A Parallel Ring Ordering Algorithm for Efficient One-sided Jacobi SVD Computations[*]

B. B. Zhou and Richard 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. An efficient parallel ring Jacobi ordering for computing singular value decomposition is described. This ordering can generate $n(n-1)/2$ different index pairs and sort column norms at the same time. The one-sided Jacobi SVD algorithm using this parallel ordering converges in about the same number of sweeps as the sequential cyclic Jacobi algorithm. The issue of equivalence of orderings for one-sided Jacobi is also discussed. We show how an ordering which does not sort column norms into order may still perform efficiently as long as it can generate the same index pairs at the same step as one which does sorting. Some experimental results on a Fujitsu AP1000 are presented.

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 factorisation 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 $n \times n$ nonnegative 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

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

[*]To appear in *J. Parallel and Distributed Computing.* Received Oct 5, 1994; revised Dec 9, 1996; accepted Jan 3, 1997. Copyright ©1994-1997, the authors. rpb153 typeset using LaTeX

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. Consider the transformation by a plane rotation:

$$\begin{pmatrix} a_i & a_j \end{pmatrix} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = \begin{pmatrix} a_i' & a_j' \end{pmatrix} \tag{1}$$

where $c = \cos\theta$, $s = \sin\theta$, and a_i and a_j are the i-th and j-th columns of the matrix A. We choose θ to make a_i' and a_j' orthogonal. 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 [24].

It can be seen from equation (1) 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 noninteracting operations as possible at each parallel time step. Many parallel orderings have been introduced in the literature [3, 4, 5, 7, 8, 10, 15, 17, 18, 19]. These orderings were mainly designed for parallel eigenvalue decompositions. Special 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 thus introduce a parallel ring Jacobi orderings and show that this ordering can generate $n(n-1)/2$ different index pairs, and also sort n elements into order at the same time, where n is the problem size. We have implemented one-sided Jacobi SVD using this ordering on a distributed memory MIMD machine, the Fujitsu AP1000. Our experimental results show that the new algorithm can achieve the same efficiency as the sequential cyclic Jacobi algorithm for SVDs, i.e. the same total number of sweeps to convergence.

The issue of equivalence of orderings was originally discussed in [19]. By the definition two orderings are equivalent if they can generate the same set of index pairs at the same step by a relabelling of the initial indices. It is known from [19] that most existing Jacobi orderings can be classified into two equivalent groups and that the odd-even ordering [18] and the round robin ordering [5] are good representatives for each of these two groups. However, our experimental results show that two orderings satisfying 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 does not. In this paper we give an example to show that an ordering which does not sort column norms into order may still perform efficiently as long as it can generate the same index pairs at the same step as one which does sorting.

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 ring Jacobi ordering is then described in §4. Some experimental results on the AP1000 are presented in §5. In that section the issue of equivalence of orderings for one-sided Jacobi is also discussed. Some conclusions are given in §6.

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 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,

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

$$b_{ij}^{(k)} = (a_i^{(k)})^T a_j^{(k)} \tag{3}$$

and

$$b_{jj}^{(k)} = (a_j^{(k)})^T a_j^{(k)} = \|a_j^{(k)}\|^2. \tag{4}$$

where $\|x\|$ is the 2-norm of the vector x. Actually $b_{ii}^{(k)}$ and $b_{jj}^{(k)}$ can be updated through a very simple computation (see (11)–(12) below), so the two inner product operations in (2) and (4) need not be performed explicitly. (Because of possible cancellation in the update formulas, it may be desirable to perform an explicit inner product computation occasionally.)

Once $b_{ii}^{(k)}$, $b_{ij}^{(k)}$ and $b_{jj}^{(k)}$ are obtained, the Jacobi rotation parameters c and s can be computed as in the two-sided Jacobi method. Define

$$\alpha = 2b_{ij}^{(k)} = 2(a_i^{(k)})^T a_j^{(k)}, \tag{5}$$

$$\beta = b_{ii}^{(k)} - b_{jj}^{(k)} = \|a_i^{(k)}\|^2 - \|a_j^{(k)}\|^2, \tag{6}$$

$$\gamma = (\alpha^2 + \beta^2)^{1/2} \tag{7}$$

and

$$\gamma\prime = sign(\beta)\gamma \tag{8}$$

where $sign(x) = 1$ if $x \geq 0$ and $sign(x) = -1$ if $x < 0$. We obtain

$$c = \left(\frac{\beta + \gamma\prime}{2\gamma\prime} \right)^{1/2} \tag{9}$$

and

$$s = \frac{\alpha}{2\gamma\prime c}. \tag{10}$$

¿From α, β and $\gamma\prime$ defined above, we can obtain $b_{ii}^{(k+1)}$ and $b_{jj}^{(k+1)}$ as

$$b_{ii}^{(k+1)} = b_{ii}^{(k)} + \frac{\alpha^2}{2(\gamma\prime + \beta)} \tag{11}$$

and

$$b_{jj}^{(k+1)} = b_{jj}^{(k)} - \frac{\alpha^2}{2(\gamma\prime + \beta)}. \tag{12}$$

Since $\gamma\prime$ has the same sign as β, we have $b_{ii}^{(k+1)} \geq b_{ii}^{(k)}$ and $b_{jj}^{(k+1)} \leq b_{jj}^{(k)}$ after the above computation if $b_{ii}^{(k)} > b_{jj}^{(k)}$. Otherwise, $b_{ii}^{(k+1)} \leq b_{ii}^{(k)}$ and $b_{jj}^{(k+1)} \geq b_{jj}^{(k)}$. Thus the larger column norm is always further increased and the smaller one further decreased when orthogonalising two columns using this algorithm.

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 orthogonalisation of the two columns. From (11) and (12) we always have $b_{ii}^{(k+1)} \geq b_{jj}^{(k+1)}$. When the cyclic ordering is applied, therefore, the computed singular values will be sorted in a nonincreasing order.

Rotation Algorithm 3 The third algorithm was derived by Nash [20] and implemented on the ILLIAC IV by Luk [17]. 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,

$$\|a_i^{(k+1)}\|^2 - \|a_i^{(k)}\|^2 = \|a_j^{(k)}\|^2 - \|a_j^{(k+1)}\|^2 \geq 0. \tag{13}$$

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

The parameters α, β and γ are calculated as in Algorithm 1. If $\beta < 0$, then

$$s = \left(\frac{\gamma - \beta}{2\gamma}\right)^{1/2}, \tag{14}$$

$$c = \frac{\alpha}{2\gamma s}, \tag{15}$$

$$\|a_{ii}^{(k+1)}\|^2 = \|a_{ii}^{(k)}\|^2 + \frac{\gamma - \beta}{2} \tag{16}$$

and

$$\|a_{jj}^{(k+1)}\|^2 = \|a_{jj}^{(k)}\|^2 - \frac{\gamma - \beta}{2}. \tag{17}$$

Otherwise, we compute

$$c = \left(\frac{\gamma + \beta}{2\gamma}\right)^{1/2}, \tag{18}$$

$$s = \frac{\alpha}{2\gamma c}, \tag{19}$$

$$\|a_{ii}^{(k+1)}\|^2 = \|a_{ii}^{(k)}\|^2 + \frac{\alpha^2}{2(\gamma + \beta)} \tag{20}$$

and

$$\|a_{jj}^{(k+1)}\|^2 = \|a_{jj}^{(k)}\|^2 - \frac{\alpha^2}{2(\gamma + \beta)}. \tag{21}$$

matrix size ($m = n$)	60	80	100	120	140	160	180	200
Algorithm 1 (sweeps)	10	11	12	11	12	12	12	12
Algorithm 2 (sweeps)	8	9	8	9	9	9	9	9
Algorithm 3 (sweeps)	8	9	8	9	9	9	9	10

Table 1: Results for the cyclic Jacobi ordering on a Sun workstation.

It is known from numerical experiments that an implementation which uses Rotation Algorithm 2 or 3 is more efficient than one using 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 in Rotation Algorithms 2 and 3 we always have $b_{ii}^{(k+1)} \geq b_{jj}^{(k+1)}$ 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, the norm of column i may be increased when two columns i and j are orthogonalised in a sweep, but the norm of column j may 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, 21]. It is probably the main reason why applying Rotation Algorithm 2 or 3 is more efficient than applying Rotation Algorithm 1. (We found that Hüper and Helmke recently proved that Jacobi methods with sorting may achieve an optimal convergence rate for the computation of singular value decomposition as well as symmetric eigenvalue decomposition [14].)

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.

To alleviate the problem caused by adopting Rotation Algorithm 1, a pivoting strategy was proposed in [21]. Each sweep is divided into $n-1$ sub-sweeps. In the k^{th} sub-sweep a set of $n-k$ columns $(a_k, a_{k+1}, \cdots, a_n)$ are involved. Each sub-sweep consists of two steps. First the column with largest norm is determined and interchanged with the first column in the set. The first column, which now is the one with largest norm in the set, is then orthogonalised with all other columns in the set once. It is easy to prove that the first column still has the largest norm on the completion of the sub-sweep.

A so called *accelerated one-sided Jacobi method* for symmetric positive definite eigenvalue problem is analysed in [6]. This algorithm (originally introduced in [23]) consists of three steps. First the Cholesky factor L of the given matrix A is formed using *diagonal pivoting*. There is a permutation matrix P such that $P^TAP = LL^T$. Next the singular values σ_i and left singular vectors u_i of L are computed using one-sided Jacobi. The eigenvalues $\lambda_i = \sigma_i^2$ and eigenvectors $v_i = Pu_i$ are then obtained. Since the effect of the diagonal pivoting in the initial step is to reorder or to sort the row norms of matrix L, this tends to improve the condition number of L (according to the analysis in [6]).

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 some results from our experiments and show that optimal efficiency may not be obtained without a proper procedure for

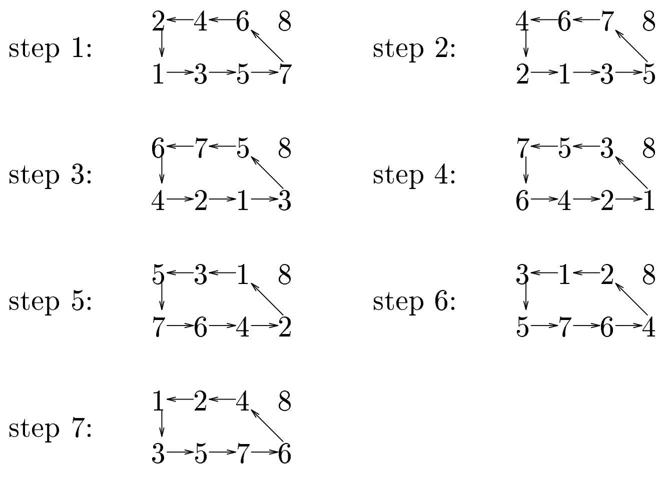
step 1: 2 4 6 8 / 1 3 5 7
step 2: 4 6 7 8 / 2 1 3 5
step 3: 6 7 5 8 / 4 2 1 3
step 4: 7 5 3 8 / 6 4 2 1
step 5: 5 3 1 8 / 7 6 4 2
step 6: 3 1 2 8 / 5 7 6 4
step 7: 1 2 4 8 / 3 5 7 6

Figure 1: The round robin ordering.

matrix size ($m = n$)		200	400	600	800	1000	1200	1400
Algorithm 1	time (sec.)	14.43	72.74	226.4	519.6	994.4	1811	2978
	sweeps	12	13	14	15	15	16	18
Algorithm 2	time (sec.)	19.31	97.59	290.4	645.5	1255	2219	3495
	sweeps	16	17	17	17	18	18	18
Algorithm 3	time (sec.)	19.11	94.45	291.6	641.9	1228	2207	3439
	sweeps	16	16	17	17	17	18	18

Table 2: Results for the round robin ordering, with index sorting, on an AP1000 with 100 processors organised as a linear array.

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. In the figure the indices are placed in two rows. The indices in the same column form a index (or Jacobi) pair. In each step $n/2$ index pairs are generated. The indices are then shifted according to the arrow lines, so another $n/2$ different index pairs are produced in the next step. Thus the $n(n-1)/2$ different pairs can be generated in $n-1$ steps, which complete one sweep of ordering.

Using the round robin ordering, $n/2$ pairs of columns of the matrix are simultaneously orthogonalised in each step. When Rotation Algorithm 2 or 3 described in the §2 is applied, the problem encountered is which column in a column pair should be considered as the column associated with index i (or j) so that its column norm is increased (or decreased). This problem does not occur when Rotation Algorithm 1 is applied because the algorithm never attempts to sort the norms.

If we count from left to right (between columns) and from bottom up (within each column), the n indices in Fig. 1 are initially placed in a nondecreasing order. After $n-1$ steps (or a sweep) this order is again restored. Thus our first attempt is to use an *index sorting* procedure, that is, we always increase the norm of a column associated with smaller index when orthogonalising a pair of columns using Rotation Algorithm 2 or 3. Since the indices are placed in a nondecreasing order after each sweep, the final results should eventually be in a nonincreasing order.

We implemented the above idea on the Fujitsu AP1000. In the experiment both singular values and singular vectors are calculated using 100 PEs which are organised as a one-dimensional array.

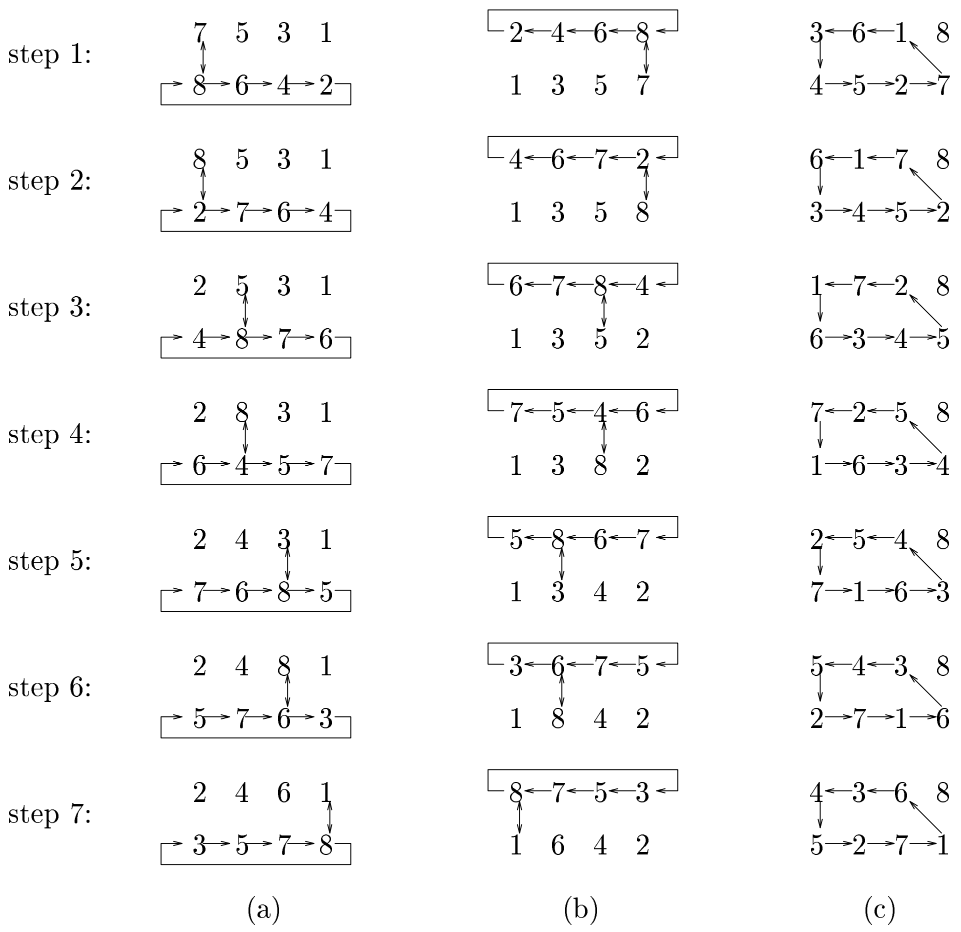

Figure 2: The ring Jacobi ordering. (a) forward sweep, (b) backward sweep, and (c) the round robin ordering with a new initialisation of indices.

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. The main reason we believe is that, although the natural order of the indices is restored, the round robin ordering cannot sort the column norms properly in a nonincreasing (or nondecreasing) order in one sweep (or $n-1$ steps). In the next two sections we introduce a ring Jacobi ordering and show that more efficient results may be obtained if sorting of columns norms in each sweep is considered.

4 The Ring Jacobi Ordering

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 are generated at the next step. The up-and-down arrow in Fig. 2, moving from the leftmost column towards the rightmost column by one column every two steps, indicates the exchange of two indices in the column before one is shifted. This arrow plays a crucial rule in both index ordering and sorting. Without it the indices initially placed in the same row can never meet each other and the elements will not be sorted. In the following we claim that 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.

It can easily be verified that the forward sweep and the backward sweep are essentially the same. The difference is that, when sorting is considered, one sorts the elements into nondecreasing order and the other sorts the elements into nonincreasing order. Thus we only prove that the forward sweep can do both index ordering and sorting.

In the following discussion we assume that the total number of indices (or elements) n is even. For n odd we may simply pad one additional index, which will not affect the final result. The n indices are placed in $n/2$ columns. These columns are numbered from left to right, starting from column one.

Proposition 1 *One forward sweep can generate exactly $n(n-1)/2$ distinct Jacobi pairs.*

Proof. Since each step generates $n/2$ Jacobi pairs, a total number of $n(n-1)/2$ Jacobi pairs can be generated in $n-1$ steps. To ensure those Jacobi pairs are all distinct, any two indices must not meet each other more than once. Thus we need only to prove that each index can meet all other indices exactly once in a sweep.

We first show that the two rows will be exchanged and the order of the indices in the same row reversed after a sweep. In other words, if we define the index initially on the top of the k^{th} column as t_k and the one at the bottom of the k^{th} column as b_k for $1 \leq k \leq n/2$, we want to prove that t_k (or b_k) will be shifted to the bottom (or the top) of the $(n/2-k+1)^{th}$ column for $1 \leq k \leq n/2$ after $n-1$ steps. An example is depicted in Fig. 2(a). In that figure the indices are initially placed in a decreasing order at step 1. After $n-1$ steps the two rows are swapped and the order of indices in each row becomes increasing (which is not drawn in the figure, but can easily be worked out from step 7).

Index b_k is originally placed at the bottom of the k^{th} column. Following the arrow lines it moves, starting at step 1, one column each step to the right. Thus it will be shifted to the rightmost column at the $(n/2-k+1)^{th}$ step (or arrive at the leftmost column after that step). Since it travels twice as fast as the up-and-down arrow, it is easy to see that this index will take another $n/2-k+1$ steps to catch up with the arrow at the $(n/2-k+1)^{th}$ column and then be lifted to the top and stay there permanently.

Index t_k cannot move until the up-and-down arrow comes to the k^{th} column. Since it moves one column every two steps, the arrow will reach that column at the $(2k-1)^{th}$ step. After being dragged down by the up-and-down arrow, t_k then moves $n/2-k$ steps to reach the rightmost column Since one sweep has $n-1$ steps and

$$(n-1)-(2k-1)-(n/2-k) = n/2-k,$$

t_k will continue to travel (one column per step from the rightmost column) to reach the $(n/2-k)^{th}$ column at step $n-1$. Therefore, after that step it is settled at the bottom of the $(n/2-k+1)^{th}$ column.

We now show that t_k for $1 \leq k \leq n/2$ will meet all other $n-1$ indices exactly once in a sweep. When the up-and-down arrow comes to a column, the index on the top will be dragged down and shifted forward. Thus all the indices on the left of the k^{th} column will come across that column. Before it is dragged down, t_k can certainly meet $2(k-1)+1$ distinct indices from its left (including the one originally placed at the bottom of the same column), that is, those t_is for $1 \leq i \leq k-1$ and b_is for $1 \leq i \leq k$. On the way towards the rightmost column, index t_k will meet all other t_is for $k+1 \leq i \leq n/2$. ¿From the previous discussion, b_i for $k+1 \leq i \leq n/2$ will be placed on the top of the first $n/2 - k$ columns after $n-1$ steps. Since these indices arrive at their destination earlier than t_k, it is obvious that t_k can meet all those indices when coming across the first $n/2-k$ columns to its final position, that is, the bottom of the $(n/2-k+1)^{th}$ column.

Finally we show that b_k will meet all other $n-1$ indices exactly once in a sweep. Since we have proved that every index originally placed on the top can meet all other $n-1$ indices once in $n-1$ steps, we need only to prove that b_i for $1 \leq i \leq n/2$ will meet each other only once in a sweep.

It is seen that all b_is except b_1 move in the same speed of one column per step towards the rightmost column (starting from step 1). Thus they cannot meet each other until they come to the leftmost column. The rightmost index $b_{n/2}$ comes to the first column at step 2, where it catches up with the up-and-down arrow (and meets b_1 as well) and is then lifted to the top and stay there permanently. Since all other b_i for $2 \leq i \leq n/2-1$ will follow $b_{n/2}$ to come across the leftmost column, $b_{n/2}$ can certainly meet them all, but only once in a sweep. Similarly, b_k will come across the first $n/2-k$ columns and meet those b_is for $k+1 \leq i \leq n/2$ and then stay at the $(n/2-k+1)^{th}$ column where it encounters all other b_is for $1 \leq i \leq k-1$ since the order of those indices is swapped after $n-1$ steps. □

Actually our ring ordering is equivalent to the round robin ordering. To see that the two orderings are equivalent, we first permute the initial positions of n indices for the round robin ordering and then show that the orderings can generate the same index pairs at any step.

The initial positions of indices for the round robin ordering in Fig. 1 is reorganised as follows. Divide the original index pairs (at step 1) into two almost equal parts (if the number of index pairs is odd, the right part will contain one more pair than the left part). Next swap the two rows in the left part. Finally fold the two parts together from the middle so that the index pairs in the two parts are interleaved. It should be noted that the rightmost index pair in the original pattern must also be the rightmost one after the permutation.

When we apply the round robin ordering with the above new initialisation, the same index pairs for each step in the ring ordering can then be generated. The detailed proof of this claim is tedious and thus omitted. An example of $n = 8$ is depicted in Fig. 2.

We now consider sorting. The sorting procedure using a forward sweep is described as follows: The n elements to be sorted are initially placed in two rows. During sorting the data flow pattern is also the same as that for index ordering. To sort these elements in a nondecreasing order, however, additional *compare-exchange* operations in each step will be required. Each step now consists of two sub-steps. The first sub-step compares the two elements in each column and places the smaller one on the top and the larger one at the bottom except in even steps the larger element is placed on the top if the column has a up-and-down arrow in it. The second sub-step simply shifts the elements located at the bottom to the next column according to the arrow ring, which is the same as that for index ordering. The n elements can be sorted in a nondecreasing order after $n-1$ such steps.

It should be noted that, since the up-and-down arrow indicates the exchange of the two elements in the column, these arrows in even steps can be removed by letting the smaller elements be placed at the top of the corresponding columns. Therefore, we can place smaller elements on the top and larger ones at the bottom for all columns in both odd and even steps as long as the up-and-down arrow appears only at odd steps. When the up-and-down arrow appears at an odd step, the two elements in the column have to exchange their positions before the shift takes place, which is the same as that for index ordering.

In the following we prove that the above sorting procedure works correctly. To simplify the discussion we assume that all elements have different values.

Proposition 2 *One forward sweep can sort n elements in a nondecreasing order.*

Proof. Since only the elements placed at the bottom can be shifted, it is clear that only the larger element in a column which does not contain the up-and-down arrow can move forward. When the up-and-down arrow appears in a column at an odd step, however, the smaller element in that column will be placed at the bottom. Thus the up-and-down arrow drags "small" elements down from the top and forces them to move forward. We show that during sorting no elements can fall behind this arrow to its left and no elements can overtake this arrow to its right.

Let s_e denote the smaller element and l_e the larger element in a column which contains the up-and-down arrow. At step 1 the up-and-down arrow appears in the leftmost column. The smaller element s_e is then shifted rightward to column 2. At the next step (an even step) s_e is compared with the element on the top of column 2. If s_e is smaller, it will be lifted to the top. Otherwise, s_e (which is then not called s_e) is shifted. Meanwhile, l_e in column 1 is compared with an element which is just moved in from the rightmost column. The larger one of these two is shifted to column 2. The up-and-down arrow then comes to column 2 at step 3 and pushes the smaller element (the new s_e which is no greater than the old s_e) to the next column and lifts the larger one (the new l_e which is no smaller than the old l_e) to the top. In general, at the $(2k)^{th}$ step s_e pushes a larger element in column $k+1$ to move rightward and l_e blocks a smaller one to come over column k. At the next immediate odd step, that is, step $2k+1$ the up-and-down arrow will arrive at column $k+1$ to lift l_e to the top and to push s_e to move rightward.

It is easy to see from the above discussion that all elements (except the one which is the larger element in the leftmost column at the initial step) must move forward to reach the rightmost column, back to the leftmost column and then continue to move rightward (if possible). Since small elements is more likely to be lifted to the top, they move slower than those large ones. However, large elements will be slowed down by the up-and-down arrow. As a consequence, large elements and small elements will be gathered respectively at the left side and the right side of the up-and-down arrow. As this arrow reaches the rightmost column, small elements are then placed in the left columns and large ones are in the right columns. In the following we show that after $n-1$ steps the n elements will be placed in a perfect nondecreasing order.

Let e_i denote an element with $e_i < e_j$ for $i < j$. We first show that after $n-1$ steps e_1 and e_n are placed at the bottom of the leftmost column and on the top of the rightmost column, respectively. Because e_1 is the smallest element, it will stay at the top of its original column until the up-and-down arrow arrives. Then it becomes the smallest s_e and moves rightward one column every two steps with the up-and-down arrow. Since e_n is the largest element, it will travel without stopping until it catches up with the up-and-down arrow and become the largest l_e. Then it also moves rightward with the up-and-down arrow. At step $n-1$, therefore, e_1, e_n and the up-and-down arrow must be in the rightmost column. It is thus clear that e_n will be lifted to the top and e_1 be shifted to the bottom of the leftmost column after that step.

We now show that the second smallest element e_2 must appear on the top of the leftmost column after $n-1$ steps. Suppose that e_2 is initially placed behind the smallest element e_1. It cannot move until the up-and-down arrow arrives and then it becomes s_e and is pushed to move forward by the arrow. When it reaches the column where e_1 stays, certainly e_2 will overtake e_1 after that step since it is greater. If e_2 is initially placed in front of e_1, it will be pushed to move forward by e_1. Thus no matter where e_2 is initially placed, it will reach the leftmost column before e_1. After having arrived at the leftmost column, it will stay on the top of that column and can move only when e_1 comes. However, e_1 comes to the leftmost column immediately after the last step!

Assume that the first $k-1$ elements e_i for $1 \le i \le k-1$ are in their proper positions after $n-1$ steps. If all the larger elements can move to the right side of element e_k during sorting, e_k must come to its correct position after $n-1$ steps. To prove this we consider the following extreme case.

In order to simplify the discussion, we assume that k is even. The case for k odd is similar. As the elements are placed in two rows, at least $k/2$ of those elements e_i for $1 \le i \le k$ will initially be placed on the top row since no other elements are smaller. Once any of these elements is lifted to the top, only a smaller element coming from its left can replace it, or the up-and-down arrow arrives to drag it down. Thus we place at the initial step these k elements in the rightmost $k/2$ columns so that e_k may be shifted as far as possible to the right after arriving at the leftmost column. We also place e_{k+1} in the leftmost column to keep it as distant as possible from e_k. Since e_{k+1} can never overtake an element which is larger, all other large elements e_i for $k+2 \leq i \leq n$ will be on the right side of e_k if e_{k+1} is after $n-1$ steps. An example is given in Fig. 3, where $n = 10$, $k = 4$, $e_k = 4$ and $e_{k+1} = 5$.

There are $k/2$ small elements being placed on the top row at step 1. Since there is no any smaller element on their left, those elements can move only after the up-and-down arrow arrives. It is obvious that e_{k+1} can certainly overtake them before arriving to the leftmost column. Because there are only $k/2$ smaller elements moving to the leftmost column before e_{k+1}, e_k cannot be shifted over the $(k/2)^{th}$ column before being caught up by e_{k+1}.

We now count how many steps for e_{k+1} to move to the $(k/2)^{th}$ column from its initial position. Since it is the smallest one among those e_i for $k+1 \leq i \leq n$, it then acts as s_e and move with the up-and-down arrow until it comes at the $(n-k)^{th}$ step to the $((n-k)/2+1)^{th}$ column where it meets a smaller element first time. After that it will come across those smaller elements to the rightmost column one column per step in $k/2-1$ steps and then to the $(k/2)^{th}$ column in another $k/2$ steps. Therefore, e_{k+1} will catch up with e_k in $(n-k) + (k/2-1) + k/2 = n-1$ steps. Immediately after the $(n-1)^{th}$ step, e_{k+1} is shifted to the right of e_k and comes to the bottom of the $(k/2+1)^{th}$ column, which is indeed its correct destination. $\Box$

Since the sorting has the same data flow pattern as the index ordering, the two procedures can be done simultaneously, that is, when the elements are placed in a nonincreasing (or nondecreasing) order at the beginning of each sweep, a total number of $n(n-1)/2$ different Jacobi pairs can be produced by a forward (or backward) sorting procedure. This can easily be seen from Fig. 2(a) (or (b)) where the numbers are not considered as indices, but as the values of n elements. Then the elements are sorted from nonincreasing (or nondecreasing) order to nondecreasing (or nonincreasing) order and they also meet each other exactly once after $n-1$ steps.

It should be noted that the required Jacobi pairs cannot be generated during sorting if the column norms are initially placed in an arbitrary order. However, this happens only at the first a few sweeps and the column norms tend to be sorted after each sweep of computations. If the two sweeps (forward and backward) are applied alternately, therefore, both index ordering and sorting can be done simultaneously after the first a few sweeps. This process is just equivalent to that in sequential SVD computation using cyclic orderings.

5 Experimental Results and Discussions

In order to see the importance of sorting the column norms in a parallel implementation of the one-sided Jacobi SVD, we implemented our ring ordering algorithm on the Fujitsu AP1000 at the Australian National University. In the experiment both singular values and singular vectors are computed on the AP1000, which is configured as a one-dimensional array.

An algorithm without partitioning is not very useful in practice for general-purpose parallel computation because the system configuration is fixed, but the size of user's problem may vary. Our partitioning strategy is based on the method described in [22]. However, a major difference is that we take sorting into consideration. Assume that the given system has p processors and $2p$ divides n. We first divide n columns of the matrix into $2p$ blocks. At the beginning of a sweep, the columns in each block are orthogonalised with each other exactly once using the cyclic-by-rows ordering. If Rotation Algorithm 2 or 3 is applied, the norms of columns in each block should be sorted in order. We then consider each block as a super index and follow the designed ordering

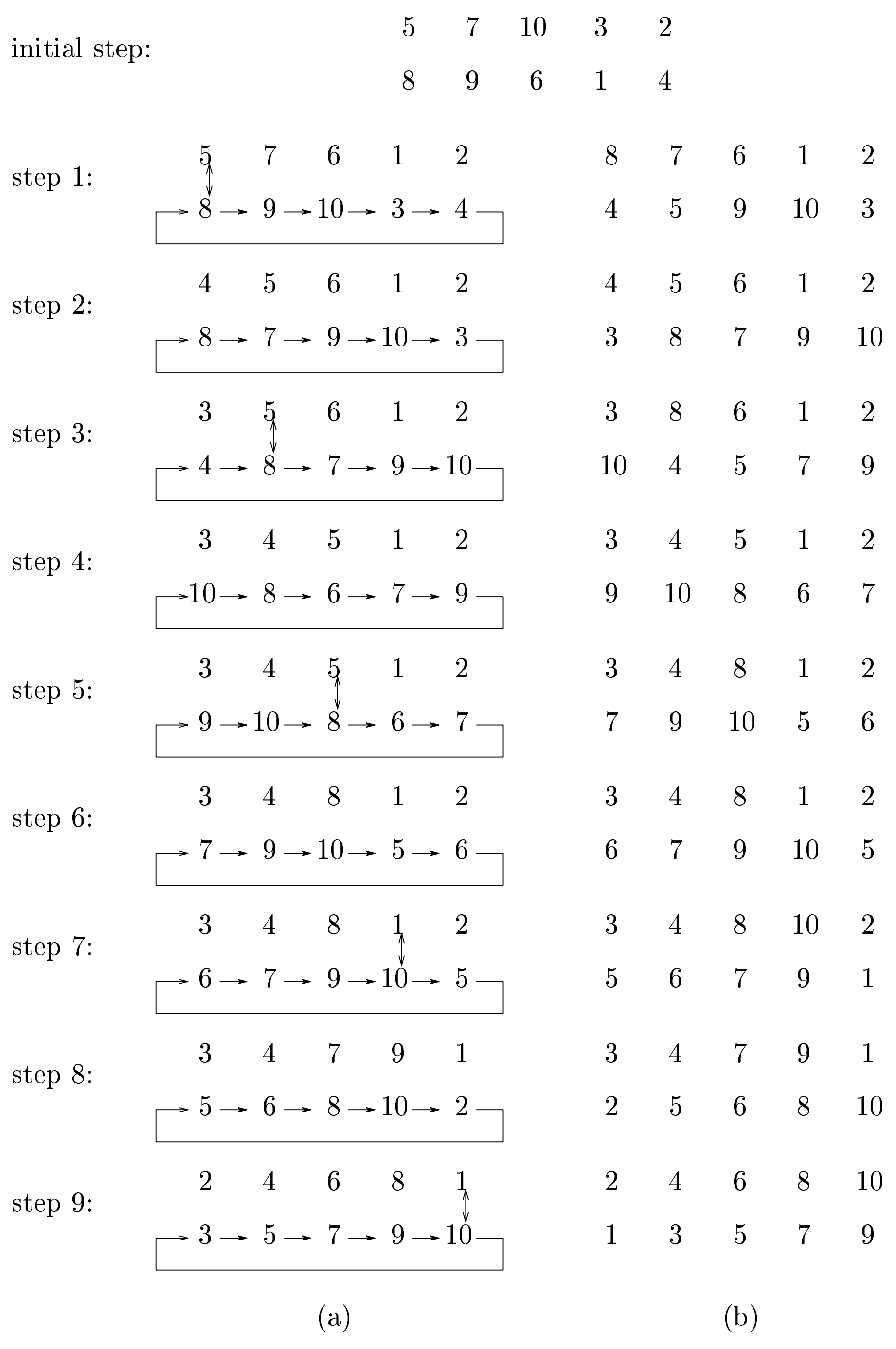

Figure 3: An example of sorting ten integers. (a) after exchange (the first substep) and (b) after shift (the second substep).

matrix size ($m = n$)		200	400	600	800	1000	1200	1400
Algorithm 1	time (sec.)	11.58	65.35	210.8	499.5	945.2	1702	2787
	sweeps	12	13	14	15	15	16	17
Algorithm 2	time (sec.)	10.01	57.39	187.1	416.7	799.1	1407	2280
	sweeps	10	11	12	12	12	12	13
Algorithm 3	time (sec.)	10.02	57.42	185.6	416.3	799.8	1445	2272
	sweeps	10	11	12	12	12	13	13

Table 3: Results for the ring Jacobi ordering on an AP1000 with 100 processors organised as a linear array.

so that $p(p-1)/2$ super index pairs can be generated in $p-1$ super steps. In the computation of each super index pair each column in one block must be orthogonalised with each column in the other block once only using the cyclic-by-rows ordering, but no columns in the same block are orthogonalised. If a block in a super index pair is considered as the column associated with index i (or index j), the norms of all columns in that block should be increased (or decreased) during the orthogonalisation with the columns in the other block when Rotation Algorithm 2 or 3 is applied. It is easy to show that the sorting procedure is implemented on the completion of the sweep.

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 Algorithm 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). 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 thus clear evidence which shows how important it is to adopt a proper sorting procedure in each sweep.

It is known that the odd-even index ordering [18] 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. Similar to the ring Jacobi ordering, to combine these sorting and index ordering procedures also require two different sweeps. One sorts the column norms from a nondecreasing order to a nonincreasing order and the other sorts the column norms from a nonincreasing order to a nondecreasing order. The detailed description of this ordering can be found in [25].

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 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.

In the following we give an example to show that an ordering (the round robin ordering) which does not sort column norms may still perform efficiently as long as it can generate the same index pairs at the same step as one (the ring ordering) which does sort. Since most existing Jacobi ordering are equivalent in terms of Jacobi pairs to either the round robin (or our ring) ordering, or the odd-even ordering, they may then efficiently be applied for the One-sided Jacobi SVD if certain special care is taken into consideration.

As discussed in §4, the round robin ordering is equivalent to the ring ordering in terms of the generation of Jacobi pairs. When the round robin ordering is adopted with a proper relabelling of initial indices, the same Jacobi pairs can be generated at the same step as the ring ordering. If the

matrix size ($m = n$)			128	156	384	512	640	768
Algorithm 2	odd-even	time (sec.)	17.85	137.5	472.2	1237	2320	4018
		sweeps	11	11	11	12	12	12
	ring	time (sec.)	15.98	130.1	456.3	1156	2298	3888
		sweeps	10	11	11	11	13	12
Algorithm 3	odd-even	time (sec.)	17.87	137.5	472.2	1233	2326	4027
		sweeps	11	11	11	12	12	12
	ring	time (sec.)	15.99	130.2	456.5	1160	2242	3894
		sweeps	10	11	11	11	12	12

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

matrix size ($m = n$)		200	400	600	800	1000	1200	1400
Algorithm 2	time (sec.)	14.23	67.15	205.4	446.9	848.1	1522	2398
	sweeps	11	11	12	12	12	13	13
Algorithm 3	time (sec.)	13.58	66.03	202.1	442.6	842.9	1530	2386
	sweeps	11	11	12	12	12	13	13

Table 5: Experimental results 5. The round robin ordering with the consideration of sorting is adopted and the experiment is conducted on the AP1000 with 100 processors organised as a linear array.

same sorting procedure is applied to each Jacobi pair, the larger norm goes with column i and the smaller one with column j if it is so when using the ring ordering. The two orderings will generate the same intermediate results at any time. Therefore, they must perform equally well for the same problem. With this procedure the two orderings are made equivalent also in terms of sorting even though the column norms are not sorted in a proper order such as nonincreasing, or nondecreasing using the round robin ordering.

We re-implemented 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.

6 Conclusions

In this paper we showed that parallel orderings without proper consideration of sorting may fail to achieve high efficiency when applied to the one-sided Jacobi SVD. We proved that our parallel ring Jacobi ordering (and the odd-even Jacobi ordering mentioned in the paper) can do both index ordering and sorting simultaneously in a sweep. The experimental results demonstrate that the one-sided Jacobi SVD algorithm using these parallel orderings converge in about the same number of sweeps as the sequential cyclic Jacobi algorithm.

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. We then discussed how to efficiently implement a Jacobi ordering algorithm which does not sort, but can generate the same set of index pairs at the same step as an ordering which does sort.

Acknowledgements

The authors wish to thank the anonymous referees for helpful comments and suggestions.

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] K. Hüper and U. Helmke, "Structure and convergence of Jacobi-type methods", Tech. Report, TUM-LNS-TR-95-02, Institute of Network Theory and Circuit Design, Technical University of Munich, Germany, 1995.

[15] 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.

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

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

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

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

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

[21] 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.

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

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

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

[25] B. B. Zhou and R. P. Brent, "On the parallel implementation of the one-sided Jacobi algorithm for singular value decompositions", *Proc. of 3rd Euromicro Workshop on Parallel and Distributed Processing*, San Remo, Italy, Jan, 1995, pp. 401-408.