Parallel One-Sided Block Jacobi SVD Algorithm: II. Implementation

Gabriel Okša[a] Marián Vajteršic

[a]Mathematical Institute, Department of Informatics, Slovak Academy of Sciences, Bratislava, Slovak Republic

Technical Report 2007-05 November 2007

Parallel One-Sided Block Jacobi SVD Algorithm: II. Implementation

Gabriel Okša[*] and Marián Vajteršic[†]

Abstract. *This technical report is devoted to the description of implementation details of the accelerated parallel one-sided block Jacobi SVD algorithm, whose analysis and design was described in [21]. We provide discuss a suitable data layout for a parallel implementation of the algorithm on a parallel computer with distributed memory. This discussion is complicated by the fact that different computational phases can have different optimal data layouts. We discuss in some detail the optimal data layout for the matrix pre-processing (QR factorization with column pivoting), and then our block-column-oriented data layout for the SVD computation. It turns out that the transition between both layouts is needed in the form of an (optimal) communication algorithm whose main features are described in detail. Another important issue is a communication-efficient adaptation of the dynamic ordering from the two-sided block Jacobi algorithm to the one-sided one.*

1 Introduction

This report is the second one in the on-going project for the analysis, design and implementation of the parallel one-sided block Jacobi algorithm (OSBJA) for the computation of the singular value decomposition (SVD) of a general matrix $A \in \mathbb{R}^{m\times n}$ on a parallel architecture. The first part [21] was devoted to the analysis and design using some new ideas for accelerating the (slow) Jacobi method and for enhancing its efficiency when working with matrix blocks and special matrix recursion.

This report is devoted to a description of some implementation details when considering the target architecture as a parallel machine with distributed memory (e.g., a cluster of personal computers). The emphasis is given to the optimal data layout for all computational phases. In our experience with the parallel *two-sided* block Jacobi SVD algorithm [20], the block-column-oriented data layout is not well suited for the pre-processing step, which consists of the QR decomposition with column pivoting. We analyse the reason why it is so and suggest another data layout - so called *block-cyclic* distribution of matrix blocks that is closely connected to a logical mesh of processors. However, for the SVD computation, the block-column-oriented data

[*]Mathematical Institute, Department of Informatics, Slovak Academy of Sciences, Bratislava, Slovak Republic, email: Gabriel.Oksa@savba.sk.

[†]Department of Computer Sciences, University of Salzburg, Salzburg, Austria, email: marian@cosy.sbg.ac.at.

layout is preferable, because the OBSJA is based on a mutual orthogonalization of columns between two different block columns. This means that an (optimal) communication algorithm is needed after the pre-processing step that will change the data distribution using a minimum amount of iter-processor communication. We discuss main features of such an algorithm.

Once the data layout is block-column-oriented, one needs to implement some block ordering strategy for mutual orthogonalization of pairs of block columns in such a way that all processors will be busy in each parallel iteration step. We discuss a possible generalization of our *dynamic* ordering (which has been designed for the two-sided block Jacobi method, see [4]) to the one-sided case. It turns out that this generalization can be done in two ways; however, the problem is the communication complexity of both generalizations, which is much higher than in the two-sided block Jacobi method.

This report should be read in close relationship with the first part in [21]. The first part is referred quite extensively; in particular, we do not repeat many details regarding the OBSJA and ideas leading to its acceleration and higher efficiency. The notation from [21] is also used throughout the whole report.

In section 2 we shortly repeat the main ideas about the accelerated OSBJA from [21]. Section 3 describes in details our parallelization strategy. Much attention is devoted to the design of an efficient communication algorithm for changing the data layout from the block-cyclic one to the block-column one, and to the extension of the dynamic ordering from the two-sided block Jacobi algorithm to the one-sided one.

2 Accelerated One-Sided Block-Jacobi Algorithm

The OSBJA is suited for the SVD computation of a general complex matrix A of order $m \times n$, $m \geq n$. However, we will restrict ourselves to real matrices with obvious modifications in the complex case.

We start with the block-column partitioning of A in the form

$$A = [A_1, A_2, \ldots, A_r],$$

where the width of A_i is n_i, $p \leq i \leq r$, so that $n_1 + n_2 + \cdots + n_r = n$. The most natural choice is $n_1 = n_2 = \cdots = n_{r-1} = n_0$, so that $n = (r-1)n_0 + n_r$, $n_r \leq n_0$. Here n_0 can be chosen according to the available cache memory, which is up to 10 times faster than the main memory; this connection will be clear later on.

The OSBJA can be written as an iterative process:

$$\begin{aligned} A^{(0)} &= A, \quad V^{(0)} = I_n, \\ A^{(k+1)} &= A^{(k)}U^{(k)}, \quad V^{(k+1)} = V^{(k)}U^{(k)}, \quad k \geq 0. \end{aligned} \tag{1}$$

Here the $n \times n$ orthogonal matrix $U^{(k)}$ is the so-called *block rotation* of the form

$$U^{(k)} = \begin{pmatrix} I & & & & \\ & U_{ii}^{(k)} & & U_{ij}^{(k)} & \\ & & I & & \\ & U_{ji}^{(k)} & & U_{jj}^{(k)} & \\ & & & & I \end{pmatrix}, \tag{2}$$

where the unidentified matrix blocks are zero. The purpose of matrix multiplication $A^{(k)}U^{(k)}$ in (1) is to mutually orthogonalize the columns between column-blocks i and j of $A^{(k)}$. The matrix blocks $U_{ii}^{(k)}$ and $U_{jj}^{(k)}$ are square of order n_i and n_j, respectively, while the first, middle and last identity matrix is of order $\sum_{s=1}^{i-1} n_s$, $\sum_{s=i+1}^{j-1} n_s$ and $\sum_{s=j+1}^{r} n_s$, respectively. The orthogonal matrix

$$\hat{U}^{(k)} = \begin{pmatrix} U_{ii}^{(k)} & U_{ij}^{(k)} \\ U_{ji}^{(k)} & U_{jj}^{(k)} \end{pmatrix} \tag{3}$$

of order $n_i + n_j$ is called the *pivot submatrix* of $U^{(k)}$ at step k. During the iterative process (1), two index functions are defined: $i = i(k)$, $j = j(k)$ whereby $1 \leq i < j \leq r$. At each step k of the OSBJA, the pivot pair (i, j) is chosen according to a given *pivot strategy* that can be identified with a function $\mathcal{F} : \{0, 1, \ldots\} \to \mathbf{P}_r = \{(l, m) : 1 \leq l < m \leq r\}$. If $\mathbf{O} = \{(l_1, m_1), (l_2, m_2), \ldots, (l_{N(r)}, m_{N(r)})\}$ is some ordering of $\mathbf{P}_r$ with $N(r) = r\,(r-1)/2$, then the *cyclic* strategy is defined by:

If $k \equiv r - 1 \mod N(r)$ then $(i(k), j(k)) = (l_s, m_s)$ for $1 \leq s \leq N(r)$.

The most common cyclic strategies are the *row-cyclic* one and the *column-cyclic* one, where the orderings are given row-wise and column-wise, respectively, with regard to the upper triangle of A. The first $N(r)$ iterations constitute the first *sweep* of the OSBJA. When the first sweep is completed, the pivot pairs (i, j) are repeated during the second sweep, and so on, up to the convergence of the entire algorithm.

Notice that in (1) only the matrix of right singular vectors $V^{(k)}$ is iteratively computed by orthogonal updates. If the process ends at iteration t, say, then $A^{(t)}$ has mutually highly orthogonal columns. Their norms are the singular values of A, and the normalized columns (with unit 2-norm) constitute the matrix of left singular vectors.

One (serial) step of the OSBJA can be described in three parts:

1. For the given pivot pair (i, j), the symmetric, positive semidefinite cross-product matrix is computed:

$$\hat{A}_{ij}^{(k)} = [A_i^{(k)}\ A_j^{(k)}]^T\,[A_i^{(k)}\ A_j^{(k)}] = \begin{pmatrix} A_i^{(k)T} A_i^{(k)} & A_i^{(k)T} A_j^{(k)} \\ A_j^{(k)T} A_i^{(k)} & A_j^{(k)T} A_j^{(k)}. \end{pmatrix} \tag{4}$$

2. $\hat{A}_{ij}^{(k)}$ is diagonalized, i.e., the eigenvalue decomposition of $\hat{A}_{ij}^{(k)}$ is computed:

$$\hat{U}^{(k)T}\,\hat{A}_{ij}^{(k)}\,\hat{U}^{(k)} = \hat{\Lambda}_{ij}^{(k)} \tag{5}$$

and the eigenvector matrix $\hat{U}^{(k)}$ is partitioned according to (3). The matrix $\hat{U}^{(k)}$ defines the orthogonal transformation $U^{(k)}$ in (2) and (1), which is then applied to $A^{(k)}$ and $V^{(k)}$. Notice that the explicit diagonalization of $\hat{A}^{(k)}$ is equivalent to the implicit mutual orthogonalization of columns between column blocks i and j in $A^{(k)}$, i.e., in $(A_i^{(k)}, A_j^{(k)})$.

3. Finally, an updating of two block-columns of $A^{(k)}$ and $V^{(k)}$ is required.

2.1 Matrix preprocessing

It is well known that the one- or two-sided Jacobi method can be efficiently preprocessed by the QR factorization of A (usually with the complete column pivoting) followed by the LQ factorization of R-factor; see [7, 8, 9, 20]. The Jacobi method is then applied to the final L-factor. This leads to a strong reduction of the total number of Jacobi steps, including a strong decrease in the number of orthogonal updates of the matrix $V^{(k)}$ of right singular vectors in (1).

The second preprocessing step initializes certain three matrices (see details in 2.1.1), which are then iterated during the Jacobi process. Here is the connection with the implementation of the fast scaled block-orthogonal transformations mentioned above. It also makes all columns within each column block mutually orthogonal, whereas this property remains invariant during the whole computation, so that diagonal blocks of the cross-product matrix $\hat{A}_{ij}^{(k)}$ in (4) are themselves diagonal. Consequently, at step k, the the columns need to be orthogonalized only *between* two block columns $A_i^{(k)}$ and $A_j^{(k)}$ (not *within* them).

Mathematical details regarding both preprocessing steps can be found in [21].

2.1.1 Initialization

Recall that once the diagonalization in (5) is performed over all block columns of A, then the diagonal blocks in each cross-product matrix $\hat{A}_{ij}^{(k)}$ are themselves diagonal. Hence, it is not necessary to compute their elements except of the diagonal ones. This computation can be arranged into recursion. Let

$$\Gamma^{(k)} = \mathrm{diag}(\Gamma_1^{(k)}, \ldots, \Gamma_r^{(k)}) = \mathrm{diag}(\hat{A}^{(k)}) \quad \text{with} \quad \hat{A}^{(k)} = A^{(k)T} A^{(k)},$$

where $\Gamma_1^{(k)}, \ldots, \Gamma_r^{(k)}$ is the partition inherited from the block-column partition of $A^{(k)}$. At step k, the diagonal of $\hat{A}_{ij}^{(k)}$, which is equal to $\mathrm{diag}(\Gamma_i^{(k)}, \Gamma_j^{(k)})$, is transformed and written to $\hat{\Lambda}_{ij}^{(k)}$. Hence, $\hat{\Lambda}_{ij}^{(k)} = \mathrm{diag}(\Gamma_i^{(k+1)}, \Gamma_j^{(k+1)})$, so that $\Gamma^{(k)}$ (represented in a computer by the vector $\gamma^{(k)}$) can be updated very simply (once we have the eigendecomposition of the cross-product matrix in (5) and in parallel to updating $A^{(k)}$. To initialize the computation, we apply the following algorithm after the QR and LQ decompositions to the L-factor $L = [L_1, L_2, \ldots, L_r]$:

```
1. for $i = 1 : r$
2.    $\hat{A}_{ii}^{(0)} = L_i^T L_i$;
3.    $\hat{A}_{ii}^{(0)} = Q_{ii}^{(0)} \Gamma_i^{(0)} Q_{ii}^{(0)T}$; (spectral decomposition)
```

```
4.    ($A_i^{(0)} = L_i Q_{ii}^{(0)}$); (not performed, just an illustration of the connection)
5. end;
```

Thus the above algorithm initializes three important matrices:

$$\begin{aligned} B^{(0)} &= [B_1^{(0)}, B_2^{(0)}, \dots, B_r^{(0)}] = [L_1, L_2, \dots, L_r], \\ Q^{(0)} &= \operatorname{diag}(Q_{11}^{(0)}, Q_{22}^{(0)}, \dots, Q_{rr}^{(0)}), \\ \Gamma^{(0)} &= \operatorname{diag}(\Gamma_1^{(0)}, \Gamma_2^{(0)}, \dots, \Gamma_r^{(0)}). \end{aligned}$$

2.2 Fast scaled block-orthogonal transformations

Now we need to find recursions for the computation of the matrix triplet $B^{(k)}$, $Q^{(k)}$ and $\Gamma^{(k)}$ at step k of the Jacobi process. The main idea here is to use small matrices of order n_i, n_j or $n_i \times n_j$ for all updates (computed as matrix multiplications), so that these updates can be done in the fast cache memory; see [12].

Let us assume that at step k we have $B^{(k)}$, $Q^{(k)}$ and $\Gamma^{(k)}$ fulfilling. Then, according to (4), we need to compute the cross-product matrix $\hat{A}_{ij}^{(k)}$ for the given pivot pair (i, j):

$$\begin{aligned} \hat{A}_{ij}^{(k)} &= \begin{pmatrix} Q_{ii}^{(k)} & \\ & Q_{jj}^{(k)} \end{pmatrix}^T \begin{pmatrix} B_i^{(k)T} B_i^{(k)} & B_i^{(k)T} B_j^{(k)} \\ B_j^{(k)T} B_i^{(k)} & B_j^{(k)T} B_j^{(k)} \end{pmatrix} \begin{pmatrix} Q_{ii}^{(k)} & \\ & Q_{jj}^{(k)} \end{pmatrix} \\ &= \begin{pmatrix} \Gamma_i^{(k)} & \tilde{A}_{ij}^{(k)} \\ \tilde{A}_{ij}^{(k)T} & \Gamma_j^{(k)} \end{pmatrix}, \text{ where } \tilde{A}_{ij}^{(k)} \equiv Q_{ii}^{(k)T} (B_i^{(k)T} B_j^{(k)}) Q_{jj}^{(k)}. \end{aligned} \tag{6}$$

Next, we compute the eigendecomposition of $\hat{A}_{ij}^{(k)}$ according to (5). Having the orthogonal eigenvector matrix $\hat{U}^{(k)}$, Hari [12] proposed to compute its cosine-sine (CS) decomposition

$$\begin{aligned} \hat{U}^{(k)} &= \begin{pmatrix} V_{ii}^{(k)} & \\ & V_{jj}^{(k)} \end{pmatrix} \begin{pmatrix} C_{ii}^{(k)} & -S_{ij}^{(k)} \\ S_{ji}^{(k)} & C_{jj}^{(k)} \end{pmatrix} \begin{pmatrix} W_{ii}^{(k)} & \\ & W_{jj}^{(k)} \end{pmatrix}^T \\ &\equiv \hat{V}^{(k)} \hat{T}^{(k)} \hat{W}^{(k)T}, \end{aligned} \tag{7}$$

where the matrix blocks $V_{ii}^{(k)}$, $C_{ii}^{(k)}$, $W_{ii}^{(k)}$ ($V_{jj}^{(k)}$, $C_{jj}^{(k)}$, $W_{jj}^{(k)}$) are square of order n_i (n_j), and

$$\hat{T}^{(k)} = \begin{pmatrix} C_{ii}^{(k)} & -S_{ij}^{(k)} \\ S_{ji}^{(k)} & C_{jj}^{(k)} \end{pmatrix} = \begin{cases} \begin{pmatrix} I_{n_i - n_j} & 0 & 0 \\ 0 & C^{(k)} & -S^{(k)} \\ 0 & S^{(k)} & C^{(k)} \end{pmatrix}, & \text{if } n_i \geq n_j, \\ \begin{pmatrix} C^{(k)} & 0 & -S^{(k)} \\ 0 & I_{n_j - n_i} & 0 \\ S^{(k)} & 0 & C^{(k)} \end{pmatrix}, & \text{if } n_j \geq n_i, \end{cases} \tag{8}$$

and

$$\begin{gathered} C^{(k)} = \operatorname{diag}(c_1^{(k)}, \dots, c_{\nu_{ij}}^{(k)}), \quad S^{(k)} = \operatorname{diag}(s_1^{(k)}, \dots, s_{\nu_{ij}}^{(k)}), \\ c_1^{(k)} \geq c_2^{(k)} \geq \cdots \geq c_{\nu_{ij}}^{(k)} \geq 0, \quad 0 \leq s_1^{(k)} \leq s_2^{(k)} \leq \cdots \leq s_{\nu_{ij}}^{(k)}, \\ (c_r^{(k)})^2 + (s_r^{(k)})^2 = 1, \quad 1 \leq r \leq \nu_{ij}, \quad \nu_{ij} = \min\{n_i, n_j\}. \end{gathered}$$

Next step in the OSBJA is the multiplication of the pivot block-column matrix $(A_i^{(k)}, A_j^{(k)})$ by $\hat{U}^k$ from the left (see (1)) to get the new iteration $(A_i^{(k+1)}, A_j^{(k+1)})$. This can be written in the factored form:

$$\begin{aligned}(B_i^{(k+1)}Q_{ii}^{(k+1)}, B_j^{(k+1)}Q_{jj}^{(k+1)}) &= (B_i^{(k)}Q_{ii}^{(k)}, B_j^{(k)}Q_{jj}^{(k)})\,\hat{V}^{(k)}\,\hat{T}^{(k)}\,\hat{W}^{(k)T} \\ &= (B_i^{(k)}(Q_{ii}^{(k)}V_{ii}^{(k)}), B_j^{(k)}(Q_{jj}^{(k)}V_{jj}^{(k)}))\,\hat{T}^{(k)}\,\mathrm{diag}(W_{ii}^{(k)}, W_{jj}^{(k)})^T,\end{aligned}$$

which leads immediately to a recursion for matrices B and Q:

$$\begin{gathered}(B_i^{(k+1)}, B_j^{(k+1)}) = (B_i^{(k)}(Q_{ii}^{(k)}V_{ii}^{(k)}), B_j^{(k)}(Q_{jj}^{(k)}V_{jj}^{(k)})\,\hat{T}^{(k)}, \\ Q_{ii}^{(k+1)} = W_{ii}^{(k)T}, \quad Q_{jj}^{(k+1)} = W_{jj}^{(k)T}.\end{gathered} \tag{9}$$

(Recall that the new $\Gamma^{(k+1)}$ is obtained simply by copying $n_i + n_j$ eigenvalues from $\hat{\Lambda}_{ij}^{(k)}$ to appropriate places of $\Gamma^{(k)}$.) It is immediately seen from (9) that the original number of flops required for updating in (1) is significantly reduced using the new recursion. First, in the computation of $(Q_{ii}^{(k)}V_{ii}^{(k)})$ and $(Q_{jj}^{(k)}V_{jj}^{(k)})$, only the small dimensions n_i and n_j are involved. Second, once these two matrix multiplications are computed, the update of B_i and B_j requires the matrix multiplication of the form XY, where X is of order $n \times n_i$ or $n \times n_j$, and Y is square of order n_i or n_j. The final update of B_i and B_j requires the matrix multiplication by $\hat{T}^{(k)}$ from the left, which is equivalent, due to the special structure of $\hat{T}^{(k)}$, to simple rotations of columns of length n. Notice that we have eliminated the dimension $m \gg n$, which is the main source of inefficient updating of original A in (1). The price paid is the recursion of three matrices, where two of them are updated by simple copying of elements. The main idea in this auxiliary recursion exploits the fact that the dimensions of blocks can be chosen so that all computations in this phase can be done in fast (cache) memory.

3 Parallel Implementation

Next we describe main ideas behind the parallelization of one-sided block-Jacobi SVD method. We are interested in the parallelization of the above algorithm assuming the *distributed* paradigm of parallel processing. In particular, we would like to implement the OSBJA using the Message Passing Interface (MPI) and BLACS libraries for communication, and the ScaLAPACK library for a distributed computation. The serial computation inside each processor can be performed using the standard LAPACK library.

We start with discussion of data layout for the QR decomposition and continue with changing this data layout for the SVD computation. Then we describe possible extensions of our dynamic block ordering from the two-sided block Jacobi method to the one-sided one. Afterwards, we shortly discuss a possible parallel implementation of all phases of the algorithm.

3.1 Data layout for the parallel QR decomposition

Numerical experiments with the two-sided block Jacobi SVD algorithm with preprocessing in [20] have shown that the block-column data distribution (see Fig. 3) is not well suited for the preprocessing step, which consists of the QR decomposition of an original matrix with column pivoting followed by an optional QR decomposition of the R-factor. We reproduce here Fig. 1, from which it is clear that the time complexity of the preprocessing step dominates is at least 30 per cent, but for matrices of order 6000 this number increases to 75 per cent! Taking

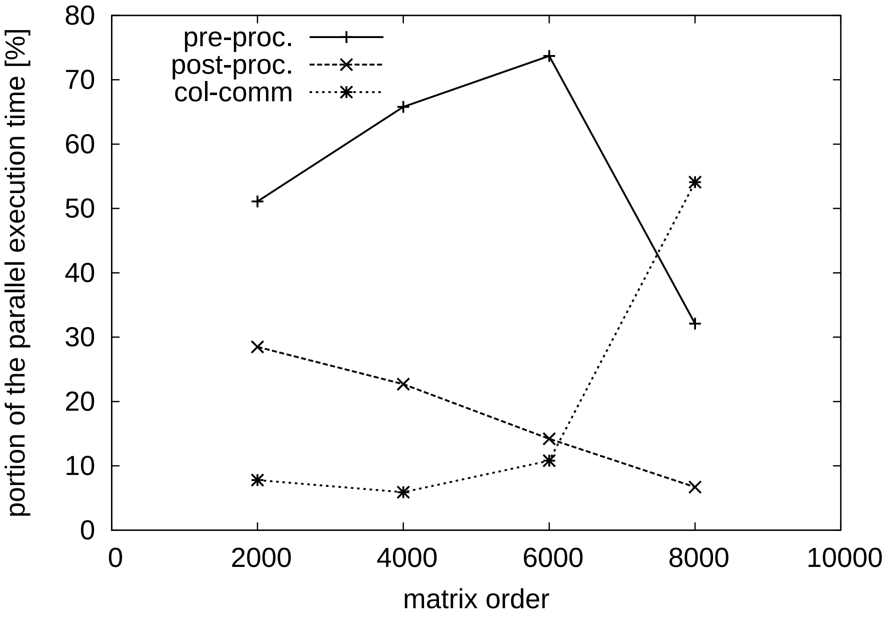

Figure 1: Portion of the total parallel execution time for the pre-processing (the QR decomposition), collective communication and post-processing (matrix-matrix multiplication) in the two-sided parallel block Jacobi SVD algorithm. The horizontal axis is the matrix dimension.

into account that the preprocessing step is done only at most twice at the beginning of the iterative process, its portion of the total parallel execution time is huge and should be somehow decreased.

The reason of this inefficiency is the implicit “serialization” of the QR decomposition. Although the data is distributed in a block-column manner among processors, the ScaLAPACK procedure `PDGEQRF` with column pivoting proceeds columnwise from the first column (the leftmost one) to the last column (the rightmost one). After a possible exchange of columns by column pivoting, the actual column is reduced to the triangular form and only after that next column is processed (again after a possible column exchange by column pivoting). Hence, when a matrix is distributed by block columns among processors, then the column reduction is made in one processor (which is efficient), but after finishing the reduction of all columns residing in a given processor that processor becomes idle for the rest of the QR decomposition (which is highly inefficient).

It is better to keep all processors busy during the whole QR decomposition regardless to which column is actually processed. This can be achieved by the *block-cyclic* matrix distribution among processors; one example of it is depicted in Fig. 2. The main idea is to keep busy as many processors as possible during the triangular reduction of *any* matrix column. There are two parameters which control the efficiency of the distributed QR decomposition. The first one is the number of processors p and their arrangement into a logical (rectangular) mesh of size $u \times v$ where $p = uv$. The second one is the blocking factor r_1 and r_2 in the row and

P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2
P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5
P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8
P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2
P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5
P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8
P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2
P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5
P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8
P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2
P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5
P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8
P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2
P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5
P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8
P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2	P0	P1	P2
P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5	P3	P4	P5
P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8	P6	P7	P8

Figure 2: Block-cyclic matrix distribution for the parallel QR decomposition with the square grid of $p = \sqrt{p} \times \sqrt{p}$ processors and blocking factor $r = 2p$. Here $p = 9$, $r = 18$ and the logical square grid of processors is enhanced in the left upper corner of the mesh.

column direction, respectively, so that, for a matrix A of size $m \times n$, the matrix blocks are of size $m/r_1 \times n/r_2$. Notice that the block-cyclic data distribution is *perfectly balanced* (i.e., each processor contains the same amount of data) if and only if m/r_1 and n/r_2 are both integers, and u divides r_1 as well as v divides r_2.

On Fig. 2, a special configuration of the processor mesh and blocking factor is depicted. The processor mesh is square of size $\sqrt{p} \times \sqrt{p}$ (i.e., one has together p processors, and the blocking factor is the same in both directions: $r = r_1 = r_2 = 2p$. The size of the blocking factor was chosen with respect to the data distribution used for the Jacobi iterations in the SVD computations (see the next subsection), since each processor will have exactly two block columns in that computational phase of the algorithm.

So, at the beginning of computation we have an original matrix A distributed in a block-cyclic fashion. Then the first preprocessing step is needed. We compute the QR factorization of A with column pivoting (QRFCP) followed by an optional LQ factorization (LQF) of the R-factor. In particular, the QRFCP and the LQF can be implemented by the ScaLAPACK routine `PDGEQPF` and `PDGELQF`, respectively. Note that also the results of the QRFCP (and LQF)–i.e., the matrices Q and R–will be distributed in the same block-cyclic fashion than A.

3.2 Data layout for the parallel SVD computation

After the first preprocessing step, the data layout is changed from the block-cyclic one to the block-column one (compare Fig. 2 with Fig. 3). All subsequent computations (i.e., the

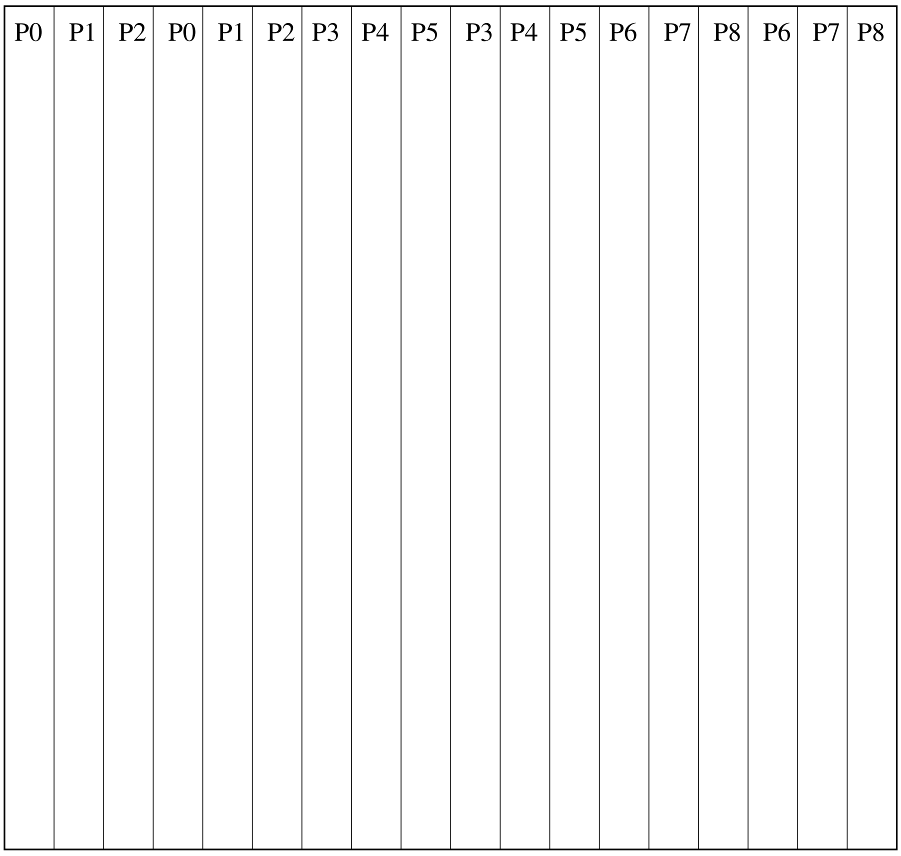

Figure 3: Block-column matrix distribution for the parallel SVD computation with $p = 1 \times p$ processors and blocking factor $r = 2p$, here with $p = 9$, $r = 18$. Notice that block columns are not assigned consecutively to processors; this special permutation minimizes the communication complexity when going from the block-cyclic to block-column data layout. Each processor contains two block columns.

initialization of a three-matrix recursion as well as the Jacobi iteration process) will be done with this new data layout, which has proven to be very efficient in our experiments with the two-sided block Jacobi method; see a discussion of numerical results in [2, 3, 4, 20].

To achieve the change of data distribution, a special communication algorithm is needed which will change the data layout. Notice that this 'data re-configuration step' is needed only once in the whole computation, because the parallel QR decomposition is never repeated inside the Jacobi iterations. Nevertheless, to achieve a good performance (measured by the total parallel execution time) and a good scalability (measured by the dependence of the total parallel execution time on the problem's size), a communication algorithm should be optimal in the sense of minimizing the amount of serialized communication steps and transferred data.

This special communication algorithm has not been designed yet in all details. However, some observations help to guess its main features for square grids of $\sqrt{p} \times \sqrt{p}$ processors that cover completely the partition of matrix A:

1. Starting with the block-cyclic distribution (Fig. 2), notice that any fixed block column is distributed only among $\sqrt{p}$ processors.

2. Any two block columns exactly $\sqrt{p}$ apart are distributed among the same processors.

3. Hence, using both properties mentioned above, it would *not* be efficient to build up the block-column partition for the SVD computation by assigning the block columns to processors consecutively (i.e., two first block columns for processor $P0$, next two block columns for processor $P1$, etc.). It is easy to see that such a consecutive assignment means that any given processor Pj has to communicate with either $2\sqrt{p}-1$ other processors (in the case when Pj appears in one of those two consecutive block columns assigned to it), or $2\sqrt{p}$ other processors have to send their data to Pj (when Pj does not appear in any of those two consecutive block columns assigned to it). Both cases can be demonstrated on Fig. 2. For example, let first two block columns be assigned to processor $P0$ and next two block columns to processor $P1$. Since $P0$ appears in block column 1 but it does not appear in block column 2, it has to communicate with $2\sqrt{p}-1$ other processors–namely, $P3$, $P6$, $P1$, $P4$ and $P7$. But processor $P1$ appears neither in the third nor in the fourth block column, so that all $2\sqrt{p}$ processors–namely, $P0$, $P2$, $P3$, $P5$, $P6$ and $P8$–must send their data to $P1$.

4. The difference between above two cases is not only in the number of communicated processors (this number differs only by one), but also in the amount of transferred matrix blocks. In the first case, exactly $2\sqrt{p}$ matrix blocks of R and Q each are already assigned to the target processor, so that only $4p-2\sqrt{p}=2\sqrt{p}(2\sqrt{p}-1)$ matrix blocks of R and Q each have to be transferred from other processors. In the second case, all $4p$ matrix blocks have to be transferred. When p is large, the first approach can be much more efficient than the second one.

5. However, there is another way how to achieve even less communication complexity. Since, as mentioned above, any two block columns exactly $\sqrt{p}$ apart are distributed among the same processors, we can assign to a given processor Pj exactly one pair of such block columns. Thus using Fig. 3, processor $P0$ will contain block columns $(1,4)$, processor $P1$ block columns $(2,5)$, processor $P2$ block columns $(3,6)$, processor $P3$ block columns $(7,10)$, processor $P4$ block columns $(8,11)$, processor $P5$ block columns $(9,12)$, processor $P6$ block columns $(13,16)$, processor $P7$ block columns $(14,17)$ and processor $P8$ block columns $(15,18)$. In general, for the square grid of $\sqrt{p}\times\sqrt{p}$ processors numbered from left to right and from top to bottom by $0,\ 1,\cdots,\ p-1$, and for the blocking factor $r=2p$, processor Pj with index $j=k\sqrt{p}+\ell,\ 0\leq k\leq\sqrt{p}-1,\ 0\leq\ell\leq\sqrt{p}-1$, will contain block columns $(2k\sqrt{p}+\ell+1, 2(k+1)\sqrt{p}+\ell+1)$. Hence, after reshaping the data layout from the block-cyclic one to the block-column one, the Jacobi process will start with the *specially permuted* matrices Q and R (L). But the amount of communication is further decreased, because each processor will have exactly $4\sqrt{p}$ matrix blocks of R and Q each in place, so that only $4\sqrt{p}(\sqrt{p}-1)$ matrix blocks of R and Q each have to be received from other processors. Moreover, each processor has to communicate only with $\sqrt{p}-1$ other processors, which is halved in comparison with both cases mentioned above.

It is easy to see that for a square grid of p processors covering completely the block partition in a perfectly balanced manner, the communication complexity of the specially permuted case mentioned as the last possibility above is optimal (minimal). A (small) disadvantage is that we will start the Jacobi iterations with block-column permuted matrices Q and R (the Jacobi process needs only the matrix R; the matrix Q is needed in the post-processing). However, during the Jacobi iterations, the parallel block ordering itself causes permutations of block columns, and the final assembling of matrices has to be done according to a final permutation describing, say, a decreasing sequence of singular values. Hence, the first permutation of block columns made during a transition from the block-cyclic to block-column data layout represents no difficulty.

3.3 Dynamic ordering in the one-sided algorithm

At the beginning of each parallel iteration step it is necessary to choose $p = r/2$ pivot pairs (i, j) that define, for p processors, p subtasks that can be computed in parallel. This means to assign one pivot pair per one processor, and to move (at most) two block columns with block indices equal to the pivot pair to that processor. In other words, we need to design a proper *parallel block ordering.*

In the past, the parallel orderings were designed mostly for the scalar Jacobi method and perhaps the best discussion is provided in [16]. In those days,some 20 years ago, the emphasis was given to the requirement that the processors should exchange their elements on the nearest-neighbor basis, and the amount of communicated data should be minimized. Today, working with modern parallel architectures, the requirement of the nearest neighbor communication is not so important, whereas it is still useful to keep the amount of exchanged data at minimum due to the start-up time and transfer time per one double variable needed for the synchronous/asynchronous data transfer, which can be several orders of magnitude larger than that for computation.

Luk and Park [16] analyzed the caterpillar-track and caterpillar-tractor orderings, odd-even ordering, round-robin ordering. They showed that they are equivalent for n odd or n even (n is the matrix order). However, the main disadvantage of these parallel orderings (with exception of the round-robin ordering) is the low exploitation of the computational power: only at each second stage there are $n/2$ parallel rotations, which 'cover' all $n/2$ processors (for simplicity, we take here n even). The round-robin parallel ordering is optimal: for n even, *each* stage consists of exactly $n/2$ parallel rotations, which can be implemented exactly on $n/2$ processors. Unfortunately, the convergence of the Jacobi method with the parallel round-robin ordering is not guaranteed for n even. As was shown in [17], there exists a matrix of even order (albeit with a very special structure), for which, when applying the one-sided Jacobi SVD algorithm with the round-robin ordering, its off-diagonal norm does not converge to zero (it stagnates).

All above mentioned parallel scalar orderings can be easily and directly extended to the block case. Recall that our blocking factor $r = 2p$ is even (p is the number of processors). With respect to the convergence of parallel block-Jacobi SVD algorithms, the actual situation can be described as 'terra incognita'. We know only one paper [13], which proves the global convergence of a *serial* block-oriented quasi-cyclic Jacobi method for symmetric matrices. To our best knowledge, there are no global convergence results for any *parallel* block-Jacobi method. Therefore, we should try the block version of the most-efficient scalar parallel ordering—namely, the round-robin ordering and conduct extensive numerical experiments. Alternatively, we could try to design a communication-efficient version of the *dynamic* ordering [4].

The dynamic ordering is based on a complete weighted graph with $r = 2p$ vertices–hence the number of vertices is equal to the blocking factor; see Fig. 4. In the two-sided block Jacobi

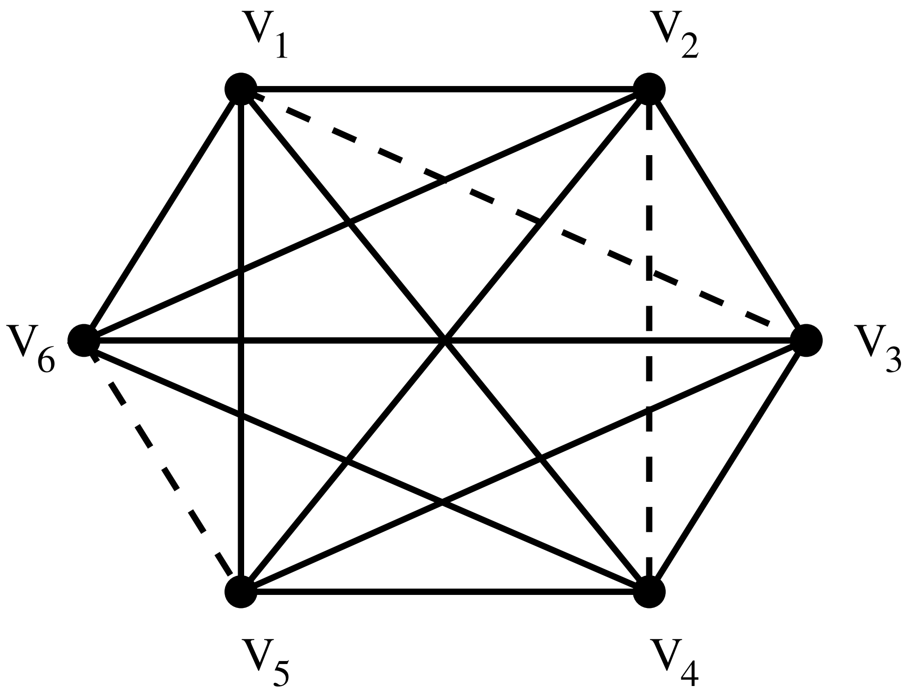

Figure 4: Maximum-weight perfect matching on a complete graph for $r = 6$. The chosen edges are dashed.

method, each edge is weighted by the non-negative weight $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$, where $\|A_{uv}\|_F^2$ is the square of the Frobenius norm of matrix block A_{uv}. Recall that the convergence of the two-sided block Jacobi algorithm is based on the convergence of the off-diagonal Frobenius norm of matrix A to zero. Hence, the purpose is to choose, at the beginning of each parallel iteration step, those block pairs (A_{ij}, A_{ji}) that would decrease (after their zeroing) the off-diagonal norm as much as possible. Moreover, we need $r/2$ disjunct pairs, one per processor. This task is equivalent to finding a *maximum-weight perfect matching* on a complete graph (see Fig. 4). It is known that there exist the optimal polynomial algorithm for this task and we have designed the suboptimal polynomial algorithm in [4].

Our experiments with the two-sided block Jacobi SVD algorithm have shown that the ordering algorithm is very efficient, and although it runs at the beginning of each parallel iteration step, it takes only some 5 per cent of the total parallel execution time for matrices of order 10^4. The reason of this efficiency lies in the fact that Frobenius norms (or their squares) of individual matrix blocks can be easily computed *locally* within processors (since each processor stores exactly 2 block columns, it can locally compute the square of Forbenius norms for $2r$ matrix blocks). Then, these Frobenius norms are centralized in processor $P0$, sorted in decreasing order and the maximum-weight perfect matching is found. The result is then broadcast to all processors, and after assembling chosen pairs of matrix blocks in individual processors, the next

p parallel SVD computations can start. Therefore, from a communication point of view, the finding of a maximum-perfect matching costs only one `MPI_ALLGATHER` and one `MPI_BROADCAST`.

Unfortunately, the situation is more complicated in case of the one-sided block Jacobi algorithm, which is based on a *mutual orthogonalization* of two different block columns in one processor. (Recall that after initialization the columns *within* each block column are mutually orthogonal and remain so during the whole iteration process.) Now, the principle of the maximum-weight perfect matching can be easily extended also to the paradigm of mutual orthogonality of block columns. An ideal case is the mutual orthogonality of *all* pairs of block columns. The departure from this ideal case can be measured either by a sum of squares of cosines of angles between all pairs of columns in two given block columns, or by the maximum cosine of these angles. Hence, for each pair of block columns (A_i, A_j), we can define the departure from their mutual orthogonality by

$$w_{ij} = \sum_{u,v=1}^{n/r} \cos^2 \angle(a_u^{(i)}, a_v^{(j)}) \quad \text{or} \quad w_{ij} = \max_{1\leq u,v\leq n/r} \{\cos^2 \angle(a_u^{(i)}, a_v^{(j)})\}, \tag{10}$$

where $a_t^{(k)}$ is the t-th column of the matrix block A_k (for simplicity, we have omitted the iteration index). The number w_{ij} is then the weight in the complete graph between vertices i and j. Note that computation of weights w_{ij} for one pair of block columns (A_i, A_j) requires $(n/r)^2$ scalar products, each of length m. Then the result of a maximum-perfect matching means to choose those p pairs of matrix blocks for which the sum of departures from mutual orthogonality is maximum. This is highly desirable because in orthogonalizing the block columns we prefer to work precisely with those pairs, which depart a lot from their mutual orthogonality.

But, in contrast to the two-sided block Jacobi method, the weights defined in (10) can *not* be updated locally (inside processors). At the end of a parallel iteration step, each processor contains two mutually orthogonal block columns, so we know which p weights in (10) are zero. However, the angles between any two columns residing in two *different* processors could have changed. To see how much, we have to compute the cosine of angle between them. In other words, we need to organize the update of cosines and weights in (10) in such a way that each matrix block column A_i must meet each matrix block column A_j, $j \neq i$, in some processor, in which the updated weight w_{ij} is computed according to (10). Therefore, the updating of weights in case of the one-sided block Jacobi method is much more communication-demanding than is was for the two-sided block Jacobi method. Since this updating is needed at the beginning of each parallel iteration step, it is of crucial importance to design an efficient strategy how to minimize the communication complexity of this subtask. As for now, this question remains open.

3.4 Computations with the block-column data layout

The second preprocessing consists of the initialization that computes a spectral decomposition of p diagonal blocks of the cross-product matrix $\hat{A}^{(0)} = L^T L$ (when using two factorizations in the preprocessing); see those 5 steps at the beginning of section 2.1.1. This means that each processor that stores two block columns i and j will compute serially exactly two cross-products $\hat{A}_{ll}^{(0)} = L_l^T L_l$, $l = i, j$ and then, again serially, two spectral decompositions of two symmetric, positive definite matrices $\hat{A}_{ll}^{(0)}$. Recall that we need to preserve a high relative accuracy, so that these spectral decompositions can be computed, e.g., by the Kogbetliantz method.

When two block columns are assigned to each processor, all computations in the modified algorithm are performed in parallel for $p = r/2$ subtasks. No inter-processor communication of any kind is needed during this computation, because all computations and updates are local. Recall that assigning a pivot pair (i, j) to a processor actually means (in the worst case) the transfer of matrix blocks B_i, B_j, Q_{ii}, Q_{jj} and vectors γ_i and γ_j to that processor.

In contrast to local computations in the modified algorithm, the implementation of the stopping criterion (see [21]) requires some sort of global communication between processors. The update of ω^2 and ν (see [21]) requires the local computation of the squared Frobenius norm of each nullified matrix block in each processor, then the global sum of local squares and, finally, the broadcast of an updated value to all processors. This can be implemented using routines `MPI_ALLREDUCE` and `MPI_ALLGATHER` from the ScaLAPACK. The computation of α is even more complex, because one needs to scale the columns and rows of B by the values stored in vector γ. This means that all elements of vector γ must be known to all processors (the routine `MPI_ALLGATHERV`), and, after local scaling, the Frobenius norm of a whole scaled matrix must be computed from the local Frobenius norms (routines `MPI_ALLREDUCE` and `MPI_ALLGATHER`).

4 Conclusions

We have designed a possible parallelization strategy for the accelerated one-sided block-Jacobi SVD algorithm on a parallel architecture with distributed memory. New ideas include the transition from the block-cyclic data layout, which is efficient for the parallel QR decomposition with or without column pivoting, to the block-column data layout, which is desirable for the Jacobi iteration process. Another new approach is the possible extension of the dynamic ordering from the two-sided block Jacobi algorithm to its one-sided version. However, here it is important to minimize the communication complexity at the beginning of each parallel iteration step, which is an open problem.

References

[1] A. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen, *LAPACK Users' Guide*, Second ed., SIAM, Philadelphia, 1999.

[2] M. Bečka and M. Vajteršic, *Block-Jacobi SVD algorithms for distributed memory systems: I. Hypercubes and rings*, Parallel Algorithms Appl. 13 (1999) 265-287.

[3] M. Bečka and M. Vajteršic, *Block-Jacobi SVD algorithms for distributed memory systems: II. Meshes*, Parallel Algorithms Appl. 14 (1999) 37-56.

[4] M. Bečka, G. Okša and M. Vajteršic, *Dynamic ordering for a parallel block-Jacobi SVD algorithm*, Parallel Computing 28 (2002) 243-262.

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

[6] Z. Drmač, *Implementation of Jacobi rotations for accurate singular value computation in floating-point arithmetic*, SIAM J. Sci. Comp. 18 (1997) 1200-1222.

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

[8] Z. Drmač and K. Veselić, *New fast and accurate Jacobi SVD algorithm: I.*, LAPACK Working Note 169, August 2005.

[9] Z. Drmač and K. Veselić, *New fast and accurate Jacobi SVD algorithm: II.*, LAPACK Working Note 170, August 2005.

[10] V. Hari and J. Matejaš, *Accuracy of the Kogbetliantz method*, preprint, University of Zagreb, 2005.

[11] V. Hari and V. Zadelj-Martič, *Parallelizing Kogbetliantz method*, accepted for publication at Int. Conf. on Numerical Analysis and Scientific Computation, Rhodos, Greece, September 2006.

[12] V. Hari, *Accelerating the SVD block-Jacobi method*, Computing 75 (2005) 27-53.

[13] V. Hari, *Convergence of a block-oriented quasi-cyclic Jacobi method*, accepted for publication in SIAM J. Matrix Anal. Appl.

[14] E. Kogbetliantz, *Diagonalization of general complex matrices as a new method for solution of linear equations*, Proc. Intern. Congr. Math. Amsterdam 2 (1954) 356-357.

[15] E. Kogbetliantz, *Solutions of linear equations by diagonalization of coefficient matrices*, Quart. Appl. Math. 13 (1955) 123-132.

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

[17] F. T. Luk and H. Park, *A proof of convergence for two parallel Jacobi SVD algorithms*, IEEE Trans. Comp. 38 (1989) 806-811.

[18] W. Mascarenhas, *On the convergence of the Jacobi method*, poster presentation, 4th SIAM Conference on Parallel Processing for Scientific Computing, Chicago, USA, December 1989.

[19] J. Matejaš, *Convergence of scaled iterates by Jacobi method*, Lin. Alg. Appl. 349 (2002) 17-53.

[20] G. Okša and M. Vajteršic, *Efficient preprocessing in the parallel block-Jacobi SVD algorithm*, Parallel Computing 31 (2005) 166-176.

[21] G. Okša and M. Vajteršic, *Parallel one-sided block Jacobi SVD algorithm: I. Analysis and design*, Technical report, Salzburg University, Salzburg, Austria, June 2007.

[22] P. M. de Rijk, *A one-sided Jacobi algorithm for computing the singular value decomposition on a vector computer*, SIAM J. Sci. Stat. Comp. 10 (1989) 359-371.

[23] K. Veselić and V. Hari, *A note on a one-sided Jacobi algorithm*, Numer. Math. 56 (1989) 627-633.