

# A Block JRS Algorithm for Highly Parallel Computation of SVDs

Mostafa I. Soliman<sup>1</sup>, Sanguthevar Rajasekaran<sup>2</sup>, and Reda Ammar<sup>2</sup>

<sup>1</sup> Computer & System Section, Electrical Engineering Department, South Valley University, Aswan, Egypt

`soliman@mail.eun.eg`

<sup>2</sup> Department of Computer Science and Engineering, University of Connecticut, Storrs, USA  
{rajasek, reda}@engr.uconn.edu

**Abstract.** This paper presents a new algorithm for computing the singular value decomposition (SVD) on multilevel memory hierarchy architectures. This algorithm is based on one-sided JRS iteration, which enables the computation of all Jacobi rotations of a sweep in parallel. One key point of our proposed block JRS algorithm is reusing the loaded data into cache memory by performing computations on matrix blocks ( $b$  rows) instead of on strips of vectors as in JRS iteration algorithms. Another key point is that on a reasonably large number of processors the number of sweeps is less than that of one-sided JRS iteration algorithm and closer to the cyclic Jacobi method even though not all rotations in a block are independent. The relaxation technique helps to calculate and apply all independent rotations per block at the same time. On blocks of size  $b \times n$ , the block JRS performs  $O(b^2n)$  floating-point operations on  $O(bn)$  elements, which reuses the loaded data in cache memory by a factor of  $b$ . Besides, on  $P$  parallel processors,  $(2P-1)$  steps based on block computations are needed per sweep.

## 1 Introduction

Singular value decomposition (SVD) is an extremely powerful and useful tool in Linear Algebra. There are many applications of the SVD in scientific computing and digital signal processing. See [1] for some applications. SVD problem is a very computationally intensive problem that needs to exploit the growing availability of parallel hardware. Thus, many researchers have worked on designing efficient techniques to compute SVDs in parallel to reduce the execution time especially for real time applications. This paper presents our proposed algorithm block JRS, which is based on the one-sided JRS iteration algorithm proposed in [2]. The JRS iteration algorithms enable the computation of all Jacobi rotations of a sweep in parallel. Each sweep of this algorithm consists of several rotations and in each rotation, the value of an off-diagonal element is decreased to a fraction of its current value (instead of to zero).

The well-known sequential bidiagonalization-based Golub-Kahan-Reinsch SVD algorithm [3] takes  $O(mn^2)$  time on an  $n \times m$  matrix. For large matrices the execution

time may be unacceptable. Thus, it is essential to develop efficient parallel algorithms. The bidiagonalization-based SVD algorithm has been found to be difficult to parallelize and hence most works on parallel SVD focus on Jacobi-based techniques. Both two-sided Jacobi and one-sided Jacobi techniques have been studied in this context. Brent and Luk [4] presented a parallel one-sided SVD algorithm using a linear array of  $n$  processors, with a run time of  $O(mnS)$ , where  $S$  is the number of sweeps. They also presented an  $O(nS)$  time algorithm to compute the singular values of a symmetric matrix using an array of  $n^2$  processors. Zhou and Brent [5] described an efficient parallel ring ordering algorithm for one-sided Jacobi.

Becka and Vajtersic [6] presented a parallel block two-sided Jacobi algorithm on hypercubes and rings with a run time of  $O(n^2S)$ . They also gave an  $O(nS)$  time algorithm on meshes with  $n^2$  processors [7]. Becka et al. [8] proposed a dynamic ordering algorithm for a parallel two-sided block-Jacobi SVD with a run time of  $O(n^2S)$ . Oksa and Vajtersic [9] designed a systolic two-sided block-Jacobi algorithm with a run time of  $O(nS)$ . Strumpfen et al. [10] presented a stream algorithm for one-sided Jacobi that has a run time of  $O(n^3Sp^2)$ , where  $p$  is the number of processors ( $p$  being  $O(n^{1/2})$ ). They created parallelism by computing multiple Jacobi rotations independently and applying all the transformations thereafter. Rajasekaran *et al.* [2] employed the idea of separating rotation computations and transformations. They used a novel scheme to reduce the number of sweeps.

We propose a new algorithm for computing SVDs based on the relaxation technique proposed by Rajasekaran *et al.* [2]. This algorithm is fundamentally different from all the algorithms that have been proposed for SVD. The proposed block JRS algorithm employs the same idea of decreasing (relaxing) the off-diagonal elements proposed by Rajasekaran *et al.* [2]. However, it differs from the JRS algorithms in the partitioning strategy of the input matrix among processors. Besides, it is a highly parallel algorithm designed to exploit the memory hierarchy by processing blocks to reuse the loaded data into cache memories. However, JRS iteration algorithms are working on strips of vectors. On the block JRS, the input matrix  $A_{m \times n}$  is divided into blocks of size  $b \times n$ , where  $\lfloor m/2P \rfloor \leq b \leq \lceil m/2P \rceil$  and  $P$  is the number of available processors. This means that not all block sizes are necessarily the same, which is more flexible. One of the key points of the block JRS is reusing the loaded data into cache memory by performing the computations on blocks ( $b$  rows) instead on strips of vectors. To be specific, it performs  $O(b^2n)$  floating-point operations on  $O(bn)$  elements, which reuses the loaded data in cache memory by a factor of  $b$ . Another key point is the number of sweeps is less than one-sided JRS iteration algorithm and closer to the cyclic Jacobi method even though not all rotations in a block are independent. The relaxation technique helps to calculate and apply multiple rotations per block at the same time.

This paper is organized as follows. The following section reviews the singular value decomposition. Two-sided and one-sided Jacobi as well as one-sided JRS algorithms are explained. Section 3 describes the proposed block JRS iteration algorithms. Our results are shown in Section 4. Finally, Section 5 concludes our paper.

## 2 Singular Value Decomposition

A singular value decomposition of a real matrix  $A_{m \times n}$  is defined as the computation of three matrices  $U_{m \times m}$ ,  $\Sigma_{m \times n}$ , and  $V_{n \times n}$  such that:

$$A_{m \times n} = U_{m \times m} \Sigma_{m \times n} V_{n \times n}^T, \tag{1}$$

where  $U_{m \times m}$  and  $V_{n \times n}$  are orthogonal matrices (i.e.  $U^T U = I_m$  and  $V^T V = I_n$ ), and  $\Sigma_{m \times n}$  is a diagonal matrix  $diag(\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_{n-1})$  on top of  $(m-n)$  rows of zeros, assuming that  $m \geq n$ . The  $\sigma_i$  are the singular values of  $A_{m \times n}$ . Matrix  $U_{m \times m}$  contains  $n$  left singular vectors, and matrix  $V_{n \times n}$  consists of  $n$  right singular vectors. The singular values and singular (column) vectors of  $U_{m \times m}$  and  $V_{n \times n}$  form the relations

$$A v_i = \sigma_i u_i \quad \text{and} \quad A^T u_i = \sigma_i v_i. \tag{2}$$

All the existing parallel algorithms use the Jacobi iteration as the basis. The Jacobi iteration algorithm attempts to diagonalize the input matrix  $A_{m \times n}$  by a series of Jacobi rotations where each rotation tries to zero-out an off-diagonal element. It needs to perform  $n(n-1)/2$  rotations (in the case of a symmetric  $n \times n$  matrix) attempting to zero-out all the off-diagonal elements. These  $n(n-1)/2$  transformations constitute a sweep. It can be shown that after each sweep the norm of the off-diagonal elements decreases and hence the algorithm converges. It is believed that the number  $S$  of sweeps needed for convergence of the sequential Jacobi iteration algorithm is  $O(\log n)$  [3]. There are two variants of Jacobi based algorithm, namely, one-sided and two sided.

### 2.1 Two-Sided Jacobi SVD

The two-sided Jacobi iteration algorithm transforms a symmetric matrix  $A_{n \times n}$  into a diagonal matrix  $\Sigma_{n \times n}$  by a sequence of Jacobi rotations ( $J$ ), where each transform attempts to zero-out a given off-diagonal element of  $A_{n \times n}$ .

$$\Sigma_{n \times n} = (J_n^T \cdots (J_3^T (J_2^T (J_1^T A J_1) J_2) J_3) \cdots J_n^T) = (J_1 J_2 J_3 \cdots J_n)^T A (J_1 J_2 J_3 \cdots J_n) \tag{3}$$

The Jacobi rotation  $J(i, j, \theta)$  for an index pair  $(i, j)$  and a rotation angle  $\theta$  is a square matrix that is equal to the identity matrix  $I$  plus four additional entries at the intersections of rows and columns  $i$  and  $j$ :

$$J(i, j, \theta) = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{matrix} i \\ j \end{matrix}, \tag{4}$$

where  $c = \cos(\theta)$  and  $s = \sin(\theta)$ . It is clear that the Jacobi rotation is an orthogonal matrix (i.e.  $J(i, j, \theta)^T J(i, j, \theta) = I$ ) from the fact of  $\cos^2(\theta) + \sin^2(\theta) = 1$ . The  $c$  and  $s$  are computed as follows. Consider one of the transformations:  $B = J^T A J$ ,  $c$  and  $s$  are chosen such that the resultant matrix  $B$  is diagonal, i.e.,  $b_{ij} = b_{ji} = 0$ .

$$\begin{pmatrix} b_{ii} & b_{ij} \\ b_{ji} & b_{jj} \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \tag{5}$$

By solving this equation and taking the smaller root (see [3] for more detail),  $c$  and  $s$  are obtained by:

$$c = \frac{1}{\sqrt{1+t^2}} \quad \text{and} \quad s = t^*c, \tag{6}$$

where

$$t = \frac{\text{sign}(\tau)}{|\tau| + \sqrt{\tau^2 + 1}} \quad \text{and} \quad \tau = \frac{a_{ii} - a_{jj}}{2a_{ij}} \tag{7}$$

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

### 2.2 One-Sided Jacobi SVD

The two-sided Jacobi algorithms are computationally more expensive than the one-sided algorithms. Moreover, to implement the two-sided Jacobi method, it needs to traverse both row and column of the given matrix; however, matrices are stored either in row-major or column-major format. Thus, one of the two traversals will be less efficient on conventional memory architectures. In other words, one of the two traversals accesses the elements of the input  $m \times n$  matrix with unit stride, which is efficient; however, the other traversal performs stride  $n$  accesses, which is expensive because of cache miss handling time. On the other hand, the one-sided rotation modifies rows only, which is more suitable for memory hierarchy architectures. Thus to achieve efficient parallel SVD computation the best approach may be to adapt the Hestenes one-sided Jacobi transformation method [11] as advocated in [4, 12].

Hestenes one-sided Jacobi algorithm first produces a matrix  $B_{m \times n}$  whose rows are orthogonal by premultiplying  $A_{m \times n}$  with an orthogonal matrix  $U_{m \times m}$ :

$$U_{m \times n} A_{m \times n} = B_{m \times n}, \tag{8}$$

where rows of  $B_{m \times n}$  satisfy  $b_i^T b_j = 0$  for  $i \neq j$ . Followed by this  $B_{m \times n}$  is normalized by:

$$V_{n \times n} = S^{-1} B_{m \times n}, \tag{9}$$

where  $S_{n \times n} = \text{diag}(s_1, s_2, \dots, s_n)$ , and  $s_i = b_i^T b_i$ . It can be easily shown that  $A_{m \times n} = U_{m \times m}^T S_{m \times n} V_{n \times n}$ , which is equivalent to the definition of SVD.

One-sided Jacobi is also realized by a series of Jacobi rotations, but on one side. For a given  $i$  and  $j$ , rows  $i$  and  $j$  are orthogonalized by  $B_{m \times n} = J^T A_{m \times n}$  where  $J = J(i, j, \theta)$  is the same matrix as in the two-sided Jacobi (see equation 4) and:

$$\begin{pmatrix} b_i^T \\ b_j^T \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a_i^T \\ a_j^T \end{pmatrix}, \tag{10}$$

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

$$c = \frac{1}{\sqrt{1+t^2}} \quad \text{and} \quad s = t^*c, \tag{11}$$

where

$$t = \frac{\text{sign}(\tau)}{|\tau| + \sqrt{\tau^2 + 1}} \quad \text{and} \quad \tau = \frac{a_j^T a_j - a_i^T a_i}{2a_i^T a_j}. \tag{12}$$

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

### 2.3 One-Sided JRS Iteration Algorithm

Since any rotation in the one-sided Jacobi algorithm changes only the corresponding (two) rows, there exists inherent parallelism in the Jacobi iteration algorithms. For example, the  $n(n - 1)/2$  rotations in any sweep can be grouped into  $(n - 1)$  non-conflicting rotation sets each of which contains  $n/2$  rotations. The idea of the JRS iteration algorithms is to perform each set of rotations in parallel. One can think of the Jacobi algorithm as consisting of two phases. In the first phase all the rotation matrices are computed. In the second phase the rotation matrices are multiplied to get  $B$ , and then  $S$  and  $V$  are calculated. Consider any rotation operation. JRS performs all the  $n(n - 1)/2$  rotations of a sweep in parallel even though not all of these rotations are independent. Followed by this the second phase has to be completed. This involves the multiplication of  $O(n^2)$  rotation matrices.

The JRS iteration algorithm also has sweeps and in each sweep we perform rotations. The only difference is that in a given rotation, however, the norm of two rows does not zero-out but rather the value of this norm is decreased by a fraction of its current value. Given two column vectors  $u_i$  and  $u_j$ , the norm of them is reduced to a fraction of it. In the relaxation technique,  $c$  and  $s$  are chosen such that  $v_i^T v_j = \lambda u_i^T u_j$  instead of  $v_i^T v_j = 0$ .

From the following transformation

$$\begin{pmatrix} v_i^T \\ v_j^T \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} u_i^T \\ u_j^T \end{pmatrix} \tag{13}$$

it is easily to get

$$v_i^T = c u_i^T - s u_j^T \quad \text{and} \quad v_j^T = s u_i^T + c u_j^T, \tag{14}$$

then

$$v_i^T v_j = (c u_i - s u_j)^T (s u_i + c u_j) = u_i^T u_i (c^2 - s^2) + (u_i^T u_i - u_i^T u_j)cs = \lambda u_i^T u_j, \tag{15}$$

where  $\lambda$  is in the interval  $[0, 1)$ . When  $\lambda = 0$ , the JRS algorithm gives the same result of the original Jacobi iteration algorithm. The above equation can be solved for  $s$  and  $c$  as follows:

If  $u_i^T u_j = 0$ , then set  $c = 1$  and  $s = 0$ ;

Otherwise

$$\frac{u_i^T u_i - u_j^T u_j}{2u_i^T u_j} = \frac{c}{2s} - \frac{s}{2s} - \frac{\lambda}{2cs}. \tag{16}$$

Let

$$\tau = \frac{u_j^T u_j - u_i^T u_i}{2u_i^T u_j} \quad \text{and} \quad t = \frac{s}{c}, \tag{17}$$

then

$$(1 + \lambda)t^2 + 2\tau + \lambda - 1 = 0 \tag{18}$$

According to [3], the smaller root should be chosen, so

$$t = \frac{\text{sign}(\tau)(1 - \lambda)}{|\tau| + \sqrt{\tau^2 + (1 - \lambda^2)}} \tag{19}$$

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

$$c = \frac{1}{\sqrt{1+t^2}} \quad \text{and} \quad s = t^*c. \tag{20}$$

### 3 Block One-Sided JRS Iteration Algorithm

Any parallel algorithm for computing SVD on  $n \times n$  matrix partitions the  $n(n - 1)/2$  rotations of a sweep into rotation sets. Each rotation set consists of some number of independent rotations. All the rotations of a rotation set are performed in parallel

because they are independent. Most of the parallel SVD algorithms in the literature employ  $(n - 1)$  rotation sets, where each rotation set consists of  $n/2$  independent rotations. The streaming and JRS iteration algorithms are exceptions, where all the rotations can be calculated in parallel even though they are dependent.

For computers with memory hierarchy, it is often preferable to partition the input data and to perform the computation on the blocks. This approach provides for full reuse of data while the block is held in cache memory. It avoids excessive movement of data to and from memory and gives a surface-to-volume effect for the ratio of arithmetic operations to data movement, i.e.,  $O(n^3)$  arithmetic operations to  $O(n^2)$  data movement. In addition, on architectures that provide for parallel processing, parallelism can be exploited by performing operations on distinct blocks in parallel [13].

The proposed block JRS algorithm divides the rows of the input matrix  $A_{n \times n}$  into  $2P$  blocks, where  $P$  is the number of parallel processors. Not all the blocks should have the same size; the number of rows per block varies from  $\lfloor n/2P \rfloor$  to  $\lceil n/2P \rceil$ . The computations of the block JRS algorithm are done in two main parts. In the first part, each processor calculates all the rotations in the given block even though they are dependent and store them into 1-D arrays  $c[]$  and  $s[]$ . The length of these arrays is  $(n/2P)(n/2P-1)/2$  or  $(n^2/8P^2 - n/4P)$ , assuming that  $P$  divides  $n$ . (Each block is of the same size, namely,  $n/2P$  rows). For example if  $n/2P = 4$ , the first processor calculates the values of  $c[]$  and  $s[]$  for the following sequence of rows (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), and (3, 4), which are six rotations. After calculating the arrays for  $c[]$  and  $s[]$ , each processor applies these rotations to the corresponding block. The same is done for the second block (each processor works on the two blocks). This means that the total number of rotations in the first part of the proposed algorithm equals  $2P * (n^2/8P^2 - n/4P)$ , which equals  $(n^2/4P - n/2)$  rotations.

The second part of our proposed block JRS iteration algorithm calculates then applies the rotations between blocks, where each processor works on two blocks. This part iterates  $(2P - 1)$  steps. The well-known round-robin technique is used for generating  $(2P - 1)$  orders. Figure 1 shows an example of generating seven orders needed for executing the second part of the block JRS on four processors. In each iteration (step) of the second part, each processor computes arrays of rotation parameters ( $c[]$  and  $s[]$ ) for two blocks. Each row of the first block should be orthogonalized with all rows of the second block. A total of  $(n/2P)^2$  rotations are computed per processor working on two blocks. For example, assume that  $(n/2P) = 4$ . For a processor working on blocks 1 and 2, the following sequences are done to calculate  $c[1:16]$  and  $s[1:16]$ : (1, 5), (1, 6), (1, 7), (1, 8), (2, 5), (2, 6), (2, 7), (2, 8), (3, 5), (3, 6), (3, 7), (3, 8), (4, 5), (4, 6), (4, 7), and (4, 8), which are sixteen rotations. After calculating arrays of rotation parameters ( $c[1:(n/2P)^2]$  and  $s[1:(n/2P)^2]$ ), each processor applies  $(n/2P)^2$  rotations on its blocks. Then the round-robin routine is called for generating a new order, as shown in Figure 1. The number of rotations in the second part of the algorithm is  $P * (2P - 1) * (n/2P)^2$ , which equals  $n^2/2 - (n^2/4P)$  rotations. Thus the total number of iterations in the block JRS algorithm equals  $n^2/2 - n^2/4P + n^2/4P - n/2 = n(n - 1)/2$ .

Listing 1 depicts the proposed block JRS algorithm in detail. The first line calculates  $\delta$ , which is used as a termination parameter of the repeat-until loop. It equals the norm of the input matrix times the machine epsilon ( $\epsilon = 10^{-15}$ ). Line 2 of the proposed algorithm initializes up[] and dn[] arrays, which are used by the round-robin routine. Initially up[] array has even numbers (2, 4, 6, ...) and dn[] array has odd numbers (1, 3, 5, ...). These numbers indicate the block numbers associated with the processors at each step; processor  $i$  computes and applies the rotations of blocks up[i] and dn[i]. Figure 1 shows how the contents of the up[] and dn[] arrays are changed by calling the round-robin routine. Part one of the block JRS algorithm is depicted in lines 5 to 26, where the size of each block is stored in an array of data structure called SOB[]. SOB[i].low and SOB[i].high stores the number of the beginning row and last row of the block  $i$ . Besides, part two is depicted in lines 27 to 51. The last line computes the singular values form the orthogonal vectors.

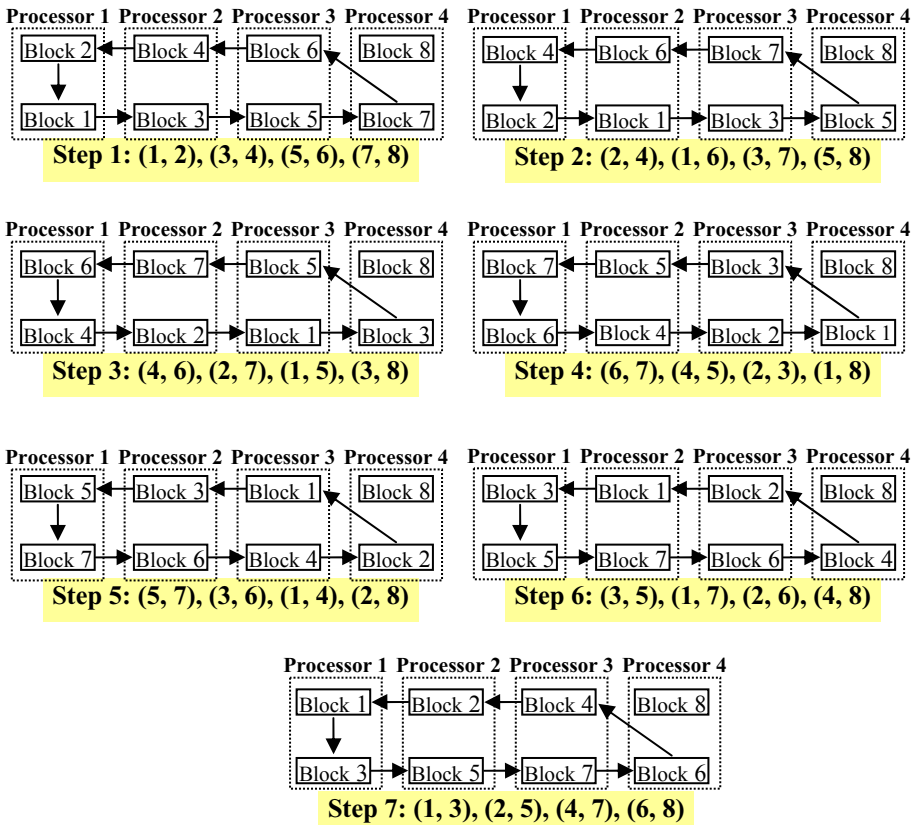


Fig. 1. Round-robin method for generating seven orders on four processors

**Listing 1: The proposed block JRS algorithm**

```

01.  $\delta = \epsilon \sum A[i]^T A[i], 1 \leq i \leq n$ 
02. for i = 1 to P { up[i]=2*i , dn[i]=2*i-1 }
03. repeat {
04.     converged = true
05.     for s = 1 to (2*P) {
06.         ind = 0
07.         for i = SOB[s].low to SOB[s].high-1 {
08.             for j = i+1 to SOB[s].high {
09.                 dii=A[i]^T A[i] , djj=A[j]^T A[j] , dij=A[i]^T A[j]
10.                 if |dij| >  $\delta$  then converged = false
11.                 if dij  $\neq$  0 then
12.                      $\tau = (djj - dii) / (2 * dij), t = \text{sgn}(\tau) / (|\tau| + \sqrt{\tau^2 + 1})$ 
13.                     c[ind]=1.0/sqrt(t2+1) , s[ind]=t*c[ind]
14.                 else c[ind] = 1 , s[ind] = 0
15.                 ind = ind + 1
16.             } }
17.         ind = 0
18.         for i = SOB[s].low to SOB[s].high-1 {
19.             for j = i+1 to SOB[s].high {
20.                 for k = 1 to n {
21.                     temp = c[ind] * A[i][k] - s[ind] * A[j][k]
22.                     A[j][k]= s[ind] * A[i][k] + c[ind] * A[j][k]
23.                     A[i][k] = temp
24.                 }
25.                 ind = ind + 1
26.             } } }
27.     for iteration = 1 to (2*P - 1) {
28.         for s = 1 to P {
29.             ind = 0
30.             for i = SOB[up[s]].low to SOB[up[s]].high {
31.                 for j = SOB[dn[s]].low to SOB[dn[s]].high {
32.                     dii=A[i]^T A[i] , djj=A[j]^T A[j] , dij=A[i]^T A[j]
33.                     if |dij| >  $\delta$  then converged = false
34.                     if dij  $\neq$  0 then
35.                          $\tau = (djj - dii) / (2 * dij), t = \text{sgn}(\tau) / (|\tau| + \sqrt{\tau^2 + 1})$ 
36.                         c[ind]=1.0/sqrt(t2+1) , s[ind]=t*c[ind]
37.                     else c[ind] = 1 , s[ind] = 0
38.                     ind = ind + 1
39.                 } } }
40.             ind = 0
41.             for i = SOB[up[s]].low to SOB[up[s]].high {
42.                 for j = SOB[dn[s]].low to SOB[dn[s]].high {
43.                     for k = 1 to n {
44.                         temp=c[ind]*A[i][k] - s[ind]*A[j][k]
45.                         A[j][k]=s[ind]*A[i][k] + c[ind]*A[j][k]
46.                         A[i][k]=temp
47.                     }
48.                     ind = ind + 1
49.                 } } }
50.             call round-robin(up[], dn[])
51.         } } }
52. } until converged = true
53. for i = 1 to n {  $\sigma[i] = \sqrt{A[i]^T A[i]}$  }

```

### 4 Experimental Results

The proposed block JRS algorithm has been implemented and tested on matrices with sizes varying from 50x50 up to 1000x1000 in steps of 50. The contents of these matrices were generated randomly to have a value in the interval [1, 10]. Besides, the number of processors varies from 4 to 20 in steps of 2. We use the norm of the input matrix times  $10^{-15}$  as the convergence parameter.

Figure 2 shows the number of sweeps needed by the block JRS algorithm. As we can see, the number of sweeps is close to that needed for the cyclic one-sided Jacobi

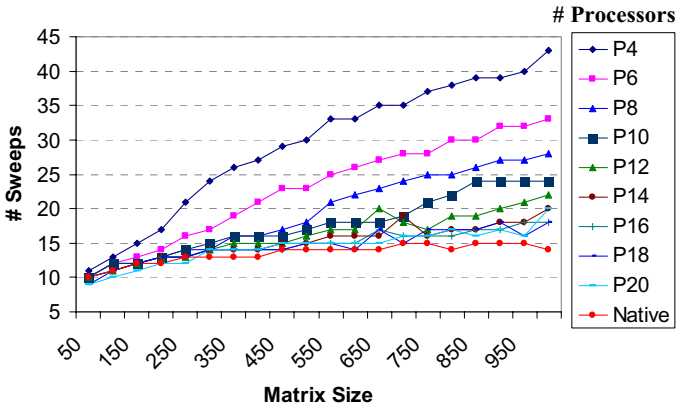


Fig. 2. The number of sweeps for the proposed algorithm

Table 1. The number of sweeps for the block JRS algorithm

Matrix Size	# Processors in Block JRS									
	Native	4	6	8	10	12	14	16	18	20
50	10	11	10	10	10	10	10	10	9	9
100	11	13	12	12	12	11	11	11	11	10
150	12	15	13	12	12	12	12	12	12	11
200	12	17	14	13	13	13	13	13	13	12
250	13	21	16	14	14	13	13	13	13	12
300	13	24	17	14	15	14	14	14	14	14
350	13	26	19	16	16	15	14	14	14	14
400	13	27	21	16	16	15	14	14	14	14
450	14	29	23	17	16	15	15	14	14	15
500	14	30	23	18	17	16	15	15	15	15
550	14	33	25	21	18	17	16	15	15	15
600	14	33	26	22	18	17	16	15	14	15
650	14	35	27	23	18	20	16	17	17	15
700	15	35	28	24	19	18	19	16	15	16
750	15	37	28	25	21	17	16	16	17	16
800	14	38	30	25	22	19	17	16	17	17
850	15	39	30	26	24	19	17	17	17	16
900	15	39	32	27	24	20	18	17	18	17
950	15	40	32	27	24	21	18	18	16	16
1000	14	43	33	28	24	22	20	18	18	20

algorithm when the number of processor is reasonably large. Besides, Table 1 shows the same result.

We have also studied the effect of the relaxation parameter  $\lambda$  on the performance of block JRS. The results are shown in Figure 3. When the number of processors is small the performance of the algorithm is very sensitive to the value of  $\lambda$ . A choice of 0.45 or more for  $\lambda$  seems to be the best the processor range [4:20] tried. When the number of processors is 10 or more even a small value of  $\lambda$  seems to yield good performance. One could identify an optimal value of  $\lambda$  empirically using the ideas given in [2].

Block JRS has also been tested for up to 250 processors and the numbers of sweeps for various values of the relaxation parameter  $\lambda$  are shown in Figure 4. The

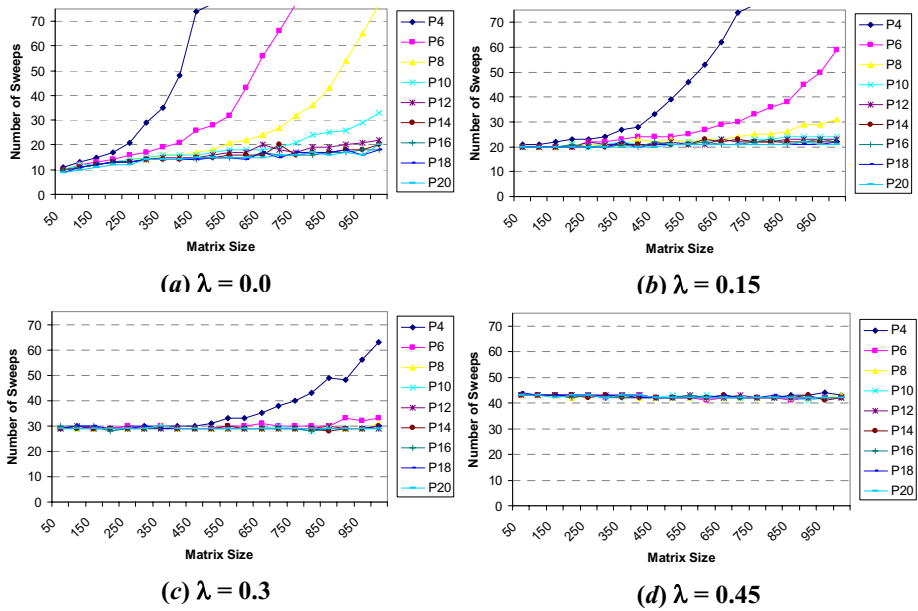


Fig. 3. The effect of  $\lambda$  on the number of sweeps

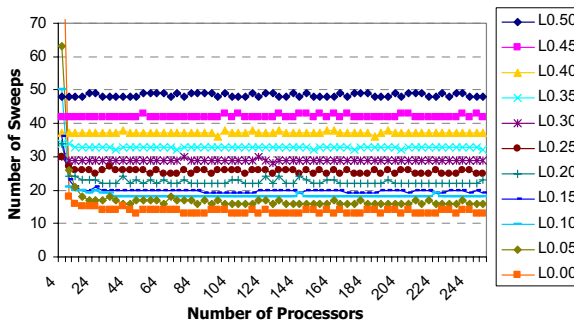


Fig. 4. The number of sweeps required for a matrix 500x500

matrix size used was 500×500. As we could see from this figure, when the number of processors is large, a value of zero or a small value for  $\lambda$  yields the best performance.

## 5 Conclusion

A new version of JRS iteration algorithm is proposed in this paper for the parallel computation of SVDs, which employs the same idea of decreasing (relaxing) the off-diagonal elements instead of zeroing them out. The block JRS algorithm is a highly parallel algorithm designed to exploit the memory hierarchy by processing blocks to reuse the loaded data into cache memories. It differs from the JRS algorithms in the partitioning strategy of the input matrix among processors. One of the key points of the block JRS is reusing the loaded data into cache memory by performing the computations on blocks. It performs  $O(b^2n)$  floating-point operations on  $O(bn)$  elements, which reuses the loaded data in cache memory by a factor on  $b$ . Another key point of the proposed algorithm is that on reasonably large number of processors, the number of sweeps is close to the cyclic Jacobi method even though not all the rotations in a block are independent. The relaxation technique helps to calculate and apply multiple rotations per block at the same time.

## References

1. Klema, V., Laub, A.: The Singular Value Decomposition: Its Computation and Some Applications. *IEEE Transactions on Automatic Control* AC-25(2) (1980)
2. Rajasekaran, S., Song, M.: A Novel Scheme for the Parallel Computation of SVDs. In: *Proc. High Performance Computing and Communications*, pp. 129–137 (2006)
3. Golub, G., Van Loan, C.: *Matrix Computations*, 2nd edn. John Hopkins University Press, Baltimore and London (1993)
4. Brent, R., Luk, F.: The Solution of Singular-Value and Symmetric Eigenvalue Problems on Multiprocessor Arrays. *SIAM Journal on Scientific and Statistical Computing* 6(1), 69–84 (1985)
5. Zhou, B., Brent, R.: A Parallel Ring Ordering Algorithm for Efficient One-sided SVD Computations. *Journal of Parallel and Distributed Computing* 42, 1–10 (1997)
6. Becka, M., Vajtersic, M.: Block-Jacobi SVD Algorithms for Distributed Memory Systems I: Hypercubes and Rings. *Parallel Algorithms Appl.* 13, 265–287 (1999)
7. Becka, M., Vajtersic, M.: Block-Jacobi SVD Algorithms for Distributed Memory Systems II: Meshes. *Parallel Algorithms Appl.* 14, 37–56 (1999)
8. Becka, M., Oksa, G., Vajtersic, M.: Dynamic Ordering for a Parallel Block-Jacobi SVD Algorithm. *Parallel Comp.* 28, 243–262 (2002)
9. Oksa, G., Vajtersic, M.: A Systolic Block-Jacobi SVD Solver for Processor Meshes. *Parallel Algorithms and Applications* 18, 49–70 (2003)
10. Strumpfen, V., Hoffmann, H., Agarwal, A.: A Stream Algorithm for the SVD. Technical Memo 641, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology (2003)
11. Hestenes, M.: Inversion of Matrices by Biorthogonalization and Related Results. *J. SIAP* 6(1), 51–90 (1958)
12. Brent, R.: *Parallel Algorithms for Digital Signal Processing*. *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, pp. 93–110. Springer, Heidelberg (1991)
13. Dongarra, J., Foster, I., Fox, G., Kennedy, K., White, A., Torczon, L., Gropp, W.: *The Sourcebook of Parallel Computing*. Morgan Kaufmann, San Francisco (2002)