimental results obtained on a (sequential) Sun Sparc workstation.

3 A Parallel Implementation Using Index Sorting

Many parallel Jacobi orderings have been introduced in the literature. Any of these orderings may be adopted to implement the one-sided Jacobi method in parallel. However, care has to be taken in order to achieve the desired efficiency. In this section we give

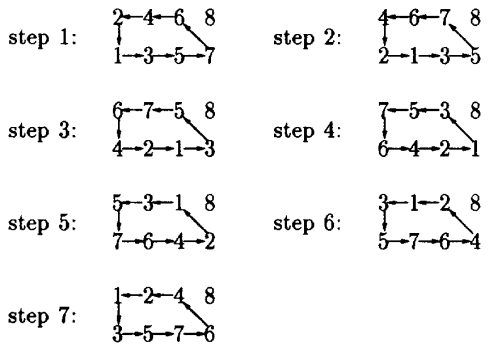


Figure 1: The round robin ordering

Size	Algorithm 1		Algorithm 2		Algorithm 3	
	T	S	T	S	T	S
200	14.43	12	19.31	16	19.11	16
400	72.74	13	97.59	17	94.45	16
600	226.4	14	290.4	17	291.6	17
800	519.6	15	645.5	17	641.9	17
1000	994.4	15	1255	18	1228	17
1200	1811	16	2219	18	2207	18
1400	2978	18	3495	18	3439	18

Table 2: Results for the round robin ordering, with index sorting, on an AP1000 with 100 processors organised as a linear array (T = time (sec.), S = sweeps).

some results from our experiments and show that optimal efficiency may not be obtained without a proper procedure for sorting the column norms in each sweep. For simplicity we assume from now on that n (the number of columns of A) is even.

In our experiment the well-known round robin ordering is applied. This ordering is depicted in Fig. 1. When Rotation Algorithm 2 or 3 described in the §2 is applied, we always increase the norm of a column associated with smaller index when orthogonalising a pair of columns. Since the indices are placed in a nondecreasing order, the final results should be in a nonincreasing order.

In the experiment on a Fujitsu AP1000 both singular values and singular vectors are calculated using 100 PEs which are organised as a one-dimensional array. The results are given in Table 2. It can be seen from Table 2 that the results from using Rotation Algorithm 2 or 3 are not as good as those from using

Rotation Algorithm 1 in terms of both time and the number of sweeps. This is inconsistent with the conclusion obtained in sequential computation using the cyclic ordering by rows. The main reason, which will become more clear in the following sections, is that the round robin ordering cannot sort the elements properly in a nonincreasing (or nondecreasing) order in one sweep (or $n-1$ steps).

Although the natural order of the indices is restored after each sweep, large norms will not necessarily go with small indices when rotation algorithm 2 is applied. The exchange of columns is then inevitable. Consequently certain pairs of columns may not be generated in the sweep. This degrades the overall performance. When Rotation Algorithm 3 is adopted, there is no need to swap columns. Since we always increase the norm of the column associated with the smaller index in orthogonalising a pair of columns, the values of some large norms (which should be increased) will be decreased in the next sweep because the corresponding columns are placed in the positions associated with the larger indices in the index pairs. As a consequence, more sweeps are required for computing a given problem.

In the next two sections we introduce two Jacobi ordering and show that more efficient results may be obtained if a proper sorting of columns norms in each sweep is considered.

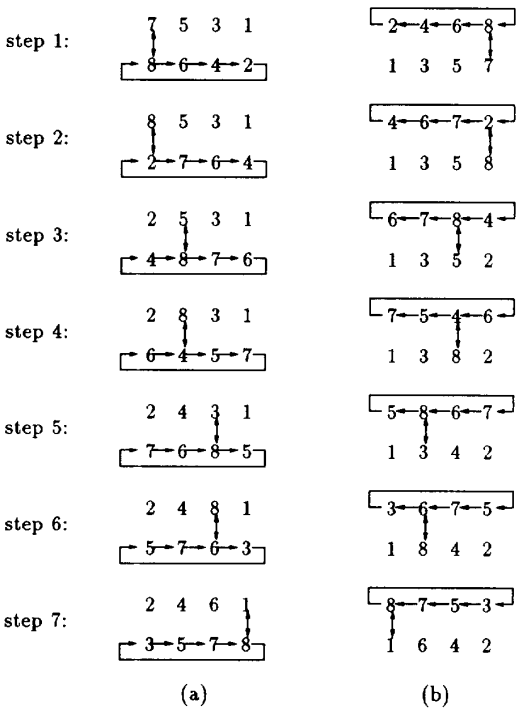


Figure 2: The ring Jacobi ordering. (a) forward sweep and (b) backward sweep.

4 The Ring Jacobi Ordering

In this section we describe a parallel ring ordering. This ordering can not only generate the required index pairs in a minimum number of steps, but also sort column norms at the same time.

Our Jacobi ordering consists of two procedures, *forward sweep* and *backward sweep*, as illustrated in Fig. 2. They are applied alternately during the computation.

In either forward or backward sweep the n indices are organised into two rows. Any two indices in the same column at a step form one index pair. One index in each column is then shifted to another column as shown by the arrows so that different index pairs can be generated at the next step. The up-and-down arrow in Fig. 2 indicates the exchange of two indices in the column before one is shifted. Each sweep (forward or backward), taking $n-1$ steps, can generate $n(n-1)/2$ different Jacobi pairs, as well as sort the values of n elements into nonincreasing (or nondecreasing) order.

We outline a proof that $n(n-1)/2$ different Jacobi pairs can be generated in $n-1$ steps by either a forward or backward sweep. To do this, we first permute the initial positions of n indices for the *round robin* ordering [5] and then show that the orderings can generate the same index pairs at any step. Since it is well known that the round robin ordering generates $n(n-1)/2$ different index pairs in $n-1$ steps, this shows the correctness of our claim. The detailed proof is tedious and is omitted.

It can easily be verified that the forward sweep and the backward sweep are essentially the same, except that one sorts the elements into nondecreasing order and the other sorts the elements into nonincreasing order. Thus we only use the forward sweep as an example to show the procedure on how to sort n elements into nondecreasing order. (For detailed proof see [24].)

If the numbers in Fig. 2(a) are not considered as indices, but as the values of n elements, the figure gives an example of sorting n elements from nonincreasing order to nondecreasing order. In each step the smaller element in each column is placed on the top except in even steps the larger element is placed on the top if

the column has a up-and-down arrow in it. Since the up-and-down arrow indicates the exchange of the two elements in the column, these arrows can be removed in even steps by letting the smaller elements be placed at the top of the corresponding columns. Thus, we may describe the sorting procedure as follows:

One forward sweep can be applied to sort n elements in a nondecreasing order. Each step in the sweep consists of two substeps. The first substep compares the two elements in each column and places the smaller one on the top and the larger one at the bottom. The second substep then shifts the elements located at the bottom to the next column according to the arrows which form a ring, as depicted in Fig. 2(a). At each odd step the two elements in the column with a up-and-down arrow have to exchange their positions before the shift takes place. For n elements they are sorted in a nondecreasing order after $n-1$ such steps.

Since both index ordering and sorting can be done simultaneously in either a forward or a backward sweep, it may seem that applying these two sweeps alternately in the SVD computation is not necessary. The reason why we perform the two sweeps alternately is as follows. Suppose that the n indices are initially placed in a nonincreasing order. They will be sorted in a nondecreasing order after a forward sweep. However, the natural order of indices for index ordering at each step is maintained during sorting. Thus the $n(n-1)/2$ different index pairs are also generated after the computation. Although the original (nonincreasing) order is restored when a backward sweep is performed, the exchange of positions of some indices is inevitable. As a consequence some index pairs may not be produced after the computation. This can easily be verified by an example of sorting a small number of indices (which are initially placed in a nonincreasing order) using the backward sweep.

We implemented our ring ordering algorithm on the Fujitsu AP1000 at the Australian National University. In the experiment the system is configured as a one-dimensional array and both singular values and singular vectors are computed. Some of the experimental results from applying different rotation algorithms are given in Table 3. It is easy to see from the table that the program adopting rotation algorithm 1 is not as efficient as those adopting rotation algorithms 2 or 3, especially when the problem size is large. If the total number of sweeps is counted, these results are consistent with those in Table 1 (obtained in sequential computation using the cyclic ordering by rows). In our experiment we also measured the sensitivity of the performance to the different number of processors used in the computation. The results show that the total number of sweeps required for the computation of the same SVD will not vary as the processor number is changed. Our experimental results are clear evidence which shows how important it is to adopt a proper sorting procedure in each sweep.

Size	Algorithm 1		Algorithm 2		Algorithm 3	
	T	S	T	S	T	S
200	11.58	12	10.01	10	10.02	10
400	63.35	13	57.39	11	57.42	11
600	210.8	14	187.1	12	185.6	12
800	499.5	15	416.7	12	416.3	12
1000	945.2	15	799.1	12	799.8	12
1200	1702	16	1407	12	1445	13
1400	2787	17	2280	13	2272	13

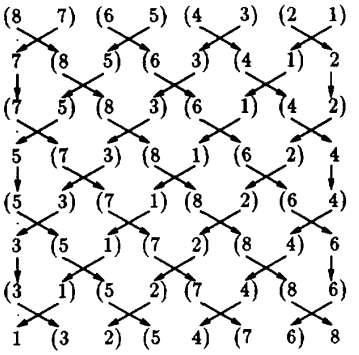
Table 3: Results for the ring Jacobi ordering on an AP1000 with 100 processors organised as a linear array (T = time (sec.), S = sweeps).

5 The Odd-Even Ordering

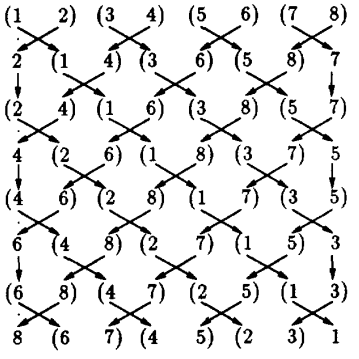
It is known that the odd-even index ordering [17] has the same data flow pattern as the odd-even transposition sort [2]. The two procedures may be combined into an efficient algorithm for one-sided Jacobi.

To combine both sorting and index ordering we require two different sweeps. For simplicity they are called the *forward sweep* and the *backward sweep* (see Fig. 3). These two sweeps are performed alternately during the computation. In either sweep the n indices are placed in a row at each step. Any two indices in the same parentheses form an index pair. Each sweep takes $2n$ steps to complete. It is easy to see that the two sweeps are equivalent when only the index ordering is considered. However, the forward sweep always places the larger index to the left in a parentheses, and so sorts the elements into a nondecreasing order. Similarly, the backward sweep sorts the elements into a nonincreasing order.

There are several special-purpose architectures introduced in the literature for parallel implementation of the odd-even Jacobi ordering algorithm, e.g., [17]. However, extra care has to be taken in order to efficiently implement the algorithm on general-purpose distributed memory machines. There is also a procedure for parallel implementation of the odd-even transposition sort [1]. When this procedure is applied



(a) Forward sweep



(b) Backward sweep

Figure 3: The odd-even Jacobi ordering. (a) forward sweep and (b) backward sweep

for the purpose of the one-sided Jacobi SVD computation, we may not achieve an active/idle ratio of processors higher than 0.5 because only half the total number of PEs in the system are active at each step. In the following we describe an improved procedure. In this procedure almost all the PEs are active during the computation and the communication cost can also greatly be reduced.

To simplify the discussion we only consider sorting n elements into a nonincreasing order on a parallel system of P PEs. We assume that $2P$ divides n and that the PEs are organised as a linear array. To distribute the n elements into the P PEs we first divide the numbers into $2P$ blocks, each block holding $n/(2P)$ elements. The $2P$ blocks are then allocated evenly to the P PEs, that is, each PE stores two blocks in its local memory. According to the ordering described in Fig. 3(a) the computation takes $2P$ steps to complete. At the beginning of the computation the elements in each block are sorted in a nonincreasing order. Afterwards, at each odd step each PE will merge the two blocks stored in the local memory and then transfer one block which contains $n/(2P)$ smaller elements to its right neighbour, while at each even step all the PEs, except the leftmost one which holds only one block of sorted elements, merge the two blocks and then transfer one block holding larger elements to their left neighbours. It is easy to see that in the above procedure almost all the PEs are active during the computation.

Size	Algorithm 2				Algorithm 3			
	odd-even		ring		odd-even		ring	
	T	S	T	S	T	S	T	S
128	17.85	11	15.98	10	17.78	11	15.99	10
256	137.5	11	130.1	11	137.5	11	130.2	11
384	472.2	11	456.3	11	472.2	11	456.5	11
512	1237	12	1156	11	1233	12	1160	11
640	2320	12	2298	13	2326	12	2242	12
768	4018	12	3888	12	4027	12	3894	12

Table 4: Comparison of odd-even and ring Jacobi orderings on the AP1000 with 8 processors organised as a linear ring.

Assume that the cost for sending (or receiving) a message of w words to (or from) a neighbouring PE is $c_0 + c_1 w$ where c_0 is the *start up* time and $1/c_1$ is the *transfer rate*. It is easy to see that the communication cost at each step is $c_0 + c_1 n/(2P)$ and the total communication cost is $2c_0P + c_1 n$. In contrast with the result using the conventional procedure for parallel implementation of the odd-even sorting algorithm, the total communication cost is reduced almost by half. (The analysis of the cost required by the conventional procedure is straightforward.) Therefore, our new procedure will be more efficient in computing the one-sided Jacobi.

We have implemented the above procedure on the AP1000 and obtained the same result as the ring ordering in terms of the number of sweeps for solving a given SVD problem (see Table 4). It is known that the odd-even ordering requires one more step to complete a sweep than the ring ordering (for n even). However, which one is more efficient is very much dependent upon the real machine configuration. If all PEs are configured in a linear array without a connection between the two boundary PEs, for example, the ring

Size	Algorithm 2		Algorithm 3	
	T	S	T	S
200	14.23	11	13.58	11
400	67.15	11	66.03	11
600	205.4	12	202.1	12
800	446.9	12	442.6	12
1000	848.1	12	842.9	12
1200	1522	13	1530	13
1400	2398	13	2386	13

Table 5: Results for the round robin ordering, with consideration of sorting, on the AP1000 with 100 processors organised as a linear array.

ordering would be less efficient because of the high cost for a message to travel from one end to the other. The results given in Table 4 show that the ring ordering will be more efficient if the PEs are physically interconnected in a ring.

6 Equivalence of Orderings

Equivalence of different orderings was defined in [18]. In that definition only the order of index pairs is considered, that is, two orderings are equivalent if they produce the same set of index pairs at the same step (by a relabelling of the initial indices). As we discussed in the previous sections, however, two orderings satisfying only this definition may not share the same convergence properties for one-sided Jacobi since an ordering which can also sort column norms in each sweep will certainly converge faster than the one which cannot. In the following we give an example to show that an ordering which cannot sort column norms may still perform efficiently as long as it can generate the same index pairs at the same step as one which does sorting.

As discussed in §4, the round robin ordering can generate the same index pairs at the same step as the ring ordering by a relabelling of the initial indices. It is easy to verify that sorting column norms can be guaranteed (although the final results may not be placed in a nonincreasing or nondecreasing order) if the round robin ordering with this initialisation of indices is adopted and other details are the same as for the application of the ring ordering.

We reimplemented the round robin ordering for the one-sided Jacobi SVD based on the above method. Some of the experimental results are given in Table 5. Comparing this table with Table 2, it is easy to see that a faster convergence rate can be achieved if a proper sorting procedure is adopted in the computation. Thus we may modify the definition of a *good ordering* for one-sided Jacobi as follows:

Definition A good ordering for one-sided Jacobi is one which

1. completes a sweep in a minimum number of steps;
2. has a simple and regular communication pattern between steps; and
3. sorts column norms in each sweep, or generates (by a relabelling of the initial indices) the same index pairs at the same step as an ordering which does sorting.

7 Conclusions

We have shown that parallel orderings without proper consideration of sorting may fail to achieve high efficiency when applied to the one-sided Jacobi SVD. Our parallel ring Jacobi ordering introduced in this paper can do both index ordering and sorting simultaneously in a sweep.

We have introduced an algorithm which combines the odd-even index ordering and the odd-even transposition sort, and given a procedure to efficiently implement this algorithm on general-purpose distributed memory machines. The experimental results show that both the ring and the modified odd-even Jacobi orderings can achieve the same convergence rate as the sequential cyclic Jacobi ordering in terms of the total number of sweeps.

The concept of equivalence of orderings can greatly simplify the work involved in analysing convergence properties of newly introduced orderings. However, the original definition only considers the equivalence of index ordering. Our experimental results show that this is not sufficient to affirm that two orderings give the same convergence properties when applied to the one-sided Jacobi SVD.

Finally, we have described how to efficiently implement a Jacobi ordering algorithm which cannot sort, but can generate the same set of index pairs at the same step as an ordering which does sort.

References

[1] S. G. Akl, *Parallel Sorting Algorithms*, Academic Press, Orlando, Florida, 1985.

[2] G. Baudet and D. Stevenson, "Optimal sorting algorithms for parallel computers", *IEEE Trans. on Computers*, C–27, 1978, 84–87.

[3] C. H. Bischof, "The two-sided block Jacobi method on a hypercube", in *Hypercube Multiprocessors*, M. T. Heath, ed., SIAM, 1988, pp. 612-618.

[4] R. P. Brent, "Parallel algorithms for digital signal processing", *Proceedings of the NATO Advanced Study Institute on Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, Leuven, Belgium, August, 1988, pp. 93-110.

[5] R. P. Brent and F. T. Luk, "The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays", *SIAM J. Sci. and Stat. Comput.*, 6, 1985, pp. 69-84.

[6] J. Demmel and K. Veselić, "Jacobi's method is more accurate than QR", *SIAM J. Sci. Stat. Comput.*, 11, 1992, pp. 1204-1246.

[7] P. J. Eberlein and H. Park, "Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory architectures", *J. Par. Distrib. Comput.*, 8, 1990, pp. 358-366.

[8] L. M. Ewerbring and F. T. Luk, "Computing the singular value decomposition on the Connection Machine", *IEEE Trans. Computers*, 39, 1990, pp. 152-155.

[9] G. E. Forsythe and P. Henrici, "The cyclic Jacobi method for computing the principal values of a complex matrix", *Trans. Amer. Math. Soc.*, 94, 1960, pp. 1-23.

[10] G. R. Gao and S. J. Thomas, "An optimal parallel Jacobi-like solution method for the singular value decomposition", in *Proc. Internat. Conf. Parallel Proc.*, 1988, pp. 47-53.

[11] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, second ed., 1989.

[12] P.. Henrici, "On the speed of convergence of cyclic and quasicyclic Jacobi methods for computing eigenvalues of Hermitian matrices", *J. Soc. Indust. Appl. Math.*, 6, 1958, pp. 144-162.

[13] M. R. Hestenes, "Inversion of matrices by biorthogonalization and related results", *J. Soc. Indust. Appl. Math.*, 6, 1958, pp. 51-90.

[14] T. J. Lee, F. T. Luk and D. L. Boley, *Computing the SVD on a fat-tree architecture*, Report 92-33, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York, November 1992.

[15] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing", *IEEE Trans. Computers*, C-34, 1985, pp. 892-901.

[16] F. T. Luk, "Computing the singular-value decomposition on the *ILLIAC IV*", *ACM Trans. Math. Softw.*, 6, 1980, pp. 524-539.

[17] F. T. Luk, "A triangular processor array for computing singular values", *Lin. Alg. Applics.*, 77, 1986, pp. 259-273.

[18] F. T. Luk and H. Park, "On parallel Jacobi orderings", *SIAM J. Sci. and Stat. Comput.*, 10, 1989, pp. 18-26.

[19] J. C. Nash, "A one-sided transformation method for the singular value decomposition and algebraic eigenproblem", *Comput. J*, 18, 1975, pp. 74-76.

[20] P. P. M. De Rijk, "A one-sided Jacobi Algorithm for computing the singular value decomposition on a vector computer", *SIAM J. Sci. and Stat. Comput.*, 10, 1989, pp. 359-371.

[21] R. Schreiber, "Solving eigenvalue and singular value problems on an undersized systolic array", *SIAM. J. Sci. Stat. Comput.*, 7, 1986, pp. 441-451.

[22] K. Veselić and V. Hari, "A note on a one-sided Jacobi algorithm", *Numerische Mathematik*, 56, 1990, pp. 627-633.

[23] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965, pp. 277-278.

[24] B. B. Zhou and R. P. Brent, "A parallel ring ordering algorithm for efficient one-sided Jacobi SVD computations" in *Proc. Sixth IASTED/ISMM Internat. Conf. on Parallel and Distributed Computing and Systems*, Washington, USA, October, 1994, pp. 369-372.