European Society of Computational Methods in Sciences and Engineering (ESCMSE)

Journal of Numerical Analysis, Industrial and Applied Mathematics (JNAIAM)
vol. 2, no. 1-2, 2007, pp. 49-66
ISSN 1790–8140

Parallelizing the Kogbetliantz Method: A First Attempt [1]

Vjeran Hari[2]

Department of Mathematics, University of Zagreb,
Bijenička cesta 30, 10002 Zagreb, Croatia

Vida Zadelj-Martić[3]

Faculty of Geodesy, University of Zagreb,
A. Kačića-Miošića 26, 10000 Zagreb, Croatia

Received 21 December, 2005; accepted in revised form 15 February, 2007

Abstract: The paper investigates a way how can the two-sided Jacobi-type method for computing the singular value decomposition of triangular matrices, known as Kogbetliantz method, be adapted for use with parallel computers with shared memory. The slower row operations can be replaced, at low extra cost, by the faster column operations. It is shown how can the method be further modified to work with blocks. In any case, the initial triangular or rectangular matrix has to be brought to a special, butterfly-like form. In the iterative part of the algorithm, this special form gradually changes, but after a fixed number of parallel steps, which corresponds to two standard sweeps, the initial butterfly-like form is retained. This property simplifies the algorithm and enhances its performance.

Keywords: singular value decomposition, Kogbetliantz method, parallel algorithm, block algorithm

Mathematics Subject Classification: 65F16, 68W10

1 Introduction

Recently, Drmač and Veselić [8], [9] have modified the one-sided Jacobi method for computing the singular value decomposition (SVD) of general matrices. Their modification is faster than the QR method and almost as fast as the divide and conquer (DC) method. Since the Jacobi method is relatively accurate for well-behaved matrices [5], which cannot be said for the QR and DC methods, the diagonalization-type methods have drawn new attention of researchers in the field of matrix eigenvalue computation.

Several key ideas from [8], [9] can be used in connection with the Kogbetliantz method for computing the SVD of triangular matrices [23], [24], [15]. Typically, the preprocessing which uses one or two QR factorizations (see [9]), can be used with the Kogbetliantz method. Computation of the right or left singular vectors a posteriori, using the starting triangular matrix and the data computed by the method (see [7], [9]), can be made with the Kogbetliantz method in the same way. We note that the Kogbetliantz method is relatively accurate (see [21] and [25]). Therefore, it is in many respects similar to the two-sided Jacobi

[1] Published electronically April 14, 2007
[2] E-mail: hari@math.hr
[3] E-mail: vzadelj@geodet.geof.hr

method for positive definite matrices. But, when compared with the one-sided Jacobi method, this also means that there exists a serious weakness of the Kogbetliantz method.

One-sided methods are fast because they operate only on columns. BLAS 1 operations are several times faster when applied to the columns than to the rows. And the Kogbetliantz method rotates both, the columns and the rows of the current matrix. Although, this is a severe drawback, the two-sided methods have also some advantages over the one-sided methods. In particular, they can easily terminate the process, without any extra cost. It is known that the one-sided method has to pay each check for convergence, with computing $\approx n^2/2$ dot products. The Kogbetliantz method shows its full power on almost diagonal triangular matrices. In contrast to the one-sided Jacobi methods, it further diagonalizes the current matrix and, in some way, preserves its triangular form.

Another advantage of the one-sided methods lies in their inherent parallelism, which makes them amenable for distributed memory computing. Here, we show that Kogbetliantz method can be adapted for parallel computing with shared memory. It is interesting, that our approach considerably eliminates the slowdown coming from the row operations.

We finally note that the one-sided Jacobi methods can easily be turned into BLAS 3 algorithms [20]. This means working with block-columns instead of columns alone. Here we show that the Kogbetliantz method can in a similar way be transformed into a BLAS 3 algorithm.

The rest of the paper is divided into four sections. In Section 2, we introduce the butterfly form and briefly describe how to reduce a triangular or a rectangular matrix to this form. In Section 3, we describe the parallel strategy and the parallel Kogbetliantz method. In Section 4, we show how to modify the method to become a parallel BLAS 3 algorithm. Finally, in Section 5, we summarize the main results of the paper.

2 Reduction to Butterfly Form

In this section we introduce the butterfly form and show how to reduce a general rectangular matrix to this form, using the standard tools such as Householder reflectors and Givens rotations. Since our aim is to construct a fast SVD algorithm, we shall need the same preprocessing (which additionally serves as preconditioning) as the one used in [9]. This means applying the QR factorization with column pivoting (QRP), followed possibly by the LQ factorization. Afterwards, R or L has to be brought to the butterfly form. Therefore, we also show how to obtain the butterfly form from a matrix in the triangular form.

2.1 The butterfly form

For $n = 6$ and $n = 7$, the butterfly form of a square matrix B of order n, has the following appearance

$$B = \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & x \\ x & x & x & 0 & x & x \\ x & x & x & x & x & x \\ x & x & 0 & 0 & x & x \\ x & 0 & 0 & 0 & 0 & x \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & 0 & x \\ x & x & x & 0 & 0 & x & x \\ x & x & x & x & x & x & x \\ x & x & x & 0 & x & x & x \\ x & x & 0 & 0 & 0 & x & x \\ x & 0 & 0 & 0 & 0 & 0 & x \end{bmatrix},$$

respectively. Here x denotes an element of B which need not be zero.

On conventional computers it is easy to reduce any rectangular matrix A to butterfly form. The algorithm is similar to the one for computing the QR factorization. Let A be $m \times n$, $m \geq n$. One can use a finite sequence of Householder reflectors or an appropriate sequence of Givens rotations, applying them from the left-hand side. This transformation will preserve the relative accuracy of the singular values and vectors in the presence of rounding errors, provided that A is well-behaved.

We shall denote by H_i and H_i' the Householder matrices (the direct sum of the appropriate identity matrices and the Householder reflectors) which are used at step i of the reduction process. The algorithm

has the following form

$$B = \begin{cases} H'_k H_k \cdots H'_2 H_2 \, H'_1 H_1 \, A, & k = n/2, \;\; n \text{ even} \\ B = H'_k H_k \cdots H'_2 H_2 \, H_1 \, A \,, & k = (n+1)/2, \;\; n \text{ odd.} \end{cases}$$

Here B is the matrix in butterfly form. Each matrix H_i (and H'_i) makes an appropriate number of zeros in the current column. The procedure is illustrated for $m = 7$, $n = 6\,, 5$. The symbols $*$ and $\star$ denote the elements which will be annihilated by H_i and H'_i, respectively, in the ith step of the algorithm

$$\begin{bmatrix} x & x & \star & * & x & x \\ x & x & \star & * & x & x \\ x & x & x & * & x & x \\ x & x & x & x & x & x \\ x & x & \star & * & x & x \\ x & x & \star & * & x & x \\ x & x & \star & * & x & x \end{bmatrix} \to \begin{bmatrix} x & \star & 0 & 0 & * & x \\ x & x & 0 & 0 & * & x \\ x & x & x & 0 & x & x \\ x & x & x & x & x & x \\ x & x & 0 & 0 & x & x \\ x & \star & 0 & 0 & * & x \\ x & \star & 0 & 0 & * & x \end{bmatrix} \to \begin{bmatrix} x & 0 & 0 & 0 & 0 & * \\ x & x & 0 & 0 & 0 & x \\ x & x & x & 0 & x & x \\ x & x & x & x & x & x \\ x & x & 0 & 0 & x & x \\ x & 0 & 0 & 0 & 0 & x \\ \star & 0 & 0 & 0 & 0 & * \end{bmatrix} \to \left[\begin{array}{cccccc} x & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & x \\ x & x & x & 0 & x & x \\ x & x & x & x & x & x \\ x & x & 0 & 0 & x & x \\ x & 0 & 0 & 0 & 0 & x \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

$$\begin{bmatrix} x & x & * & x & x \\ x & x & * & x & x \\ x & x & x & x & x \\ x & x & * & x & x \\ x & x & * & x & x \\ x & x & * & x & x \\ x & x & * & x & x \end{bmatrix} \to \begin{bmatrix} x & \star & 0 & * & x \\ x & x & 0 & * & x \\ x & x & x & x & x \\ x & x & 0 & x & x \\ x & \star & 0 & * & x \\ x & \star & 0 & * & x \\ x & \star & 0 & * & x \end{bmatrix} \to \begin{bmatrix} x & 0 & 0 & 0 & * \\ x & x & 0 & 0 & x \\ x & x & x & x & x \\ x & x & 0 & x & x \\ x & 0 & 0 & 0 & x \\ \star & 0 & 0 & 0 & * \\ \star & 0 & 0 & 0 & * \end{bmatrix} \to \left[\begin{array}{ccccc} x & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & x \\ x & x & x & x & x \\ x & x & 0 & x & x \\ x & 0 & 0 & 0 & x \\ \hline 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] .$$

The butterfly form is square, so we have separated it from the remaining zero rows by the horizontal line. If A has more columns than rows, we can work with A^T, which is a standard technique in the SVD computation. Column pivoting can be implemented as well.

We shall skip an attempt to parallelize this part of the algorithm for two reasons. First, such a parallel algorithm has to be similar to some parallel QR factorization algorithm for the given multiprocessor machine. So, it surely can be implemented. Second, and more important, the initial part of the algorithm has to precondition the initial matrix for the iterative part of the SVD computation. And this can be accomplished, as shown in [9], by applying a parallel QRP algorithm, followed possibly by a parallel LQ algorithm. So, we can assume that efficient implementations of such algorithms are at disposal for the given parallel machine. Hence it remains to show how to bring the triangular matrix R or L, obtained after this pre-processing, to the butterfly form.

Let T be the triangular matrix of order n (obtained by the QRP or by the QRP followed by the LQ factorization). We can assume that T is upper-triangular. Otherwise, we can work either with the transpose of T or we can construct a similar algorithm for the lower-triangular T. We shall show that T can be reduced to the butterfly form using a cheap and exact similarity transformation by a permutation matrix: $B = P^T T P$. The permutation matrix P is constructed as product of transposition matrices as follows

$$P = \begin{cases} I_{12} I_{13} (I_{14} I_{23}) (I_{15} I_{24}) (I_{16} I_{25} I_{34}) \cdot (I_{1,n} I_{2,n-1} \cdots I_{k,k+1}) & \text{if } n = 2k \\ I_{12} I_{13} (I_{14} I_{23}) (I_{15} I_{24}) (I_{16} I_{25} I_{34}) \cdot (I_{1,n} I_{2,n-1} \cdots I_{k,k+2}) & \text{if } n = 2k+1 \,. \end{cases} \tag{1}$$

Here, $I_{pq} = [e_1, \ldots, e_q, \ldots, e_p, \ldots, e_n]$ where e_j, $1 \le j \le n$ are the columns of the identity matrix I_n. The parentheses are used to indicate the transpositions which can be applied in parallel. For example, for

$n = 6$, we have $k = n/2 = 3$, and the procedure takes the form

$$\begin{bmatrix} x & * & x & x & x & x \\ 0 & x & x & x & x & x \\ 0 & 0 & x & x & x & x \\ 0 & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & x \end{bmatrix} \rightarrow \begin{bmatrix} x & 0 & * & x & x & x \\ x & x & x & x & x & x \\ 0 & 0 & x & x & x & x \\ 0 & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & x \end{bmatrix} \rightarrow \begin{bmatrix} x & 0 & 0 & * & x & x \\ x & x & * & x & x & x \\ x & 0 & x & x & x & x \\ 0 & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & x \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} x & 0 & 0 & 0 & * & x \\ x & x & 0 & * & x & x \\ x & x & x & x & x & x \\ x & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & x \end{bmatrix} \rightarrow \begin{bmatrix} x & 0 & 0 & 0 & 0 & * \\ x & x & 0 & 0 & * & x \\ x & x & x & * & x & x \\ x & x & 0 & x & x & x \\ x & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & x \end{bmatrix} \rightarrow \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & x \\ x & x & x & 0 & x & x \\ x & x & x & x & x & x \\ x & x & 0 & 0 & x & x \\ x & 0 & 0 & 0 & 0 & x \end{bmatrix}$$

Here each $*$ denotes the position of one pivot element. The subscripts of each pivot element determine the rows and columns which are to be swapped. For example, in the 5th displayed matrix, the positions of pivot elements are $(1, 6)$, $(2, 5)$, $(3, 4)$. So, the rows (as well as the columns) 1 and 6, 2 and 5, 3 and 4 will be swapped. We see that all these transformations can be performed simultaneously if three processors are available. From the formula above, we conclude that $n - 1$ "parallel steps" are needed provided that $[n/2]$ processors are available. Here $[z]$ is the largest integer $\leq z$.

To trace the movement of the non-zero elements, one can use the following code, which is the essential part of the FORTRAN 77 subroutine for this reduction. Double precision is used.

```
      DO 1 K=2,N
C                 SWAPPING COLUMNS
        DO 2 I=1,K/2
         J=K+1-I
         CALL DSWAP(N,A(1,I),1,A(1,J),1)
   2    CONTINUE
C                   SWAPPING ROWS
        DO 3 I=1,K/2
         J=K+1-I
         CALL DSWAP(N,A(I,1),LDA,A(J,1),LDA)
   3    CONTINUE
 1     CONTINUE
```

Here, `N` is order of the two-dimensional array `A`, `LDA` is the leading dimension of `A` and `SSWAP` is the BLAS 1 routine which performs a swap. We can notice that swapping columns requires unit stride, while swapping rows requires stride `LDA`. Note that the inner loops allow for parallel processing.

The effect of this transformation can be seen in the following example

$$\begin{bmatrix} 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 0 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\ 0 & 0 & 33 & 34 & 35 & 36 & 37 & 38 \\ 0 & 0 & 0 & 44 & 45 & 46 & 47 & 48 \\ 0 & 0 & 0 & 0 & 55 & 56 & 57 & 58 \\ 0 & 0 & 0 & 0 & 0 & 66 & 67 & 68 \\ 0 & 0 & 0 & 0 & 0 & 0 & 77 & 78 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 88 \end{bmatrix} \longrightarrow \begin{bmatrix} 88 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 68 & 66 & 0 & 0 & 0 & 0 & 0 & 67 \\ 48 & 46 & 44 & 0 & 0 & 0 & 45 & 47 \\ 28 & 26 & 24 & 22 & 0 & 23 & 25 & 27 \\ 18 & 16 & 14 & 12 & 11 & 13 & 15 & 17 \\ 38 & 36 & 34 & 0 & 0 & 33 & 35 & 37 \\ 58 & 56 & 0 & 0 & 0 & 0 & 55 & 57 \\ 78 & 0 & 0 & 0 & 0 & 0 & 0 & 77 \end{bmatrix}.$$

3 The Kogbetliantz Method under the Modulus Strategy

The Kogbetliantz method is a known two-sided Jacobi-type method for computing the SVD of a square matrix. Since its discovery (see [23, 24]), the method has been studied with respect to its global and quadratic convergence (see [12], [14], [31], [3]) and implementation details [4]. Later, it has been discovered in [15] and [4] that the method becomes simpler and more efficient if it is applied to the triangular matrices under the serial (i.e. the row- and the column-cyclic) pivot strategies. Then it behaves quite similar to the standard Jacobi method for symmetric matrices. The global

convergence properties of this method have been studied in [15], [10] and the quadratic convergence in [17], [29], [18]. The relative accuracy of the low-level part of the algorithm (the angle formulas and the updates of the diagonal elements) has recently been proved in [21]. Hence, combining that with the technique from [25], yields the relative accuracy of the Kogbetliantz algorithm.

The method is typically applied to the triangular matrices obtained by the QR factorization with column pivoting. It preserves the non-negativity (and the positivity) of the diagonal elements. So, if Φ is an appropriate diagonal matrix of signs, it is advantageous to work with ΦR, where $\mathrm{diag}(\Phi R) \geq 0$. This choice further simplifies the algorithm.

Here, we show that all these nice properties of the Kogbetliantz method are preserved if the method is applied to a matrix in butterfly form. However, the method has to be defined by a special pivot strategy known as the *modulus strategy*.

In addition, the method becomes amenable for parallel computing. We shall refer to it as the Kogbetliantz method under the modulus strategy or shorter KMMS. The first approach to parallelizing the Kogbetliantz method is due to Luk [26]. We also mention here the papers [2], [28], [13], [30] and [1].

Starting from the matrix B in butterfly form, KMMS generates the sequence of matrices $B^{(k)} = (b^{(k)}_{lm})$, $k \geq 0$, by the rule

$$B^{(k+1)} = (U^{(k)})^T B^{(k)} V^{(k)}, \quad k \geq 0; \quad B^{(0)} = B\,.$$

Here $U^{(k)} = U^{(k)}_{ij}$ and $V^{(k)} = V^{(k)}_{ij}$, are simple rotations in the (i,j)-plane, where $i = i(k)$, $j = j(k)$, $i < j$. Recall that i and j are the *pivot indices* and (i,j) is the *pivot pair*. If $b^{(k)}_{ij} \neq 0$ or $b^{(k)}_{ji} \neq 0$, the rotation angles $\varphi^{(k)}$, $\psi^{(k)}$ of $U^{(k)}$, $V^{(k)}$, respectively, are determined from the requirement

$$b^{(k+1)}_{ij} = b^{(k+1)}_{ji} = 0, \quad \text{which implies} \quad \|\Omega(B^{(k+1)})\|^2 = \|\Omega(B^{(k)})\|^2 - |b^{(k)}_{ij}|^2 - |b^{(k)}_{ji}|^2.$$

Here, $\Omega(X) = X - \mathrm{diag}(X)$ denotes the off-diagonal part of the square matrix X and $\|X\|$ denotes the Frobenius norm of X. In the following subsections, we show how to compute the essential elements of $U^{(k)}$ and $V^{(k)}$, and describe the modulus strategy. We also show how the initial butterfly-like zero pattern changes with iterations. After that, we briefly describe the sequential and the parallel algorithm.

3.1 An accurate SVD algorithm for 2×2 triangular matrices

We start with the case of an upper-triangular 2×2 matrix T whose non-zero elements are denoted by f, g and h. A single Kogbetliantz step which diagonalizes T reduces to the orthogonal transformation

$$T' = \begin{bmatrix} f' & 0 \\ 0 & h' \end{bmatrix} = \begin{bmatrix} c_\varphi & s_\varphi \\ -s_\varphi & c_\varphi \end{bmatrix} \begin{bmatrix} f & g \\ 0 & h \end{bmatrix} \begin{bmatrix} c_\psi & -s_\psi \\ s_\psi & c_\psi \end{bmatrix} = U^T T V. \tag{2}$$

Here U and V are plane rotations, c_φ, s_φ, c_ψ and s_ψ denote $\cos(\varphi)$, $\sin(\varphi)$, $\cos(\psi)$ and $\sin(\psi)$, respectively. We shall also use t_φ and t_ψ for $\tan(\varphi)$ and $\tan(\psi)$, respectively. It can be easily deduced (see [21]) that

$$f' = \frac{c_\varphi}{c_\psi} f = \frac{s_\psi}{s_\varphi} h = \frac{c_\psi}{c_\varphi} f + \frac{s_\psi}{c_\varphi} g = \frac{s_\varphi}{s_\psi} h + \frac{c_\varphi}{s_\psi} g, \tag{3}$$

$$h' = \frac{c_\psi}{c_\varphi} h = \frac{s_\varphi}{s_\psi} f = \frac{c_\varphi}{c_\psi} h - \frac{s_\varphi}{c_\psi} g = \frac{s_\psi}{s_\varphi} f - \frac{c_\psi}{s_\varphi} g. \tag{4}$$

There are several formulas for computing c_φ, s_φ, c_ψ and s_ψ. If the right-hand transformation V is first applied to T, we obtain (using the notation from [16] and [21]) the UR (Upper-triangular, Right transformation first) algorithm. If the left-hand transformation is first applied, we obtain the UL algorithm.

UL Algorithm

$$\tan 2\varphi = \frac{2gh}{f^2 + g^2 - h^2}\,.$$

$$\tan\psi = \frac{g + h t_\varphi}{f} = \frac{f t_\varphi}{h - g t_\varphi}\,.$$

UR Algorithm

$$\tan 2\psi = \frac{2fg}{f^2 - g^2 - h^2}\,.$$

$$\tan\varphi = \frac{f t_\psi - g}{h} = \frac{h t_\psi}{f + g t_\psi}\,.$$

The above formulas are used for computing c_ψ, s_ψ, c_φ, s_φ. The formulas (3) and (4) hold with both, the UR and UL algorithms.

If T is lower-triangular, a single Kogbetliantz step takes the form

$$T' = \begin{bmatrix} f' & 0 \\ 0 & h' \end{bmatrix} = \begin{bmatrix} c_\varphi & s_\varphi \\ -s_\varphi & c_\varphi \end{bmatrix} \begin{bmatrix} f & 0 \\ g & h \end{bmatrix} \begin{bmatrix} c_\psi & -s_\psi \\ s_\psi & c_\psi \end{bmatrix} = U^T T V.$$

Transposing this equation, one can invoke the above formulas and obtain (see [16] and [21])

LL Algorithm

$$\tan 2\varphi = \frac{2fg}{f^2 - g^2 - h^2}\,,$$

$$\tan\psi = \frac{ft_\varphi - g}{h} = \frac{ht_\varphi}{f + gt_\varphi}\,,$$

LR Algorithm:

$$\tan 2\psi = \frac{2gh}{f^2 + g^2 - h^2}\,,$$

$$\tan\varphi = \frac{g + ht_\psi}{f} = \frac{ft_\psi}{h - gt_\psi}\,.$$

The formulas (3) and (4) are "translated" into

$$f' = \frac{c_\psi}{c_\varphi} f = \frac{s_\varphi}{s_\psi} h = \frac{c_\varphi}{c_\psi} f + \frac{s_\varphi}{c_\psi} g = \frac{s_\psi}{s_\varphi} h + \frac{c_\psi}{s_\varphi} g\,, \tag{5}$$

$$h' = \frac{c_\varphi}{c_\psi} h = \frac{s_\psi}{s_\varphi} f = \frac{c_\psi}{c_\varphi} h - \frac{s_\psi}{c_\varphi} g = \frac{s_\varphi}{s_\psi} f - \frac{c_\varphi}{s_\psi} g\,. \tag{6}$$

Note that formulas (3)–(6) imply that the non-negativity (positivity) of f and g implies the non-negativity (positivity) of f' and g'.

As is shown in [21], it is always possible to chose between UR and UL (LR and LL) algorithms to achieve that c_ψ, s_ψ, c_φ, s_φ, f' and h' are computed with small relative error. This, together with the technique from [25] is the basis of the accuracy proof for the serial and for the modulus Kogbetliantz method.

Instead of the algorithm described above, one can use the `*LAESV2` routine from LAPACK. However, the error analysis from [21] reveals that the algorithm based on the above formulas might be more appropriate (than `*LAESV2`) to be used as part of the Kogbetliantz method.

3.2 The modulus pivot strategy

This pivot strategy was first described in [15] and the name was first mentioned in [28]. For simplicity, we can regard it as a special cyclic (pivot) strategy. The *modulus ordering* of the set $\mathbf{P}_n = \{\,(p,q) \,:\, 1 \le p < q \le n\,\}$ which defines the strategy, is depicted using the upper-triangular part of the matrix of order $n = 7$. The displayed numbers in the leftmost matrix represent the ordering in which the pivot elements are annihilated within one cycle. The position of each number in the matrix determines the position of the appropriate pivot element.

By $\mathcal{S}_t$, $1 \le t \le 7$ we denote the so called *rotation sets*, which are the sets of index pairs of those pivot elements which can be simultaneously annihilated within the "parallel step". This means that, using the three processors, p_1, p_2, p_3, the matrix elements b_{ij}, b_{ii}, b_{jj}, $(i,j) \in \mathcal{S}_t$ can be fetched simultaneously. Hence, the rotation matrices $U_{ij}^{(k)}$, $V_{ij}^{(k)}$, $(i,j) \in \mathcal{S}_t$ can be computed, simultaneously.

In the rightmost matrix, we have used the same number to indicate the pivot positions of the elements that are simultaneously annihilated. In other words, these numbers denote the schedule of the parallel steps.

$$\begin{bmatrix} \cdot & 6 & 9 & 11 & 14 & 16 & 19 \\ & \cdot & 12 & 15 & 17 & 20 & 1 \\ & & \cdot & 18 & 21 & 2 & 4 \\ & & & \cdot & 3 & 5 & 7 \\ & & & & \cdot & 8 & 10 \\ & & & & & \cdot & 13 \\ & & & & & & \cdot \end{bmatrix} \qquad \begin{aligned} \mathcal{S}_1 &= \{(2,7),(3,6),(4,5)\} \\ \mathcal{S}_2 &= \{(3,7),(4,6),(1,2)\} \\ \mathcal{S}_3 &= \{(4,7),(5,6),(1,3)\} \\ \mathcal{S}_4 &= \{(5,7),(1,4),(2,3)\} \\ \mathcal{S}_5 &= \{(6,7),(1,5),(2,4)\} \\ \mathcal{S}_6 &= \{(1,6),(2,5),(3,4)\} \\ \mathcal{S}_7 &= \{(1,7),(2,6),(3,5)\} \end{aligned} \qquad \begin{bmatrix} \cdot & 2 & 3 & 4 & 5 & 6 & 7 \\ & \cdot & 4 & 5 & 6 & 7 & 1 \\ & & \cdot & 6 & 7 & 1 & 2 \\ & & & \cdot & 1 & 2 & 3 \\ & & & & \cdot & 3 & 4 \\ & & & & & \cdot & 5 \\ & & & & & & \cdot \end{bmatrix}$$

For example, in the third parallel step the following computation has to be done. First, the elements of the rotation matrices which annihilate the elements at positions $(4,7)$, $(5,6)$ and $(1,3)$ are computed simultaneously. Then all three processors simultaneously apply the right transformations. Finally, all processors simultaneously apply the left transformations. If the left and the right singular vectors are wanted, then the right rotations can be accumulated while the left singular vectors can easily be computed a posteriori, like in [9]. Many ideas and implementation details from [9] can be used here.

3.3 The general parallel algorithm

In order to derive an efficient modulus Kogbetliantz algorithm for general n, we first make a partition of the set $\mathbf{P}_n$ into n subsets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$, where each subset has around $n/2$ elements. These are the rotation sets. Each one of them has the property that all pairs contained in it are mutually *disjoint*. Two pairs (i, j) and (p, q) from $\mathbf{P}_n$ are disjoint or *commuting* if the corresponding sets $\{i, j\}$ and $\{p, q\}$ are disjoint. In the context of parallel Jacobi-type methods, the rotation sets take over the role of the pivot pairs from the sequential Jacobi-type methods.

At parallel step t (t can be considered as time step), the *pivot set* $Piv(t)$ determines the positions of those matrix elements which are to be rotated. Consider the following sequence of rotation sets:

$$\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n, \mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n, \mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n, \mathcal{S}_1, \ldots \tag{7}$$

In the first (parallel) step $Piv(t)$ coincides with $\mathcal{S}_1$, in the second step it coincides with $\mathcal{S}_2$ and so on. At step n it coincides with $\mathcal{S}_n$, at step $(n+1)$ it coincides with $\mathcal{S}_1$, at step $(n+2)$ with $\mathcal{S}_2$ and so on. Thus, $Piv(t)$ runs through the sequence of rotation sets (7) in the cyclic way.

We set $B^{[1]} = B$.

At step t, $t \geq 1$, the rotation matrices $U_{ij}^{[t]}$, $V_{ij}^{[t]}$, $(i, j) \in Piv(t)$ are computed using the elements of the same matrix $B^{[t]}$. Then the transformation

$$B^{[t+1]} = {U^{[t]}}^T B^{[t]} V^{[t]}, \qquad U^{[t]} = \prod_{(i,j)\in Piv(t)} U_{ij}^{[t]}, \quad V^{[t]} = \prod_{(i,j)\in Piv(t)} V_{ij}^{[t]}, \quad t \geq 1, \tag{8}$$

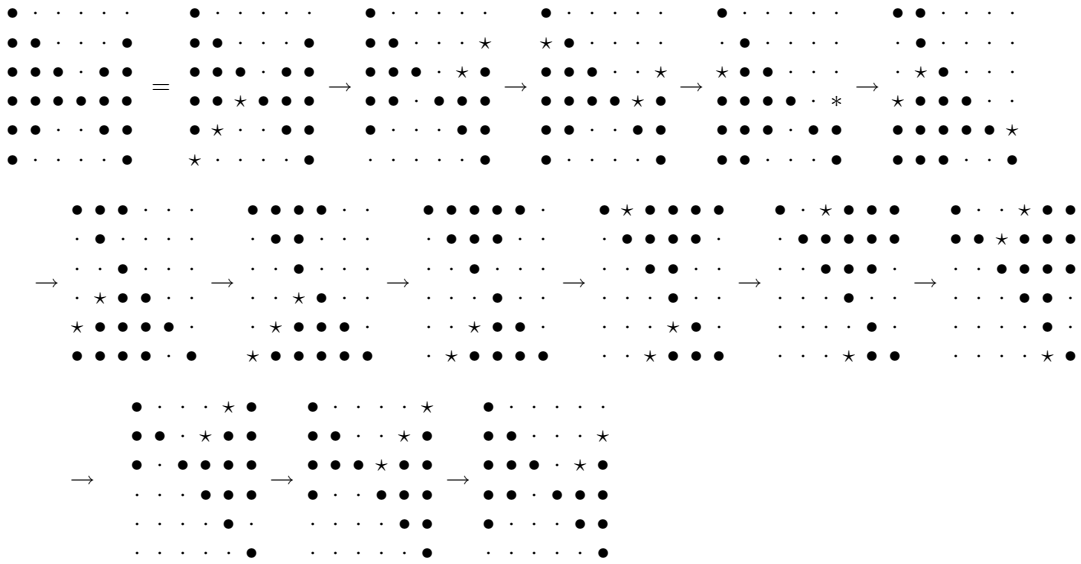
is performed. Here, $U^{[t]}$ and $V^{[t]}$ are not explicitly computed. Instead, all the factors $V_{ij}^{[t]}$, $(i, j) \in Piv(t)$ are simultaneously applied to $B^{[t]}$, and afterwards the same is done with all $U_{ij}^{[t]}$, $(i, j) \in Piv(t)$.

Note that in the above relation the matrices $U^{[t]}$ are well defined, because all rotation matrices $U_{ij}^{[t]}$, $(i, j) \in Piv(t)$ mutually commute (the order of the factors is not relevant). The same is true for $V^{[t]}$ which is defined by $V_{ij}^{[t]}$, $(i, j) \in Piv(t)$. It is irrelevant whether the transformation $B^{[t]} \mapsto {U^{[t]}}^* B^{[t]}$ or $B^{[t]} \mapsto B^{[t]} V^{[t]}$ in (8) is performed first.

At step t, the following tasks are performed: computation of the angle functions, application of the right transformation and application of the left transformation. If the right and left singular vector matrices of B are wanted, then the right transformations $V^{[t]}$ can be accumulated into V. After that, U can safely be computed a posteriori, from the equation $BV = U\Sigma$. Alternatively, one can accumulate $U^{[t]}$ into U and compute V a posteriori.

3.4 Using the butterfly form

If B is in butterfly form, then it is permutationally similar to an upper-triangular matrix. We briefly say that it is PST. Therefore, it is also essentially triangular (briefly, ET) since $b_{pq} b_{qp} = 0$ holds for all $p < q$. Butterfly form and modulus strategy go together since all the matrices, generated by the algorithm from B, are PST. Indeed, the following figure depicts (for $n = 6$) what happens with the zero structure during two consecutive cycles:

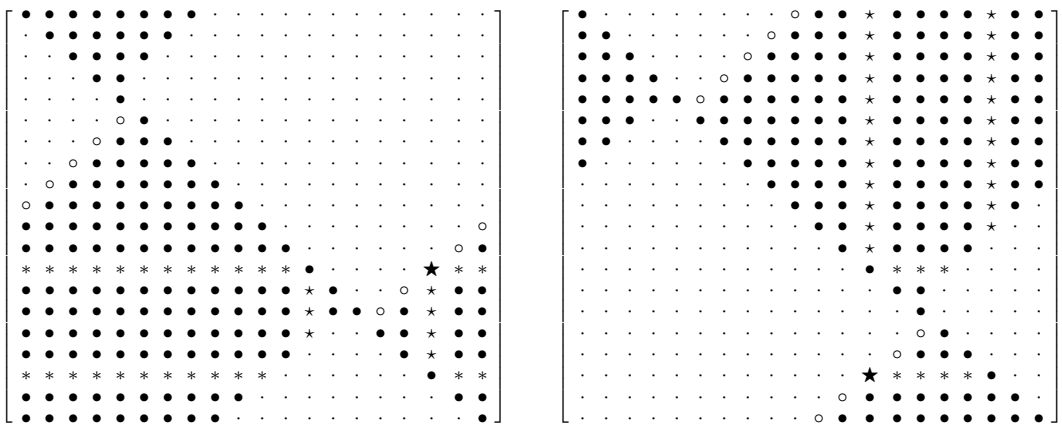


The symbol $\bullet$ denotes a possibly non-zero element, $\cdot$ denotes the null-element and $\star$ denotes the pivot element which becomes zero. We have used a preparatory "step zero" at the beginning, which annihilates the elements at positions $(6,1)$, $(5,2)$ and $(4,3)$. This corresponds to setting $Piv(0) = \mathcal{S}_6$. So, each cycle (or sweep) incorporates 6 parallel steps.

In general, when n is odd, every rotation set will contain (the same number of) $[\frac{n}{2}]$ pairs. For even n, the rotation sets will contain in turn $[\frac{n}{2}]$ and $[\frac{n-1}{2}]$ pairs. Each cycle is comprised of n parallel steps.

Let us look closer at the matrix $B^{[t]}$. As an example, we have taken $n = 20$ and $t = 11,\ 31$. In order to understand the effect of each left and right rotation within one parallel step, we have used $*$ to denote the elements which are transformed by ${U^{[11]}_{13,18}}^T$, and $\star$ to denote the elements which are transformed by $V^{[11]}_{13,18}$. The considered pivot element $b^{[11]}_{13,18}$ is denoted by $\bigstar$. The other (possibly non-zero) pivot elements of this parallel step are denoted by $\circ$. The diagonal elements are changed twice, but for them we use special transformation formulas, as noted above. They are denoted by $\bullet$. The same symbol is used for any possibly non-zero element. Note that the zeros at positions $(17,13)$ and $(18,12)$ will be destroyed.

$B^{[11]}$ $\qquad\qquad$ $B^{[31]}$



In this example, all matrix elements have changed during the parallel step. Within each parallel step, almost all matrix

elements will be transformed twice. First, the rows are transformed by the left rotations and then the columns are transformed by the right rotations. For $t = 31$, the structure of zeros takes the form which can be considered as the "transpose" of that for $t = 11$.

In general, $B^{[t]}$ and $B^{[t\pm n]}$ have the zero structures which are transposed to each other. It is known (see [15]) that each matrix $B^{[t]}$ is PST. Therefore it is ET, so all the matrices $B^{[t]}$ can be compactly saved in the upper triangle of a square array. Therefore, we can introduce the upper-triangular matrix $G^{[t]}$ by the requirement

$$G^{[t]} + G^{[t]T} = B^{[t]} + B^{[t]T}, \quad t \geq 1 \, .$$

The question arises whether the algorithm of KMMS can compactly be described in terms of the elements of the matrices $G^{[t]}$. The answer is affirmative. In addition, the transformation formulas for $G^{[t]}$ become simple and similar to those of the symmetric Jacobi method, when only the upper-triangle of the symmetric matrix is used.

3.5 KMMS on the sequential machines

The first task to solve in deriving the algorithm which uses $G^{[t]}$ is to settle $B^{[0]}$ in the upper triangle of $G^{[0]}$. In the following simple code which solves this problem, the same array `A` is used to accommodate $B^{(0)}$ and $G^{(0)}$. On input `A` accommodates $B^{(0)}$, on output $G^{[0]}$. The code (in the left box) uses the fact that the matrix in butterfly form is ET.

If $B^{[0]}$ has been obtained from some triangular matrix T (as discussed earlier) we can write a code which directly transforms the upper-triangular matrix T into the upper-triangular matrix $G^{[0]}$. Its essential part is displayed below in the right box.

```
    ZERO=0.0
    DO 1 K=1,N
      DO 2 I=K,(N+K-1)/2
       J=N+K-I
       TEMP=A(I,J)+A(J,I)
       A(I,J)=TEMP
       A(J,I)=ZERO
2     CONTINUE
      DO 3 I=1,(K-1)/2
       J=K-I
       TEMP=A(I,J)+A(J,I)
       A(I,J)=TEMP
       A(J,I)=ZERO
3     CONTINUE
1   CONTINUE
```

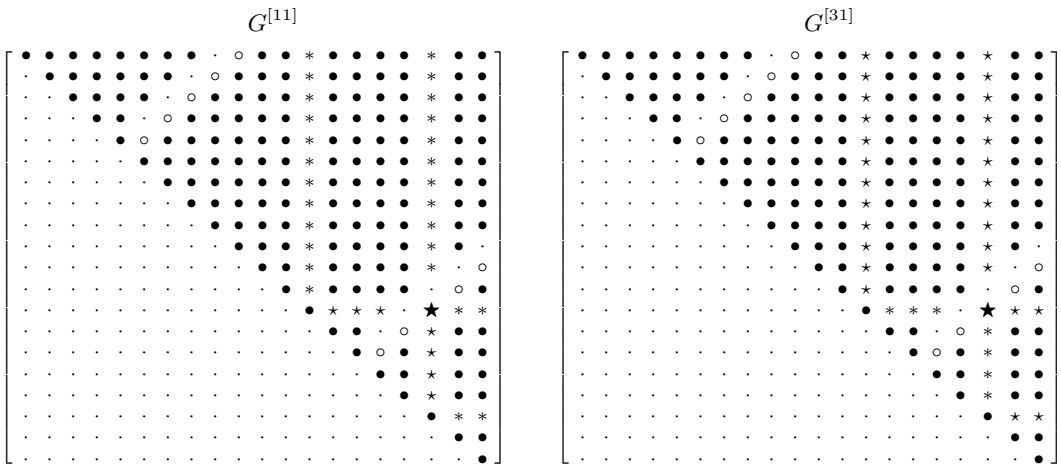
```
  DO 1 K=2,N
   DO 2 I=1,K/2
    J=K+1-I
    CALL SSWAP(I-1,A(1,I),1,A(1,J),1)
    CALL SSWAP(J-I-1,A(I,I+1),LDA,A(I+1,J),1)
    CALL SSWAP(N-J,A(I,J+1),LDA,A(J,J+1),LDA)
    TEMP=A(I,I)
    A(I,I)=A(J,J)
    A(J,J)=TEMP
    IF(SVECL)CALL SSWAP(N,U(1,I),1,U(1,J),1)
    IF(SVECR)CALL SSWAP(N,V(1,I),1,V(1,J),1)
2  CONTINUE
1 CONTINUE
```

The latter code has the following effect

$$T = \begin{bmatrix} 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 0 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\ 0 & 0 & 33 & 34 & 35 & 36 & 37 & 38 \\ 0 & 0 & 0 & 44 & 45 & 46 & 47 & 48 \\ 0 & 0 & 0 & 0 & 55 & 56 & 57 & 58 \\ 0 & 0 & 0 & 0 & 0 & 66 & 67 & 68 \\ 0 & 0 & 0 & 0 & 0 & 0 & 77 & 78 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 88 \end{bmatrix} \quad \mapsto \quad \begin{bmatrix} 88 & 68 & 48 & 28 & 18 & 38 & 58 & 78 \\ 0 & 66 & 46 & 26 & 16 & 36 & 56 & 67 \\ 0 & 0 & 44 & 24 & 14 & 34 & 45 & 47 \\ 0 & 0 & 0 & 22 & 12 & 23 & 25 & 27 \\ 0 & 0 & 0 & 0 & 11 & 13 & 15 & 17 \\ 0 & 0 & 0 & 0 & 0 & 33 & 35 & 37 \\ 0 & 0 & 0 & 0 & 0 & 0 & 55 & 57 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 77 \end{bmatrix} = G^{[0]} \, .$$

If T is lower-triangular, this code can easily be modified, or it can simply be used after transposing T. Note that after transposing T, the left and right singular vectors have to be swapped.

Let us now consider the transformation of $G^{[t]}$. If we go back to the example with the matrix of order 20, we can trace how the transformation of $B^{[11]}$ is reflected to the transformation of $G^{[11]}$. We use the same notation as before.



From the displayed matrices, we can derive the transformation formulas for one rotational step. To simplify notation, let the essential elements of $U_{ij}^{[t]}$ and $V_{ij}^{[t]}$ be denoted as in (2): $c_{\phi_{ij}}$, $s_{\phi_{ij}}$ and $c_{\psi_{ij}}$, $s_{\psi_{ij}}$, respectively. Let us consider the transformation $A \mapsto [U_{ij}^{[t]}]^T A V_{ij}^{[t]} = A'$. If $A = (a_{lm})$ and $A' = (a'_{lm})$, then for the first cycle, we have:

$$\left.\begin{aligned} a'_{li} &= c_{\phi_{ij}} a_{li} + s_{\phi_{ij}} a_{lj} \\ a'_{lj} &= s_{\phi_{ij}} a_{li} - c_{\phi_{ij}} a_{lj} \end{aligned}\right\} \quad 1 \le l \le i-1$$

$$\left.\begin{aligned} a'_{il} &= c_{\psi_{ij}} a_{il} + s_{\psi_{ij}} a_{lj} \\ a'_{lj} &= s_{\psi_{ij}} a_{il} - c_{\psi_{ij}} a_{lj} \end{aligned}\right\} \quad i+1 \le l \le j-1$$

$$\left.\begin{aligned} a'_{il} &= c_{\phi_{ij}} a_{il} + s_{\phi_{ij}} a_{jl} \\ a'_{jl} &= s_{\phi_{ij}} a_{il} - c_{\phi_{ij}} a_{jl} \end{aligned}\right\} \quad j+1 \le l \le n$$

$$a'_{ii} = \frac{c_{\phi_{ij}}}{c_{\psi_{ij}}} a_{ii}\,, \quad a'_{jj} = \frac{c_{\psi_{ij}}}{c_{\phi_{ij}}} a_{jj}$$

$$a'_{ij} = 0$$

In the second cycle the roles of angles ϕ_{ij} and ψ_{ij} are interchanged. If we speak in terms of parallel steps, then one cycle includes n such steps.

The above transformation formulas hold for odd cycles, while for even cycles each appearances of $c_{\psi_{ij}}$ and $s_{\psi_{ij}}$ has to be replaced by $c_{\phi_{ij}}$ and $s_{\phi_{ij}}$, respectively, and vice versa. To keep the programming code compact, we shall adopt that during odd cycles the variables `CSR` and `SNR` (`CSL` and `SNL`) assume the values of $c_{\psi_{ij}}$ and $s_{\psi_{ij}}$ ($c_{\phi_{ij}}$ and $s_{\phi_{ij}}$) while during even cycles they assume the values of $c_{\phi_{ij}}$ and $s_{\phi_{ij}}$ ($c_{\psi_{ij}}$ and $s_{\psi_{ij}}$). Then the code which transforms the array `A`, which accommodates the matrices $G^{[t]}$, becomes identical for all cycles. Some extra care has to be taken for updating the singular vector matrices.

Next, we display the essential parts of the programming code for KMMS, written in FORTRAN 77, double precision. This code is not yet adapted for parallel processing, but it is very simple and gives an idea how to write a code for standard computers. It uses subroutine `KOGSTEP` which performs one standard "rotational step" as described above. In `KOGSTEP` the BLAS 1 routine `DROT` (for applying a rotation to `A`) and the LAPACK routine `DLASV2` (for computing the SVD of an upper-triangular matrix of order two) have been used. To make the exposition clean, we stick with the industry-standard routines which are well documented. In a later stage we can replace `DLASV2` with a routine based on the UR, UL, LR and LL formulas described above. Logical variables `VL` and `VR` indicate whether left and right singular vectors are to be computed. The process is terminated when all off-diagonal elements become zero. The pivot element `A(I,J)` is set zero prior to the transformation if `ABS(A(I,J))` is not larger than `EPS*ABS(A(I,I))` and not larger than `EPS*ABS(A(J,J))`, where `EPS` is the machine precision.

Essential part of `KMMS`:

```
 ODD   = .TRUE.
*
  DO 10 ISWEEP=1,20
   IC=0
    DO 1 K=1,N
*     THE NEXT LOOP CAN BE PERFORMED IN PARALLEL
     DO 2 I=K,(N+K-1)/2
      CALL KOGSTEP(LDA,N,I,N+K-I,A,VL,U,VR,V,IC)
  2  CONTINUE
*     THE NEXT LOOP CAN BE PERFORMED IN PARALLEL
     DO 3 I=1,(K-1)/2
      CALL KOGSTEP(LDA,N,I,K-I,A,VL,U,VR,V,IC)
  3  CONTINUE
*
  1 CONTINUE
*
     IF(IC .EQ. 0)  RETURN
     ODD=.NOT.ODD
 10 CONTINUE
```

Essential part of `KOGSTEP`:

```
  IF(A(I,J).NE.ZERO)THEN
   IC=IC+1
   CALL DLASV2(A(I,I),A(I,J),A(J,J),
  1          SMIN,SMAX,SNR,CSR,SNL,CSL)
   CALL DROT(J-I-1,A(I,I+1),LDA,A(I+1,J),1,CSR,SNR)
   CALL DROT(I-1,A(1,I),1,A(1,J),1,CSL,SNL)
   CALL DROT(N-J,A(I,J+1),LDA,A(J,J+1),LDA,CSL,SNL)
   A(I,I)=SMAX
   A(J,J)=SMIN
   A(I,J)=ZERO
   IF(ODD)THEN
C            UPPER-TRIANGULAR PIVOT SUBMATRIX
    IF(VL) CALL DROT(N,U(1,I),1,U(1,J),1,CSL,SNL)
    IF(VR) CALL DROT(N,V(1,I),1,V(1,J),1,CSR,SNR)
   ELSE
C           LOWER-TRIANGULAR PIVOT MATRIX
    IF(VL) CALL DROT(N,U(1,I),1,U(1,J),1,CSR,SNR)
    IF(VR) CALL DROT(N,V(1,I),1,V(1,J),1,CSL,SNL)
   ENDIF
  ENDIF
```

The global and the quadratic convergence considerations of this method are given in [32].

It can be shown that a hybrid method, combining the KMMS with the Hestenes one-sided Jacobi method, proposed in [6], can be adapted to work with butterfly matrices in the same way (see [32]). This method is also provably relatively accurate, but requires more floating point operations per cycle, than KMMS.

3.6 The parallel algorithm

Before we propose a way how to parallelize the Kogbetliantz algorithm, let us note that a slowdown can be expected in executing the commands which access the elements by rows. These are: `CALL DROT(J-I-1,A(I,I+1),LDA,A(I+1,J),1,CSR,SNR)` and `CALL DROT(N-J,A(I,J+1),LDA,A(J,J+1),LDA,CSL,SNL)`, because the stride `LDA` is not unit. A way how to remove this bottleneck is to transform the rows into columns, prior to the transformation of rows. Since the algorithm applies in each parallel step around $n/2$ left rotations, this fast transpose should be performed at negligible cost.

To explain this idea, we return to the butterfly-like form of $B^{[t]}$ (see the page 56). Since $B^{[t]}$ is ET, an interchange of the elements $b^{[t]}_{pq} \neq 0$ and $b^{[t]}_{qp} = 0$ reduces to copying $b^{[t]}_{pq} \mapsto b^{[t]}_{qp}$ and copying $0 \mapsto b^{[t]}_{pq}$. Suppose we have at disposal a fast, parallel routine `transpose(t,A)` which can make a fast transpose of $B^{[t]}$ at any time t (here `A` represents $B^{[t]}$). In describing the parallel algorithm, we shall use a Pascal-like meta language and the following notation:

$$\texttt{S(t)} = \left\{1,\ldots\left[\frac{t-1}{2}\right]\right\} \cup \left\{t,\ldots\left[\frac{n+t-1}{2}\right]\right\}, \quad 1 \le t \le n,$$

$$\texttt{U(t)} = \{U^{[t]}_{ij} : (i,j) \in \texttt{S(t)}\}, \qquad \texttt{V(t)} = \{V^{[t]}_{ij} : (i,j) \in \texttt{S(t)}\},$$

$$\texttt{VECL} = \text{array, matrix of left singular vectors,} \qquad \texttt{VECR} = \text{array, matrix of right singular vectors,}$$

$$\texttt{left} = \text{logical variable, true if the left rotations are accumulated in VECL,}$$

$$\texttt{right} = \text{logical variable, true if the right rotations are accumulated in VECR,}$$

We recall, even if the both singular vector matrices are wanted, only one will be computed during iteration. Because the other one can cheaply and safely be computed a posteriori. So, at most one of the matrices `VECL`, `VECR` has to be computed in the main loop of the code. An outline of the parallel algorithm follows.

```
repeat
for t=1:n do
 - compute concurrently the parameters of all rotations from U(t) and V(t)
 - if (right) apply concurrently all the right rotations to VECR
 - apply concurrently all the right rotations to the appropriate parts of
                                                    the columns of A
 - transpose(t,A)
 - if (left) apply concurrently all the left rotations to VECL
 - apply all the left rotations on the appropriate parts of the columns of A
 - transpose(t,A)
end{t}
until convergence
```

Such an algorithm can be implemented on computer architectures with shared memory. Its performance can be enhanced if the data structures are suitably chosen. Say, if `VECL` (or `VECR`) and `A` are contained within the same array, then the bundle of left (right) rotations can be applied more efficiently, with less calls to `DROT`. When implemented, the program might assume that `A` holds the matrix $G^{[t]}$ instead $B^{[t]}$.

However, the algorithm above presumes that at least $n/2$ processors are available. For larger matrices this will not be an option. To remove this shortcoming, we have to modify the algorithm so that it works with blocks instead of elements.

4 A Block Version of the Algorithm

Working with blocks means making matrix algorithms more efficient. They become BLAS 3 algorithms which can better use the computing resources of contemporary processors. This block-oriented approach to parallelizing Kogbetliantz method is more promising (see [11]), because it can be adapted to the number of available processors and to their inherent characteristics, like the capacities of the internal memory on different hierarchy levels.

The derivation of the block version of the modulus Kogbetliantz method is related to the ideas described in [19] and [20].

We begin our consideration with a triangular T, which is obtained after applying one or two QR factorizations to the initial matrix. We shall use the following block-partition of T,

$$T = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1m} \\ 0 & T_{22} & \cdots & T_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & T_{mm} \end{bmatrix}.$$

Here, each diagonal block T_{ii} is of order $n_i \geq 1$, so that $\mathcal{M} = (n_1, n_2, \ldots, n_m)$ makes a partition of n. The choice of $\mathcal{M}$ depends on computing resources, especially on the available fast memory within each processor. To make the exposition simpler, we shall assume $n_1 = n_2 = \cdots = n_m = n/m$. The same partition will be used for all the matrices obtained from T.

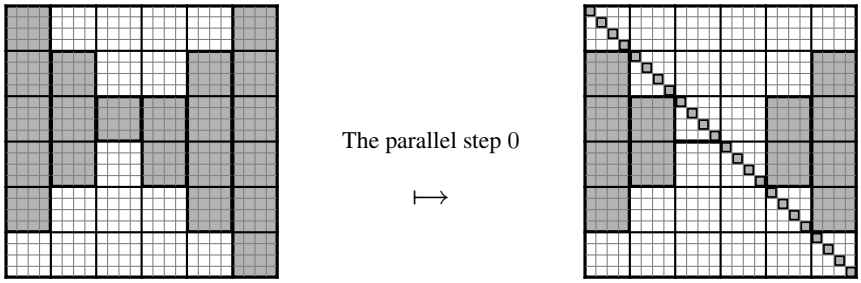
To reduce T to the "block-butterfly form", we can use permutational similarity, quite similar to (1). However, we have to assume that each I_{ij} in (1) is product of certain simple transpositions. The effect of $I_{ij}^T T$ (TI_{ij}) is to swap the block rows (block columns) i and j of T. So, the left-hand permutations in (1) have to be written transposed.

4.1 The parallel step zero

Let $B^{[0]} = B$ be a matrix in the block-butterfly form. Before commencing the iteration, we shall need a simple preparation of the starting matrix, which we call the *parallel step zero*. It consists of annihilating the blocks $B_{1m}^{[0]}$, $B_{2,m-1}^{[0]}, \ldots$, and of diagonalizing all diagonal blocks. Using the notation from the relation (8), this initial step can be described as follows

$$B^{[1]} = {U^{[0]}}^T B^{[0]} V^{[0]}, \quad U^{[0]} = \prod_{(i,j)\in \mathbf{Piv}(m)} \mathbf{U}_{ij}^{[0]}, \quad V^{[0]} = \prod_{(i,j)\in \mathbf{Piv}(m)} \mathbf{V}_{ij}^{[0]},$$

where $\mathbf{Piv}(m) = \{(1,m), (2,m-1)\ldots\}$ is the *pivot set* associated with the mth step of the block algorithm. The figure below indicates the changes in the matrix. We have taken $n = 24$, $m = 6$ and $n_i = 4$ for $1 \leq i \leq m$.



$B^{[0]}$ in butterfly form $B^{[1]}$ in butterfly form

The iterative part then generates the matrices $B^{[2]}$, $B^{[3]}$, ... in a similar fashion as earlier.

To make the block algorithm efficient, we shall represent $B^{[t]}$ in the factored form: $B^{[t]} = {E^{[t]}}^T C^{[t]} F^{[t]}$, where $E^{[t]}$ and $F^{[t]}$ are block-diagonal and orthogonal (cf. [19]). In addition, we shall keep the diagonal of the current iterate $B^{[t]}$, separately in the vector $\gamma^{[t]}$. Consequently, we shall require more outputs from the parallel step zero. In particular, $C^{[1]}$, $E^{[1]}$, $F^{[1]}$ and $\gamma^{[1]}$ will be the output data.

In the description of the block algorithm, we shall use the appropriate block-column partition of $B^{[t]}$, $B^{[t]} = [B_1^{[t]}, B_2^{[t]}, \ldots, B_m^{[t]}]$, $t \geq 0$. The following lines represent a brief algorithmic description of the parallel step zero.

Parallel step zero:
for $i = 1$ to $\circ$ div$(m, 2)$ do in parallel

set $j = m + 1 - i$

(a) Compute the SVD $\begin{bmatrix} B_{ii}^{[0]} & B_{ij}^{[0]} \\ 0 & B_{jj}^{[0]} \end{bmatrix} = \mathbf{U}_{ij}^{[0]}\, \mathbf{\Gamma}_{ij}\, {\mathbf{V}_{ij}^{[0]}}^T$ and the CS decompositions of $\mathbf{U}_{ij}^{[0]}$ and $\mathbf{V}_{ij}^{[0]}$,

$$\mathbf{U}_{ij}^{[0]} = \begin{bmatrix} \dot{U}_{ii} & 0 \\ 0 & \dot{U}_{jj} \end{bmatrix} \Theta_{ij} \begin{bmatrix} \ddot{U}_{ii} & 0 \\ 0 & \ddot{U}_{jj} \end{bmatrix}, \quad \mathbf{V}_{ij}^{[0]} = \begin{bmatrix} \dot{V}_{ii} & 0 \\ 0 & \dot{V}_{jj} \end{bmatrix} \Phi_{ij} \begin{bmatrix} \ddot{V}_{ii} & 0 \\ 0 & \ddot{V}_{jj} \end{bmatrix}$$

(b) Apply: $B_i' = B_i \dot{V}_{ii}$, $B_j' = B_j \dot{V}_{jj}$ and afterwards: $[B_i'', B_j''] = [B_i', B_j']\Phi_{ij}$
If (*right*) then $\mathsf{V}_i{}' = \mathsf{V}_i \dot{V}_{ii}$, $\mathsf{V}_j' = \mathsf{V}_j \dot{V}_{jj}$ and afterwards: $[\bar{\mathsf{V}}_i^{[1]}, \bar{\mathsf{V}}_j^{[1]}] = [\mathsf{V}_i', \mathsf{V}_j']\Phi_{ij}$

(c) Transpose: $\bar{B} = {B''}^T$ (let $\bar{B} = [\bar{B}_1, \bar{B}_2, \ldots, \bar{B}_m]$ be the block-column partition of $\bar{B}$)

(d) Apply: $\bar{B}_i' = \bar{B}_i \dot{U}_{ii}$, $\bar{B}_j' = \bar{B}_j \dot{U}_{jj}$ and afterwards: $[\bar{B}_i'', \bar{B}_j''] = [\bar{B}_i', \bar{B}_j']\Theta_{ij}$
If (*left*) then $\mathsf{U}_i' = \mathsf{U}_i \dot{U}_{ii}$, $\mathsf{U}_j' = \mathsf{U}_j \dot{U}_{jj}$ and afterwards: $[\bar{\mathsf{U}}_i^{[1]}, \bar{\mathsf{U}}_j^{[1]}] = [\mathsf{U}_i', \mathsf{U}_j']\Theta_{ij}$

(e) Transpose: $C^{[1]} = [\bar{B}'']^T$

(f) Copy: $E_{ii}^{[1]} = \ddot{U}_{ii}$, $E_{jj}^{[1]} = \ddot{U}_{jj}$, $F_{ii}^{[1]} = \ddot{V}_{ii}$, $F_{jj}^{[1]} = \ddot{V}_{jj}$
Copy the first n_i and the last n_j diagonal elements of $\mathbf{\Gamma}_{ij}$ into the appropriate parts of the vector $\gamma^{[1]}$.

end for

Here, we have assumed that the matrices U and V (of left and right singular vectors accumulated later in the process) have emerged after computing T (by the QRP followed possibly by the LQ factorization). Their columns have been later rearranged by the permutations which delivered B. Their block-column partitions are $[\mathsf{U}_1, \ldots, \mathsf{U}_m]$ and $[\mathsf{V}_1, \ldots, \mathsf{V}_m]$, respectively. In the above algorithm U and V have been updated so that $\mathsf{U}^{[1]} = \bar{\mathsf{U}}^{[1]} E^{[1]}$ and $\mathsf{V}^{[1]} = \bar{\mathsf{V}}^{[1]} F^{[1]}$ holds. In the iterative part of the algorithm, the matrices $\bar{\mathsf{U}}^{[t]}$ and $\bar{\mathsf{V}}^{[t]}$, $t \geq 1$ will be iterated.

The command "transpose" presumes that each processor will do its partial job, like transposing block columns and block rows i and j. It is presumed that the initial block-partition takes into account the cache memory capacity, i.e., that all small matrices (those with two subscripts) can all be contained in the internal memory of each processor.

4.2 The iterative part of the algorithm

When the block Kogbetliantz method, defined by the block-modulus strategy, is applied to the matrix $B^{[1]}$ which is in the block butterfly form, then it behaves like the simple modulus Kogbetliantz algorithm when applied to the matrix in

butterfly form. The essential part of the algorithm, which achieves this harmony is the annihilation of the off-block-diagonal pivot submatrices $B_{ij}^{[t]}$, $(i,j) \in \mathbf{Piv}(t)$. The algorithm becomes simpler and more efficient if in addition, the diagonal blocks $B_{ii}^{[t]}$, $B_{jj}^{[t]}$, $(i,j) \in \mathbf{Piv}(t)$ are diagonalized. Note that this has initially been achieved in the parallel step zero. Then after each parallel step, the off-norm of the current iterate will be reduced by the rule

$$\|\Omega(B^{[t+1]})\|^2 = \|\Omega(B^{[t]})\|^2 - \sum_{(i,j)\in\mathbf{Piv}(t)} \|B_{ij}^{[t]}\|^2, \quad t \geq 1.$$

Since $B^{[t]}$ is kept factored as ${E^{[t]}}^T C^{[t]} F^{[t]}$, we have to derive how $C^{[t]}$, $E^{[t]}$, $F^{[t]}$, the vector $\gamma^{[t]}$ and $\bar{\mathsf{U}}^{[t]}$, $\bar{\mathsf{V}}^{[t]}$ are updated.

In its raw form, the parallel block Kogbetliantz algorithm takes the form (8), where $U_{ij}^{[t]}$ and $V_{ij}^{[t]}$ are "block rotations" whose essential parts are $(n_i + n_j) \times (n_i + n_j)$ orthogonal matrices $\mathbf{U}_{ij}^{[t]}$ and $\mathbf{V}_{ij}^{[t]}$ which satisfy

$$\mathbf{B}_{ij}^{[t]} = \begin{bmatrix} B_{ii}^{[t]} & B_{ij}^{[t]} \\ 0 & B_{jj}^{[t]} \end{bmatrix} = \mathbf{U}_{ij}^{[t]}\, \mathbf{\Gamma}_i\, {\mathbf{V}_{ij}^{[t]}}^T, \qquad \mathbf{\Gamma}_i \text{ diagonal}. \tag{9}$$

Here $\mathbf{B}_{ij}$ is upper-triangular. The same will hold if $\mathbf{B}_{ij}$ is lower-triangular. We assume that the diagonal blocks $B_{ii}^{[t]}$ and $B_{jj}^{[t]}$ are diagonal. This is certainly true for $t = 1$. Since $B^{[t]}$ is given in factored form, we first have to see how to compute $\mathbf{B}_{ij}^{[t]}$. Note that

$$\begin{bmatrix} B_{ii}^{[t]} & B_{ij}^{[t]} \\ 0 & B_{jj}^{[t]} \end{bmatrix} = \begin{bmatrix} E_{ii}^{[t]} & 0 \\ 0 & E_{jj}^{[t]} \end{bmatrix}^T \begin{bmatrix} C_{ii}^{[t]} & C_{ij}^{[t]} \\ 0 & C_{jj}^{[t]} \end{bmatrix} \begin{bmatrix} F_{ii}^{[t]} & 0 \\ 0 & F_{jj}^{[t]} \end{bmatrix} = \begin{bmatrix} {E_{ii}^{[t]}}^T C_{ii}^{[t]} F_{ii}^{[t]} & {E_{ii}^{[t]}}^T C_{ij}^{[t]} F_{jj}^{[t]} \\ 0 & {E_{jj}^{[t]}}^T C_{jj}^{[t]} F_{jj}^{[t]} \end{bmatrix}.$$

Since, $B_{ii}^{[t]}$ and $B_{jj}^{[t]}$ are diagonal, we first copy zeros in all parts of $\mathbf{B}_{ij}^{[t]}$ except $B_{ij}^{[t]}$. Then we copy the diagonal elements from the vector $\gamma^{[t]}$ to the diagonal of $B_{ii}^{[t]}$ and of $B_{jj}^{[t]}$. After that, we have to compute ${E_{ii}^{[t]}}^T C_{ij}^{[t]} F_{jj}^{[t]}$. This should be accomplished within the processor which is associated with the pair $(i,j) \in \mathbf{Piv}(t)$. The matrix multiplications can be computed using some fast routine like the BLAS 3 routine `*GEMM`. If $\mathbf{B}_{ij}$ is lower-triangular, then $B_{ji}^{[t]} = {E_{jj}^{[t]}}^T C_{ji}^{[t]} F_{ii}^{[t]}$, so the procedure is almost the same.

Next, the SVD of $\mathbf{B}_{ij}^{[t]}$ is computed as indicated in (9). Here, one can choose among several fast and accurate methods, say the one-sided Jacobi or the Kogbetliantz (serial or modulus) method. After that, the CS decompositions of $\mathbf{U}_{ij}^{[t]}$ and $\mathbf{V}_{ij}^{[t]}$ are computed (see [20])

$$\mathbf{U}_{ij}^{[t]} = \begin{bmatrix} \dot{U}_{ii}^{[t]} & 0 \\ 0 & \dot{U}_{jj}^{[t]} \end{bmatrix} \Theta_{ij}^{[t]} \begin{bmatrix} \ddot{U}_{ii}^{[t]} & 0 \\ 0 & \ddot{U}_{jj}^{[t]} \end{bmatrix}, \quad \mathbf{V}_{ij}^{[t]} = \begin{bmatrix} \dot{V}_{ii}^{[t]} & 0 \\ 0 & \dot{V}_{jj}^{[t]} \end{bmatrix} \Phi_{ij}^{[t]} \begin{bmatrix} \ddot{V}_{ii}^{[t]} & 0 \\ 0 & \ddot{V}_{jj}^{[t]} \end{bmatrix}.$$

Here $\Theta_{ij}^{[t]}$ and $\Phi_{ij}^{[t]}$ are orthogonal matrices; actually, the products of at most $\min\{n_i, n_j\}$ commuting plane rotations.

Next, $B^{[t+1]}$ is computed as ${U^{[t]}}^T (B^{[t]} V^{[t]})$. It is useful to introduce $J_{ij} = [J_i, J_j]$ where $I_n = [J_1, \ldots, J_m]$ is the block-column partition of the identity matrix. Then $B^{[t]} J_{ij} = [B_i^{[t]}, B_j^{[t]}]$. If we post-multiply the equation $B^{[t+1]} = {U^{[t]}}^T (B^{[t]} V^{[t]})$, or better ${E^{[t+1]}}^T C^{[t+1]} F^{[t+1]} = {U^{[t]}}^T ({E^{[t]}}^T C^{[t]} F^{[t]}) V^{[t]}$ by J_{ij}, we obtain

$$\begin{aligned} {E^{[t+1]}}^T [C_i^{[t+1]}, C_j^{[t+1]}] \begin{bmatrix} F_{ii}^{[t+1]} & 0 \\ 0 & F_{jj}^{[t+1]} \end{bmatrix} &= {U^{[t]}}^T {E^{[t]}}^T [C_i^{[t]}, C_j^{[t]}] \begin{bmatrix} F_{ii}^{[t]} & 0 \\ 0 & F_{jj}^{[t]} \end{bmatrix} \mathbf{V}_{ij}^{[t]} \\ &= {U^{[t]}}^T {E^{[t]}}^T [C_i^{[t]}, C_j^{[t]}] \left(\left(\begin{bmatrix} F_{ii}^{[t]} & 0 \\ 0 & F_{jj}^{[t]} \end{bmatrix} \begin{bmatrix} \dot{V}_{ii}^{[t]} & 0 \\ 0 & \dot{V}_{jj}^{[t]} \end{bmatrix} \right) \Phi_{ij}^{[t]} \right) \begin{bmatrix} \ddot{V}_{ii}^{[t]} & 0 \\ 0 & \ddot{V}_{jj}^{[t]} \end{bmatrix}. \end{aligned}$$

Hence, we have to make the following updates

$$[\bar{C}_i^{[t]}, \bar{C}_j^{[t]}] = \left([C_i^{[t]}, C_j^{[t]}] \begin{bmatrix} F_{ii}^{[t]} \dot{V}_{ii}^{[t]} & 0 \\ 0 & F_{jj}^{[t]} \dot{V}_{jj}^{[t]} \end{bmatrix} \right) \Phi_{ij}^{[t]}, \quad F_{ii}^{[t+1]} = \ddot{V}_{ii}^{[t]}, \quad F_{jj}^{[t+1]} = \ddot{V}_{jj}^{[t]}.$$

These updates can be performed concurrently for all $(i,j) \in \mathbf{Piv}(t)$. This results in the block-diagonal orthogonal matrix $F^{[t+1]}$ and in the auxiliary matrix $\bar{C}^{[t]}$.

The rule for updating the right singular vector matrix, is obtained from: $\bar{\mathsf{V}}^{[t+1]} F^{[t+1]} J_{ij} = (\bar{\mathsf{V}}^{[t]} F^{[t]}) V^{[t]} J_{ij}$. We obtain

$$[\bar{\mathsf{V}}_i^{[t+1]}, \bar{\mathsf{V}}_j^{[t+1]}] = [\bar{\mathsf{V}}_i^{[t]}, \bar{\mathsf{V}}_j^{[t]}] \begin{bmatrix} F_{ii}^{[t]} \dot{V}_{ii}^{[t]} & 0 \\ 0 & F_{jj}^{[t]} \dot{V}_{jj}^{[t]} \end{bmatrix} \Phi_{ij}^{[t]} = [\, \bar{\mathsf{V}}_i^{[t]} (F_{ii}^{[t]} \dot{V}_{ii}^{[t]}) \,,\, \bar{\mathsf{V}}_j^{[t]} (F_{jj}^{[t]} \dot{V}_{jj}^{[t]}) \,] \Phi_{ij}^{[t]}.$$

Now, let us consider the left transformation. Note that $J_i^T X$ is the ith block-row of X. Pre-multiplying the equation ${E^{[t+1]}}^T C^{[t+1]} = {U^{[t]}}^T {E^{[t]}}^T \bar{C}^{[t]}$ by J_{ij}^T, we obtain

$$\begin{aligned} \begin{bmatrix} E_{ii}^{[t+1]} & 0 \\ 0 & E_{jj}^{[t+1]} \end{bmatrix}^T \begin{bmatrix} J_i^T C^{[t+1]} \\ J_j^T C^{[t+1]} \end{bmatrix} &= {\mathbf{U}_{ij}^{[t]}}^T \begin{bmatrix} E_{ii}^{[t]} & 0 \\ 0 & E_{jj}^{[t]} \end{bmatrix}^T \begin{bmatrix} J_i^T \bar{C}^{[t]} \\ J_j^T \bar{C}^{[t]} \end{bmatrix} \\ &= \begin{bmatrix} \ddot{U}_{ii}^{[t]} & 0 \\ 0 & \ddot{U}_{jj}^{[t]} \end{bmatrix}^T \left({\Theta_{ij}^{[t]}}^T \begin{bmatrix} \dot{U}_{ii}^{[t]} & 0 \\ 0 & \dot{U}_{ii}^{[t]} \end{bmatrix}^T \begin{bmatrix} E_{ii}^{[t]} & 0 \\ 0 & E_{ii}^{[t]} \end{bmatrix}^T \begin{bmatrix} J_i^T \bar{C}^{[t]} \\ J_j^T \bar{C}^{[t]} \end{bmatrix} \right). \end{aligned}$$

Hence, we have to make the following updates

$$[\tilde{C}_i^{[t+1]}, \tilde{C}_j^{[t+1]}] = \left([\tilde{\bar{C}}_i^{[t]}, \tilde{\bar{C}}_j^{[t]}] \begin{bmatrix} E_{ii}^{[t]} \dot{U}_{ii}^{[t]} & 0 \\ 0 & E_{jj}^{[t]} \dot{U}_{jj}^{[t]} \end{bmatrix} \right) \Theta_{ij}^{[t]}, \quad E_{ii}^{[t+1]} = \ddot{U}_{ii}^{[t]}, \quad E_{jj}^{[t+1]} = \ddot{U}_{jj}^{[t]},$$

where $\tilde{\bar{C}}^{[t]} = \bar{C}^{[t]\,T}$ and $\tilde{C}^{[t+1]} = C^{[t+1]\,T}$. Thus, after computing $\bar{C}^{[t]}$, we have to transpose it. Then $[\bar{C}^{[t]}]^T$ is updated from the right-hand side and ${C^{[t+1]}}^T$ is obtained. The block-diagonal orthogonal matrix $E^{[t+1]}$ is computed.

The rule for updating the left singular vector matrix is obtained in an obvious way, in the following form

$$[\bar{\mathsf{U}}_i^{[t+1]}, \bar{\mathsf{U}}_j^{[t+1]}] = [\bar{\mathsf{U}}_i^{[t]}, \bar{\mathsf{U}}_j^{[t]}] \begin{bmatrix} E_{ii}^{[t]} \dot{U}_{ii}^{[t]} & 0 \\ 0 & E_{jj}^{[t]} \dot{U}_{jj}^{[t]} \end{bmatrix} \Theta_{ij}^{[t]} = [\, \bar{\mathsf{U}}_i^{[t]} (E_{ii}^{[t]} \dot{U}_{ii}^{[t]}) \,,\, \bar{\mathsf{U}}_j^{[t]} (E_{jj}^{[t]} \dot{U}_{jj}^{[t]}) \,] \Theta_{ij}^{[t]}.$$

Finally, we have to copy the diagonal elements of Γ_i to the appropriate places in $\gamma^{[t+1]}$. Again all these updates can be performed concurrently, for all $(i,j) \in \mathbf{Piv}(t)$.

After all these preparations, we can write down the iterative part of the algorithm. We shall use an $n \times n$ array A to accommodate $C^{[t]}$, $\bar{C}[t]$ and their transposes. Similarly, the $n \times nb$ arrays E and F will be used to hold the diagonal blocks of the block-diagonal orthogonal matrices $E^{[t]}$ and $F^{[t]}$. Here, nb is not smaller than $\max_i n_i$. We shall use the notation E_i and F_i for $E_{ii}^{[t]}$ and $F_{ii}^{[t]}$, respectively. The block columns of A will be denoted by $A_1, \ldots, A_m$ and A_{ij} will be the (i,j)th block of A. The one-dimensional array g will play the role of the vector $\gamma^{[t]}$ and several smaller $2nb \times 2nb$ arrays B, U, V will also be used. To make description of the algorithm simpler, we shall use some additional, even smaller arrays. Actually, they are not needed, since certain blocks of the larger, already mentioned arrays, can be used for the same purpose. The matrices of singular vectors are updated in the arrays VECL or VECR, depending on the values of the logical variables *left* or *right*.

The parallel block Kogbetliantz algorithm:

repeat

for t=1:m do
for $(i,j) \in \mathbf{Piv}(t)$ do in parallel

(a1) Compute: $B_{12} = E_i^T A_{ij} F_j$

(a2) Copy: from g to $\mathrm{diag}(B_{11})$ and to $\mathrm{diag}(B_{22})$ and form: $B = \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix}$ B_{11} is diagonal, B_{22} is diagonal

(b1) Compute SVD: $B = U\,\Gamma\,V^T$

(b2) Update: g from Γ

(b3) Compute CS decomposition: $U = \begin{bmatrix} U1 & 0 \\ 0 & U2 \end{bmatrix} H \begin{bmatrix} U3 & 0 \\ 0 & U4 \end{bmatrix}$

(b4) Compute CS decomposition: $V = \begin{bmatrix} V1 & 0 \\ 0 & V2 \end{bmatrix} K \begin{bmatrix} V3 & 0 \\ 0 & V4 \end{bmatrix}$

(c1) Compute: $X = F_i\,V1, \quad Y = F_j\,V2$

(c2) Update: $A_i \leftarrow A_i\,X, \quad A_j \leftarrow A_j\,Y;$
If (*right*) update: $\mathrm{VECR}_i \leftarrow \mathrm{VECR}_i\,X, \quad \mathrm{VECR}_j \leftarrow \mathrm{VECR}_j\,Y$

(c3) Update: $[A_i, A_j] \leftarrow [A_i, A_j]K;$ if (*right*) update: $[\mathrm{VECR}_i, \mathrm{VECR}_j] \leftarrow [\mathrm{VECR}_i, \mathrm{VECR}_j]K$

(c4) Update: $F_i \leftarrow V3, \quad F_j \leftarrow V4$

(d1) Transpose: $A \leftarrow A^T$

(e1) Compute: $X = E_i\,U1, \quad Y = E_j\,U2$

(e2) Update: $A_i \leftarrow A_i X, \quad A_j \leftarrow A_j Y;$
If (*left*) update: $\mathrm{VECL}_i \leftarrow \mathrm{VECL}_i\,X, \quad \mathrm{VECL}_j \leftarrow \mathrm{VECL}_j\,Y$

(e3) Update: $[A_i, A_j] \leftarrow [A_i, A_j]H;$ If (*left*) update: $[\mathrm{VECL}_i, \mathrm{VECL}_j] \leftarrow [\mathrm{VECL}_i, \mathrm{VECL}_j]H$

(e4) Update: $E_i \leftarrow U3, \quad E_j \leftarrow U4$

(f1) Transpose: $A \leftarrow A^T$

end for
end for

until convergence

It is important that all small arrays like B, U, V, X, Y can be stored in the fast memory. Then the bookkeeping of these matrices together with all needed factorizations (SVD, CS) is performed at low extra cost in computational time. This claim especially refers to the commands (b1)–(b4), (c1) and (e1). The gain, which lies in updating the block columns (which is the slowest part of the algorithm), justifies all this extra work.

The above modification of the original algorithm uses, what can be considered as the block version of the fast-scaled rotations (see [19], [20]). In contrast to the fast-scaled rotations, here there is no danger of the potential growth of the elements, since the transformations are orthogonal. However, for such purposes the existing algorithms for computing the CS decomposition are not appropriate and new algorithms are needed (see [20]).

Note that the parallel step zero can easily be rewritten using the above notation. The whole algorithm can be rewritten assuming that the array A holds $G^{[t]}$ instead of $B^{[t]}$. To do so, one just has to see how transposing $B^{[t]}$ reflects on $G^{[t]}$.

In order to prove the relative accuracy of this modification, one can try to modify the accuracy proof from [20]. However, this is beyond the scope of this paper. We note that the complete accuracy proof for the serial Kogbetliantz method has not yet been published.

5 Concluding Remarks

In this paper we have presented ideas and arguments for modifying the Kogbetliantz method to cope with parallel processing on architectures with shared memory. On standard computers the serial (modulus) Kogbetliantz method is a reliable and accurate algorithm for computing the SVD of matrices in triangular (butterfly) form. It is generally slow, except for almost diagonal matrices, with sufficiently small off-diagonal elements. Hence, the purpose of the presented research has been to give strong arguments for the claim that an efficient parallel modification of the method can be designed. To this end, we have constructed a parallel BLAS 3 algorithm, which largely diminishes the slowdown coming from the left-hand transformations. The efficiency of the algorithm has been enhanced by a trick, proposed in [20] and [19], which accelerates updating the block columns. We are confident that the presented block algorithm is relatively accurate for well-behaved matrices provided that the inner SVD and CS decompositions of small matrices are computed with relatively accurate algorithms.

A proper testing of the new algorithm on parallel architectures with shared memory is still lacking. Also, the proper proofs of the relative accuracy and of the global convergence of this block algorithm is lacking. The ideas and techniques how to tackle these problems can be found in [20], [22] and [32]. It is also an open question how to adapt the algorithm for distributed memory computing.

We believe that this modification can be implemented as a relatively accurate and efficient part of a compound algorithm for computing the SVD of general matrices. The highest potential of the presented algorithm is expected for the case of almost diagonal initial matrices, which are in the triangular or butterfly form.

Acknowledgements. The authors are thankful to D. Kressner and G. Okša for reading the paper and to the anonymous referees for valuable comments which improved the paper.

References

[1] M. Bečka, G. Okša G. and M. Vajteršic, Dynamic Ordering for a Parallel Block-Jacobi SVD algorithm, *Parallel Computing* **28** Issue **2** 243 - 262 (2002).

[2] M. Bečka M. and G. Okša, On Variable Blocking Factor in a Parallel Dynamic Block: Jacobi SVD Algorithm, *Parallel Computing* **29** Issue **9** 1153 - 1174 (2003) .

[3] J.P. Charlier and P. Van Dooren, On Kogbetliantz's SVD Algorithm in the Presence of Clusters, *Linear Algebra and Its Applications* **95** 135–160 (1987).

[4] J.P. Charlier, M. Vanbegin and P. Van Dooren, On Efficient Implementations of Kogbetliantz's Algorithm for Computing the Singular Value Decomposition, *Numerische Mathematik* **52** 279-300 (1988).

[5] J. Demmel and K. Veselić, Jacobi's method is more accurate than QR, *SIAM J. Matrix Anal. Appl.* **13** 1204–1245 (1992).

[6] Z. Drmač, *Computing the singular and the generalized singular values*. Ph. D. University of Hagen, Germany, 1994.

[7] Z. Drmač, A posteriori computation of the singular vectors in a preconditioned Jacobi SVD algorithm, *IMA J. Numer. Anal.* **19** 191-213 (1999).

[8] Z. Drmač and K. Veselić, New Fast and Accurate Jacobi SVD Algorithm: I, LAPACK Working Note **169**, 2005. Available online at http: www.netlib.org/lapack/lawns/downloads/

[9] Z. Drmač and K. Veselić, New Fast and Accurate Jacobi SVD Algorithm: II, LAPACKWorking Note **170**, 2005. Available online at http: www.netlib.org/lapack/lawns/downloads/

[10] K.V. Fernando, Linear Convergence of the Row Cyclic Jacobi and Kogbetliantz Methods, *Numer. Math.* **56** 71-91 (1989).

[11] K.V. Fernando and S. Hammarling, On Block Kogbetliantz Methods for Computation of the SVD, *SVD and signal processing: algorithms, applications and architectures*, 349 - 355 (1989), ISBN:0-444-70439-6.

[12] G.E. Forsythe and P. Henrici, The Cyclic Jacobi Method for Computing the Principle Values of a Complex Matrix, *Trans. Amer. Math. Soc.* **94** 1–23 (1960).

[13] J. Götze, On the Parallel Implementation of Jacobi and Kogbetliantz Algorithms, *SIAM J. Sci. Comp.* **15**, Issue **6** 1331 - 1348 (1994).

[14] V. Hari, On the Quadratic Convergence of Jacobi Algorithms, *Radovi Matematički* **2** 127–146 (1986).

[15] V. Hari and K. Veselić, On Jacobi methods for singular value decompositions, *SIAM J. Sci. Stat. Comput.* Vol. **8** No. **5** 741–754 (1987).

[16] V. Hari, On the Quadratic Convergence of the Serial SVD Jacobi Methods for Triangular Matrices, *SIAM J. Sci. Stat. Comput.* Vol. **10** No. **6** 1076–1096 (1989).

[17] V. Hari, On Sharp Quadratic Convergence Bounds for the Serial Jacobi Methods, *Numer. Math.* **60** 375–406 (1991).

[18] V. Hari and J. Matejaš, Quadratic Convergence of Scaled Iterates by Kogbetliantz Method, *Computing [Suppl]* **16** 83-105 (2003).

[19] V. Hari, On Some New Applications of the CS Decomposition, *ICNAAM 2004: Extended Abstracts (Editor T. E. Simos and Ch. Tsitouras)* ISBN: 3-527-40563-1, 161-163 (2004).

[20] V. Hari, Accelerating the SVD Block-Jacobi Method, *Computing* **75** Issue **1** 27–53 (2005).

[21] V. Hari and J. Matejaš, Accuracy of the Kogbetliantz method for triangular matrices, Preprint, University of Zagreb, 2005.

[22] V. Hari, Convergence of o Block-oriented Quasi-cyclic Jacobi Method, to appear in *SIAM J. Matrix Anal. Appl.*

[23] E. Kogbetliantz, Diagonalization of General Complex Matrices as a New Method for Solution of Linear Equations, *Proc. Imtern. Congr. Math. Amsterdam* **2** 356–357 (1954).

[24] E. Kogbetliantz, Solutions of Linear Equations by Diagonalization of Coefficient Matrices, *Quart. Appl. Math.* **13** 123–132 (1955).

[25] T. Londre and N.H. Rhee, Numerical Stability of the Parallel Jacobi Method, *SIAM J. MAA.* Vol. **26** No. **4** 985–1000 (2005).

[26] F.T. Luk, A Triangular Processor Array for Computing Singular Values, *Linear Algebra and Its Applications* **77** 259–273 (1986).

[27] F.T. Luk and H. Park, A Proof of Convergence for Two Parallel Jacobi SVD Algorithms, *IEEE Transactions on Computers* Vol. **38** Issue **6** 806 - 811 (1989).

[28] F.T. Luk and H. Park, On parallel Jacobi orderings, *SIAM J. Sci. Statist. Comput.* **10** 18–26 (1989).

[29] J. Matejaš and V. Hari, Scaled Iterates by Kogbetliantz Method, *Proceedings of the 1st Conference on Applied Mathematics and Computations, Dubrovnik, Croatia, September 13–18, 1999, Publisher Dept. of Mathematics, University of Zagreb*, 1–20 (2001).

[30] G. Okša and M. Vajteršic, Systolic Block-Jacobi SVD Algorithm for Processor Meshes, *Highly parallel computaions: algorithms and applications*, 211-235, WIT Press 2001, ISBN:1-85312-748-5.

[31] C.C. Paige and P. Van Dooren, On the Quadratic Convergence of Kogbetliantz's Algorithm for Computing the Singular Value Decomposition, *Linear Algebra and Its Applications* **77** 301–313 (1986).

[32] V. Zadelj-Martić, *Diagonalization Methods for Butterfly Matrices*, Master Thesis, University of Zagreb, Croatia, 1999 (in Croatian language).