UNIVERSITÄT SALZBURG

Parallel One-Sided Block Jacobi SVD Algorithm: I. Analysis and Design

Gabriel Okša[a] Marián Vajteršic

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

Technical Report 2007-02 June 2007

Department of Computer Sciences

Jakob-Haringer-Straße 2
5020 Salzburg
Austria
www.cosy.sbg.ac.at

Technical Report Series

Parallel One-Sided Block Jacobi SVD Algorithm: I. Analysis and Design

Gabriel Okša* and Marián Vajteršic†

Abstract. *The computation of a singular value decomposition of an $m \times n$ matrix A is certainly one of the most often demanded tasks in various applications. There are many algorithms for computing the full or partial singular value decomposition. Among them, the one-sided Jacobi method (coupled with some orderings) is reputable for its ability to compute the singular values as well as left and right singular vectors with high relative accuracy. This is important, for example, in applications like quantum physics or chemistry, where the atomic and/or molecular energies of tiny values have to be computed very accurately (these energies are modeled as the eigenvalues of symmetric operators, thus they equal to singular values). Unfortunately, the Jacobi method belongs also to the slowest algorithms, and as such has been almost abandoned. Recently, some new ideas for accelerating the one-sided serial Jacobi algorithm were presented and implemented. Numerical experiments have shown that the modified Jacobi algorithm is as fast as the QR algorithm and slightly slower than the divide-and-conquer one. We describe in detail main ideas of an acceleration, namely, working with matrix blocks rather than elements, the preprocessing of an original matrix, the special initialization procedure, the new matrix recursion and the sine-cosine decomposition of certain matrix blocks. The possible parallelization strategy for the one-sided block-Jacobi algorithm is also discussed.*

1 Introduction

The one-sided Jacobi methods for computing the singular value decomposition (SVD) of a rectangular matrix are more efficient than their two-sided counterparts mainly due to the halving of the number of matrix-matrix products needed for updating the left and right singular vectors; see [1, 21, 22] Moreover, some of them were also proved to be accurate in the relative sense and at least as accurate as the two-sided Jacobi methods; see [5, 6, 7, 8, 9, 22]. When the preprocessing for positive definite matrices is used in form of the Cholesky decomposition, the one-sided Jacobi methods are very accurate eigensolvers [5]. Recently, it has been reported in [8, 9] that the clever implementation of a serial one-sided Jacobi algorithm reached the efficiency of the QR method. However, the QR method uses a bidiagonalization as a preprocessing step, which means that the relative accuracy is lost and cannot be recovered using

*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.

the subsequent one-sided Jacobi method. Consequently, the QR method should not be used in such applications where the high relative accuracy of computed singular values is required (e.g., the computation of energies of atoms and molecules in quantum physics and quantum chemistry), and the one-sided Jacobi method is the choice.

This report considers a generalization of the scalar one-sided Jacobi algorithm to the block case. Working with blocks on a serial computer leads to the better usage of available memory, especially of cache, thus boosting the data flow and computation. Moreover, the computation with blocks allows for the higher degree of parallelism. Both strategies have been already successfully applied in the case of the two-sided parallel block-Jacobi method; see [2, 3, 4].

Next section contains a detailed description of the one-sided block-Jacobi algorithm (OSBJA) for computing the SVD of a general rectangular matrix A. The emphasis is given to the block-column partition of A as well as to the block formulation of all orthogonal updates needed during computation.

Two crucial main ideas with respect to the accelerating of the serial OSBJA is given in section 3. The first idea was introduced by Drmač and Veselić in [8, 9] for the serial one-sided Jacobi method, and by Okša and Vajteršic for the parallel two-sided Jacobi method. It consists of preprocessing a given matrix before its SVD by computing its QR decomposition with column pivoting (either serial or in parallel), so that the SVD algorithm is then applied to the triangular factor. The second idea, which uses the cosine-sine (CS) decomposition of certain block-orthogonal matrix and leads to so-called fast scaled block-orthogonal transformations, was introduced by Hari in [12]. These two ideas cooperate and the result is the more efficient usage of a fast cache memory and a reduced number of sweeps required for the overall algorithm's convergence at given precision.

Section 4 discusses main guidelines for a parallel implementation of the OSBJA. We are especially interested in distributed systems with the Message Passing Interface (MPI) Standard for sending/receiving messages. In such parallel environment we describe a possible parallel implementation of individual phases of the OSBJA indicating the computational and communication complexities. In section 5 we give an outline of future work.

2 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:

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

This requires $(n_i+n_j)(n_i+n_j-1)/2$ dot products or $m\,(n_i+n_j)(n_i+n_j-1)/2$ flops. As will be soon clear, except for a part of the first sweep, the two diagonal blocks of $\hat{A}^{(k)}$ will be always diagonal. This reduces the flop count to $m\,(n_i\,n_j+n_i+n_j)$ where $m\,(n_i+n_j)$ comes from the computation of the diagonal elements of $\hat{A}^{(k)}$.

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 $(A_i^{(k)}, A_j^{(k)})$. This diagonalization requires (as will be discussed later) on average around $8(n_i+n_j)^3$ flops.

3. Finally, an updating of two block-columns of $A^{(k)}$ and $V^{(k)}$ is required, which requires $2m(n_i+n_j)^2$ flops.

In summary, the kth step of the standard OSBJA requires

$$\begin{aligned} N_{\text{flop}}(k) &\approx m(n_i n_j + n_i + n_j) + 8(n_i+n_j)^3 + 2m(n_i+n_j)^2 \\ &= 64n_0^3 + (9n_0^2+2n_0)n \quad (\text{if } m=n=n_0 r) \end{aligned} \tag{6}$$

flops.

Let us discuss shortly these three parts in terms of the CPU time needed for their computation on a serial computer. The first part needs the computation of dot products of length m, which are fast in comparison with matrix multiplications required in the third part. The second part, the eigendecomposition of $\hat{A}^{(k)}$, will be fast provided that we can choose the block width small enough to perform all needed computations in the cache memory (notice that $\hat{A}^{(k)}$ is of the order only n_i+n_j). So, the third part of each step in the OSBJA seems to be the most demanding one.

Notice that the cross-product matrix $\hat{A}^{(k)}$ in the second phase is symmetric and positive definite. Moreover, for larger indices k, the matrix $\hat{A}^{(k)}$ will be almost diagonal and we can use, for example, the two-sided Jacobi method to perform (5). This will ensure that $\hat{U}^{(k)}$ and $\hat{\Lambda}_{ij}^{(k)}$ will be computed with high accuracy; see [5].

When speaking about convergence of the OSBJA, each one-sided block-Jacobi method has its counterpart, the two-sided block-Jacobi method applied on the cross-product matrix A^TA. If the latter converges to the diagonal matrix Σ^2 (notice that A^TA is symmetric, positive definite, so it has positive eigenvalues), then $A^{(k)}$ approaches the set of orthonormal matrices whose columns are (up to a permutation) the left singular vectors of A. The same accumulation points has the matrix whose columns are the normalized columns of $A^{(k)}$. Hence, the Euclidean norms of columns of $A^{(k)}$ converge (in some order) to the singular values of A. At the same time, $V^{(k)}$ is an orthogonal matrix (by construction) whose columns approach the right singular vectors of A. Individual singular triplets of A must be recovered from the order in which the singular values converge, i.e., one has to use an appropriate permutation. In practice, the serial (and all cyclic) block-Jacobi methods always converge.

3 Accelerating the One-Sided Block-Jacobi Algorithm

To improve the speed and efficiency of the OSBJA, one can try first to reduce the number of sweeps required for the convergence of the entire algorithm at given precision. This is the highest level of improvement that requires in general some sort of *preprocessing* of an original matrix A. In subsection 3.1 we describe two steps of the preprocessing, which help to reduce, sometimes quite substantially, the number of sweeps. Next, we present new ideas of Hari [12], which reduce the flop count and the CPU time within one step of the OSBJA using a recursive implementation of so-called *fast scaled block-orthogonal transformations*.

3.1 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 3.1.2), 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 $A^{(k)}$ 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). Next two subsections contain details with respect to both preprocessing steps.

3.1.1 QR and LQ factorizations

For a rectangular matrix A of order $m \times n$ with $m \gg n$, it is a common practice first to apply the QR factorization prior to the SVD computation, since then we work only with the R-factor of much lower size $n \times n$. However, in the context of the OSBJA, it was suggested in [8] to perform *two* factorizations of an initial matrix A,

$$A = Q_1 R P^T \quad \text{and} \quad R = L Q_2^T,$$

so that the SVD computation is then applied to the final L-factor. Here Q_1 and Q_2 are orthogonal, P is a permutation matrix, R is upper and L lower triangular. The first factorization is the QR decomposition with column pivoting, the second one is the LQ decomposition of the R-factor. It is very important that both decompositions *preserve* the relative accuracy of the singular values and vectors that will be computed next by the Jacobi process (this is at least true for the case when AD is well-conditioned for some diagonal matrix D; see [5, 7]). The number of needed flops is of order mn^2.

Next, the OSBJA starts with the L-factor partitioned into r block-columns, $L = [L_1, L_2, \ldots, L_r]$. Moreover, the L-factor is almost diagonal, so that the number of Jacobi steps till the convergence at given precision is highly reduced. In [8, 9] some special pivot strategies were suggested that operate more frequently in the vicinity of diagonal. This further reduces the number of steps till the convergence. The experience with the serial implementation shows that in general only approximately $n^2/2$ Jacobi rotations are needed (although there are matrices, which require several times more rotations till convergence).

The almost diagonal form of the L-factor also implies that the rotation angles will be small. This, together with the reduced number of rotations, increases the accuracy of computed singular values and singular vectors. As recognized by Hari in [12], the Jacobi method applied to the L-factor can be much faster and accurate than when applied to the original matrix A without any preprocessing.

The additional important impact of this preprocessing is that the right singular vectors can be accurately computed *a posteriori* from the linear system

$$LV = U\Sigma,$$

where $U\Sigma$ is the accumulation point (i.e., final stage) of Jacobi iterates. Drmač [7] has shown that L is well-conditioned for the accurate computation of the columns of V. The cost is around $n^3/2$ flops. This means that in (1) we can *discard the orthogonal updating of* $V^{(k)}$, which immediately saves half of flops in the slowest part of the algorithm.

Once we have U, Σ and V, the SVD of A is obtained from:

$$A = Q_1 L Q_2^T P^T = (Q_1 U)\Sigma(PQ_2 V)^T. \tag{7}$$

Now we can summarize the flops that follow from the preprocessing step. We have mn^2 flops for two factorizations, $n^3/2$ flops for the final computing of V and around $mn^2 + n^3$ flops from matrix multiplications required in (7) for computing the final left and right singular vectors of A. The last two items define the work done in *postprocessing* rather than in preprocessing. Altogether, this yields $2mn^2 + 1.5n^3$ flops in addition to the work done in Jacobi iterates. On the other side, we hope to substantially reduce the number of Jacobi steps needed for the convergence, and we also discarded the orthogonal updates of right singular vectors during iterations.

As noted in [12], the methods based on the bidiagonalization require $2mn^2-(4/3)n^3$ flops for the bidiagonalization and $mn^2 + n^3$ flops for updating singular vectors. Thus the additional work is $3mn^2 - n^3/3$ flops, usually smaller (but not substantially smaller, depending on the relation between m and n) than for the Jacobi method, however, *without being relatively accurate.*

3.1.2 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. $\quad \hat{A}_{ii}^{(0)} = L_i^T L_i$;
3. $\quad \hat{A}_{ii}^{(0)} = Q_{ii}^{(0)} \Gamma_i^{(0)} Q_{ii}^{(0)T}$; (spectral decomposition)
4. $\quad (A_i^{(0)} = L_i Q_{ii}^{(0)})$; (not performed, just an illustration of the connection)
5. **end**;

Thus the above algorithm initializes three important matrices:

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

At step k of the Jacobi process, the matrices will be $B^{(k)}$, $Q^{(k)}$ and $\Gamma^{(k)}$ (in fact the vector $\gamma^{(k)}$), and we will show in next subsection how to update them efficiently.

If $n = n_0 r$, then the computation of the step 1 in the above initialization requires around $(n_0+1)n^2/4$ flops. Assuming that the spectral decomposition of symmetric positive semidefinite matrices in step 2 is done using some sort of the one- or two-sided Jacobi algorithm with, say, 4 sweeps, we need here around $8n_0^2 n$ flops. Altogether, the initialization requires around $(n/4 + 8n_0)n_0 n$ flops.

This idea of initialization and recursive updating of three various matrices (instead of updating $A^{(k)}$, see (1)) is due to Hari [12]. We will see in the next subsection that these updates can be done very efficiently. Moreover, there is a connection at each step $k \geq 0$ of the Jacobi algorithm:

$$A^{(k)} = B^{(k)} Q^{(k)}, \quad \hat{A}^{(k)} = A^{(k)T} A^{(k)}, \quad \Gamma^{(k)} = \mathrm{diag}(\hat{A}^{(k)}). \tag{8}$$

3.2 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 (8). Then, according to (4),

we need to compute the cross-product matrix $\hat{A}_{ij}^{(k)}$ for the given pivot pair (i,j):

$$\begin{aligned}\hat{A}_{ij}^{(k)} &= \begin{pmatrix} Q_{ii}^{(k)} & \\ & Q_{jj}^{(k)} \end{pmatrix}^T \begin{pmatrix} B_i^{(k)T}B_i^{(k)} & B_i^{(k)T}B_j^{(k)} \\ B_j^{(k)T}B_i^{(k)} & B_j^{(k)T}B_j^{(k)} \end{pmatrix} \begin{pmatrix} Q_{ii}^{(k)} & \\ & Q_{jj}^{(k)} \end{pmatrix} \\ &= \begin{pmatrix} \Gamma_i^{(k)} & \tilde{A}_{ij}^{(k)} \\ \tilde{A}_{ij}^{(k)T} & \Gamma_j^{(k)} \end{pmatrix}, \text{ where } \tilde{A}_{ij}^{(k)} \equiv Q_{ii}^{(k)T}(B_i^{(k)T}B_j^{(k)})Q_{jj}^{(k)}. \end{aligned} \tag{9}$$

Thus, using the symmetry of $\hat{A}_{ij}^{(k)}$, we have to compute only n_in_j dot products involving columns of $(B_i^{(k)}, B_j^{(k)})$ and two additional matrix multiplications—by $Q_{ii}^{(k)T}$ from left and by $Q_{jj}^{(k)}$ from right. Altogether, the formation of $\hat{A}_{ij}^{(k)}$ requires $n_in_j(n+n_i+n_j)$ flops.

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

$$\begin{aligned}\hat{U}^{(k)} &= \begin{pmatrix} V_{ii}^{(k)} & \\ & V_{jj}^{(k)} \end{pmatrix} \begin{pmatrix} C_{ii}^{(k)} & -S_{ij}^{(k)} \\ S_{ji}^{(k)} & C_{jj}^{(k)} \end{pmatrix} \begin{pmatrix} W_{ii}^{(k)} & \\ & W_{jj}^{(k)} \end{pmatrix}^T \\ &\equiv \hat{V}^{(k)}\,\hat{T}^{(k)}\,\hat{W}^{(k)T}, \end{aligned} \tag{10}$$

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

$$\hat{T}^{(k)} = \begin{pmatrix} C_{ii}^{(k)} & -S_{ij}^{(k)} \\ S_{ji}^{(k)} & C_{jj}^{(k)} \end{pmatrix} = \begin{cases} \begin{pmatrix} I_{n_i-n_j} & 0 & 0 \\ 0 & C^{(k)} & -S^{(k)} \\ 0 & S^{(k)} & C^{(k)} \end{pmatrix}, & \text{if } n_i \geq n_j, \\ \begin{pmatrix} C^{(k)} & 0 & -S^{(k)} \\ 0 & I_{n_j-n_i} & 0 \\ S^{(k)} & 0 & C^{(k)} \end{pmatrix}, & \text{if } n_j \geq n_i, \end{cases} \tag{11}$$

and

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

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 $(A_i^{(k+1)}, A_j^{(k+1)})$. Using (8) and the CS decomposition of $\hat{U}^k$, the next iteration can be written in the factored form:

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

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

$$\begin{aligned}(B_i^{(k+1)}, B_j^{(k+1)}) &= (B_i^{(k)}(Q_{ii}^{(k)}V_{ii}^{(k)}), B_j^{(k)}(Q_{jj}^{(k)}V_{jj}^{(k)})\,\hat{T}^{(k)}, \\ Q_{ii}^{(k+1)} &= W_{ii}^{(k)T}, \quad Q_{jj}^{(k+1)} = W_{jj}^{(k)T}. \end{aligned} \tag{12}$$

(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 (12) 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.

3.3 Modified algorithm

For clarity, we summarize in this section all three parts of step k of the modified algorithm, where 'modified' refers to using the block matrix recursion (12).

- The first part computes the matrix product $Z = B_i^{(k)T} B_j^{(k)}$ where (i, j) is the actual pair of pivot indices defined by the block ordering. This requires $n_i n_j n \approx n_0^2 n$ flops and can be implemented either using $n_i n_j$ dot products or one matrix multiplication. As discussed in [12], the last option is several times faster when using the BLAS 3 procedure *GEMM from the LAPACK.

- In the second part, *all computations have to be done in the fast (cache) memory.* This part can be divided into four phases:

  1. Computation of $Q_{ii}^{(k)T} Z Q_{jj}^{(k)}$, which requires $n_i^2 n_j + n_i n_j^2 \approx 2n_0^3$ flops and can be implemented by two calls of the BLAS 3 procedure *GEMM. To assemble the matrix $\hat{A}_{ij}^{(k)}$, we take the appropriate $n_i + n_j$ elements from the vector $\gamma^{(k)}$ and copy them onto diagonal of $\hat{A}_{ij}^{(k)}$ (these are the diagonal blocks $\Gamma_i^{(k)}$ and $\Gamma_j^{(k)}$ of $\hat{A}_{ij}^{(k)}$). We also copy zeros into diagonal blocks of $\hat{A}_{ij}^{(k)}$ outside its diagonal.

  2. Diagonalization (eigendecomposition) of $\hat{A}_{ij}^{(k)}$. Here we can choose from several options that preserve the relative accuracy of singular values and vectors:
     - Application of a two-sided Jacobi method with a suitable pivot strategy to $\hat{A}_{ij}^{(k)}$ (which is symmetric and positive definite). 'Optimal' strategies skip zeros and small elements(i.e., they are not nullified) and some of them were discussed in [8, 9]. Such a strategy significantly reduces the number of rotations required for the convergence. Recall that $\hat{A}_{ij}^{(k)}$ is of order $n_i + n_j$. Hari [12] mentions that for matrices of order 64, having uniformly or normally distributed elements, only one sweep (i.e., $(n_i + n_j)(n_i + n_j - 1)/2$ rotations) were needed, and for blocks generated by the OSBJA maximum three sweeps were needed for convergence. This is caused by the special structure of $\hat{A}_{ij}^{(k)}$ where there is a lot of zero elements

in diagonal blocks at the very beginning of its spectral decomposition. Moreover, the smaller number of rotations has, in general, good impact on the accuracy of output data.

- Since the diagonal blocks of $\hat{A}_{ij}^{(k)}$ are diagonal, and since for larger k the matrix $\hat{A}_{ij}^{(k)}$ is almost diagonal, the Mascarenhas strategy described in [18] can be used.
- We can use the Cholesky factorization of $\hat{A}_{ij}^{(k)}$ followed by the simple one-sided Jacobi method applied to the factor $L_{ij}^{(k)}$. Details are discussed in [12].
- Use the Cholesky factorization of $\hat{A}_{ij}^{(k)}$ followed by the Kogbetliantz serial method, which preserves the triangular structure of L-factor. As shown by Hari and Matejaš [10], the Kogbetliantz method is relatively accurate for the SVD computation when applied to the triangular factor coming from the QR or Cholesky decomposition.

In general, the experience from [12] shows that the diagonalization $\hat{A}_{ij}^{(k)}$ (together with computing the eigenvector matrix $\hat{U}^{(k)}$) requires around $6(n_i + n_j)^3 \approx 48n_0^3$ flops or less.

3. Computing the CS decomposition of the orthogonal eigenvector matrix $\hat{U}^{(k)}$ according to (10), and then the matrix products $Q_{ii}^{(k)}V_{ii}^{(k)}$ and $Q_{jj}^{(k)}V_{jj}^{(k)}$. The CS decomposition requires essentially two SVDs, one for the diagonal block $\hat{U}_{ii}^{(k)}$, another for $\hat{U}_{jj}^{(k)}$:

$$\hat{U}_{ii}^{(k)} = V_{ii}^{(k)}C_{ii}^{(k)}W_{ii}^{(k)T}, \quad \hat{U}_{jj}^{(k)} = V_{jj}^{(k)}C_{jj}^{(k)}W_{jj}^{(k)T};$$

then (10) follows with off-diagonal blocks of the middle matrix in form $V_{ii}^{(k)T}\hat{U}_{ij}^{(k)}W_{jj}^{(k)}$ and $V_{jj}^{(k)T}\hat{U}_{ji}^{(k)}W_{ii}^{(k)}$. Usually these blocks are diagonal to working accuracy and define the diagonal matrices $-S_{ij}^{(k)}$ and $S_{ji}^{(k)}$. Only if there are very close diagonal elements in $C_{ii}^{(k)}$ and $C_{jj}^{(k)}$, a special 'cleansing procedure' is needed to reliably compute $-S_{ij}^{(k)}$ and $S_{ji}^{(k)}$. This cleansing procedure (see [12]) is also needed in the later stage of the Jacobi process when $\hat{U}_{ii}^{(k)}$ becomes close to identity with $\|U_{ij}^{(k)}\|_2 \approx \|U_{ji}^{(k)}\|_2 \approx \sqrt{\epsilon}$, where ϵ is the round-off unit. Without cleansing the Jacobi method becomes stagnant. The CS decomposition with a special cleansing procedure is discussed in [12].

For the SVD of $\hat{U}_{ii}^{(k)}$, one can use, for example, the QR factorization $\hat{U}_{ii}^{(k)} = \tilde{Q}_{ii}^{(k)}\tilde{R}_{ii}^{(k)}$ followed by the Kogbetliantz method applied to $\tilde{R}_{ii}^{(k)}$ (or to $\tilde{R}_{ii}^{(k)T}$). Now we show that both processes, namely the QR factorization of $\hat{U}_{ii}^{(k)}$ and the SVD of $\tilde{R}_{ii}^{(k)}$, can be used for the computation of the product $Q_{ii}^{(k)}V_{ii}^{(k)}$ in such a way that $V_{ii}^{(k)}$ is not explicitly needed. First, having $\tilde{Q}_{ii}^{(k)}$ from the QR decomposition in the factored from (fast scaled rotations or Householder reflectors), we can apply them directly to $Q_{ii}^{(k)}$ and compute $\bar{Q}_{ii}^{(k)} = Q_{ii}^{(k)}\tilde{Q}_{ii}^{(k)}$. Second, we can apply the (fast scaled) rotations produced by the Kogbetliantz method directly to $\bar{Q}_{ii}^{(k)}$, so that $V_{ii}^{(k)}$ is not explicitly needed. Thus, at the end we have computed the required matrix product $Q_{ii}^{(k)}V_{ii}^{(k)}$. Obviously, the same strategy can be used to compute $Q_{jj}^{(k)}V_{jj}^{(k)}$.

Hari [12] summarized the flops assuming 4 sweeps of the Kogbetliantz method as follows: $(2/3)n_i^3$ flops for $\tilde{R}_{ii}^{(k)}$, n_i^3 flops for $\bar{Q}_{ii}^{(k)}$, $4n_i^3$ flops for $C_{ii}^{(k)}$, $4n_i^3$ flops for $V_{ii}^{(k)}$

and $4n_i^3$ flops for $W_{ii}^{(k)}$—in total, around $14n_i^3$ flops for the first SVD. Similarly, we have around $14n_j^3$ flops for the second SVD. Then we need to compute $V_{ii}^{(k)T}\hat{U}_{ij}^{(k)}W_{jj}^{(k)}$ and $V_{jj}^{(k)T}\hat{U}_{ji}^{(k)}W_{ii}^{(k)}$, which adds $2n_in_j(n_i+n_j)$ flops. Hence, this phase needs in total around $14(n_i^3+n_j^3)+2n_in_j(n_i+n_j)\approx 32n_0^3$ flops.

Now we can estimate the total number of flops needed in the second part of the modified algorithm:

$$n_in_j(n_i+n_j)+6(n_i+n_j)^3+14(n_i^3+n_j^3)+2n_in_j(n_i+n_j)\approx 82n_0^3 \text{ flops.}$$

- Finally, in the third part we have to compute first:

$$\tilde{B}_i^{(k)}=B_i^{(k)}(Q_{ii}^{(k)}V_{ii}^{(k)}),\quad \tilde{B}_j^{(k)}=B_j^{(k)}(Q_{jj}^{(k)}V_{jj}^{(k)})$$

and afterwards

$$(B_i^{(k+1)},B_j^{(k+1)})=(\tilde{B}_i^{(k)},\tilde{B}_j^{(k)})\hat{T}^{(k)}.$$

Since the matrix products $(Q_{ii}^{(k)}V_{ii}^{(k)})$ and $(Q_{jj}^{(k)}V_{jj}^{(k)})$ were already computed in the second part, the computation of $(\tilde{B}_i^{(k)},\tilde{B}_j^{(k)})$ requires only $n(n_i^2+n_j^2)$ flops. The final matrix multiplication by $\hat{T}^{(k)}$ uses the special structure of this matrix (see (11) and can be implemented using so-called fast scaled rotations using $2n\min\{n_i,n_j\}$ flops; details can be found in [12]. Altogether, the third part of the modified algorithm requires around $n(n_i^2+n_j^2)+2n\min\{n_i,n_j\}\approx 2n(n_0^2+n_0)$ flops. This is to be compared with the standard algorithm, for which the third part (updating of two block columns of $A^{(k)}$ and $V^{(k)}$) requires $\approx 8mn_0^2$; for $m=n$ the modified updating is about 4 times faster. If the standard algorithm would compute the right singular vectors *a posteriori*, the saving would be 50 per cent.

In summary, the preprocessing of an original matrix A by QR factorization with column pivoting together with the recursion applied to three matrices that can be updated in the fast (cache) memory enables to substantially reduce the number of flops per one iteration step [12]:

$$N_{\text{flop}}(k)\approx 82n_0^3+(3n_0^2+2n_0)n. \tag{13}$$

When $m=n$, the comparison with (6) shows that we have reduced three times the coefficient at n_0^2n 'paying the price' by increasing the coefficient at n_0^3 from 64 to 82, i.e. by 28 per cent. When $m\gg n$ one step of the modified algorithm can be several times faster than that of the standard one (despite the fact that we have increased the amount of work in cache) because the 'large' dimension m is not used at all. In addition, due to the concentration of the Frobenius norm of A towards diagonal by the preprocessing the number of sweeps needed for the convergence at given precision is also reduced.

3.4 Stopping criterion

The important question is when to stop the Jacobi iterations. If stopped too early, we could not have the convergence up to given accuracy. When stopped too late, unnecessary sweeps would be completed spending perhaps much more time than needed.

For the one-sided block Jacobi method, to define a reliable stopping criterion is more difficult than for the two-sided block Jacobi method. To understand the problem, let us introduce more notation to that defined in (9). Let

$$\hat{A} = A^T A, \quad D = (\mathrm{diag}(\hat{A}))^{1/2}, \quad \hat{A}_S = D^{-1}\hat{A}D^{-1},$$

and for $k \geq 1$,

$$\begin{aligned}
\hat{A}^{(k)} = A^{(k)T}A^{(k)} &= \begin{pmatrix} \tilde{A}_{11}^{(k)} & \cdots & \tilde{A}_{1r}^{(k)} \\ \vdots & & \vdots \\ \tilde{A}_{1r}^{(k)T} & \cdots & \tilde{A}_{rr}^{(k)} \end{pmatrix}, \\
D_k = (\mathrm{diag}(\hat{A}^{(k)}))^{1/2} &= \mathrm{diag}(\|A^{(k)}e_1\|, \ldots, \|A^{(k)}e_n\|) = (\Gamma^{(k)})^{1/2}, \\
\hat{A}_S^{(k)} &= D_k^{-1}\hat{A}^{(k)}D_k^{-1} \text{ (scaled matrix).}
\end{aligned}$$

For any symmetric matrix X,

$$\mathrm{off}(X) = \frac{\sqrt{2}}{2}\,\|X - \mathrm{diag}(X)\|_{\mathrm{F}}$$

is called *the departure from diagonal form* and equals to the Frobenius norm of its strictly upper (or lower) triangle. Hari [12] introduced two measures for advancing the Jacobi process:

$$\begin{aligned}
\alpha_k &\equiv \mathrm{off}(\hat{A}_S^{(k)}) = \frac{\sqrt{2}}{2}\,\|\hat{A}_S^{(k)} - I\|_{\mathrm{F}}, \\
\omega_k &\equiv \mathrm{off}(\hat{A}^{(k)}) = \sqrt{\sum_{r=1}^{p}\sum_{t=r+1}^{p} \|\tilde{A}_{rt}^{(k)}\|_{\mathrm{F}}^2}.
\end{aligned}$$

When the Jacobi process converges, all columns of $A^{(k)}$ become more and more orthogonal to each other so that $\hat{A}_S^{(k)} \to I_n$ as k increases. Notice that α_k is the square root of the sum of squares of cosines of all angles between pairs of columns of $A^{(k)}$ (the scaling by D_k^{-1} is substantial here!). Hence, α_k is the appropriate measure of convergence; if $\alpha_K \approx 0$ for some K we should stop the iterations. Since the computation of α_k involves $n(n-1)/2$ dot products and the normalized columns of $A^{(k)}$ can be computed with an absolute error as large as $n\epsilon$, the convergence criterion may have the form

$$\alpha_k \leq n^2\epsilon. \tag{14}$$

The problem with (14) is the updating of α_k; we need to scale the columns of $A^{(k)}$ by their norms and then to compute $n(n-1)/2$ dot products. This requires about $n^3/2$ flops and is too expensive to be computed at the end of each sweep. On the other hand, the updating of ω_k is cheap. Because $\mathrm{off}(\tilde{A}_{rr}^{(k)}) = 0 = \mathrm{off}([\hat{A}_S^{(k)}]_{rr})$ for all $k \geq 0$ and for all r, $1 \leq r \leq p$, one has at each step of the Jacobi iteration under any pivot strategy

$$\omega_{k+1}^2 = \omega_k^2 - \|\tilde{A}_{i(k),j(k)}^{(k)}\|_{\mathrm{F}}^2, \quad k \geq 0. \tag{15}$$

This recursive update can be extended to sweeps. Each full block sweep consists of $N = r(r-1)/2$ block steps, so that at the end of sweep $t+1$ one has

$$\omega_{(t+1)N}^2 = \omega_{tN}^2 - \sum_{k=tN}^{(t+1)N-1} \|\tilde{A}_{i(k),j(k)}^{(k)}\|_{\mathrm{F}}^2, \quad t \geq 0. \tag{16}$$

At the beginning (after both steps of preprocessing), one can compute ω_0 and, in addition, the initial value $\|\hat{A}^{(0)}\| = \sqrt{2\omega_0^2 + \sum_l (\gamma_l^{(0)})^2}$. At step k, the matrix block $\tilde{A}^{(k)}_{i(k),j(k)}$ is at disposal and is nullified so that the sum on the left side of (16) can be easily computed at cost of $n(n-1)/2 - r\,n_0(n_0-1)/2 = (n-n_0)n/2$ additional flops.

The numerical behavior of the recursion (16) was analyzed in [12]. Assuming the quadratic convergence of the modified method (this is to be proven yet), the quadratic reduction of ω_{tN} will begin after, say, $t = t_0$, which would lead to

$$\omega_{(t+1)N} \leq c\,\omega_{tN}^2, \quad \forall t \geq t_0, \tag{17}$$

where the constant $1/c$ is proportional to the minimum distance in the spectrum of $\hat{A}^{(0)}$ [22]. Combination of (16) and (17) gives

$$\nu_{tN} \equiv \sqrt{\sum_{k=tN}^{(t+1)N-1} \|\tilde{A}^{(k)}_{i(k),j(k)}\|_{\mathrm{F}}^2} = \omega_{tN} + O(\omega_{tN}^2), \quad t \geq t_0, \tag{18}$$

i.e., ν_{tN} can serve as the estimate of ω_{tN} in the regime of quadratic convergence. Note that ν_{tN} is computed at the end of sweep $t+1$.

When the quadratic convergence begins and ω_k^2 is computed according to (15), a severe cancellation can take place and ω_k^2 can become quickly negative. Hari [12] shows that the error in the recursive computation of ω_{tN} can be as large as $(n^2/2)(\gamma_1^{(tN)})^2\epsilon$, where $\gamma_1^{(tN)}$ is the maximum diagonal element of $\hat{A}^{(k)}$ approximating the spectral norm $\|\hat{A}^{(k)}\|_2$. On the other hand, the computation of ν_{tN} according to (18) has a small relative error. As soon as ω_{tN}^2, computed recursively according to (15), becomes as small as $(n^2/2)(\gamma_1^{(tN)})^2\epsilon$, it has probably lost all its significant digits and is not reliable anymore. Therefore, it is worth to monitor both parameters, ω_{tN}^2 and ν_{tN}, because ν_{tN} is a reliable estimate of the true value of ω_{tN}^2 when the latter parameter becomes very small (or even negative). At the end of sweep $t+1$ one has according to (17) $\omega_{(t+1)N} \lessapprox c\,\nu_{tN}^2$. Hence, if

$$\nu_{tN} \leq n\sqrt{\gamma_1^{((t+1)N)}\epsilon} \tag{19}$$

we have

$$\omega_{(t+1)N} \lessapprox c\,n^2\,\gamma_1^{((t+1)N)}\epsilon \approx c\,n^2\,\|\hat{A}^{((t+1)N)}\|_2\,\epsilon \leq c\,n^2\,\|\hat{A}^{((t+1)N)}\|_{\mathrm{F}}\,\epsilon,$$

so that $\omega_{(t+1)N}/\|\hat{A}^{((t+1)N)}\|_{\mathrm{F}}$ is as tiny as $c\,n^2\,\epsilon$. Therefore, (19) gives the stage at which one can compute $\alpha_{(t+1)N}$ using $n^3/2$ flops and check the convergence of the Jacobi iterations according to (14).

Now there are two possibilities. Either (14) holds and we stop the process, or the stopping criterion is not fulfilled and we need to estimate the number of sweeps till the convergence. For this purpose we would need the quadratic convergence result for $\alpha_{(t+1)N}$ similarly to the simple (not block) one-sided Jacobi method (see [19]), which is not proved at the moment. Hence, we can prescribe a small number of sweeps (say, 2) and then check the parameter α_k again.

4 Parallelization Strategy

Next we describe main ideas behind the parallelization of one-sided block-Jacobi SVD method. We start with discussion of data layout and parallel block ordering. Afterwards, we shortly discuss a possible parallel implementation of all phases of the algorithm.

4.1 Data layout, preprocessing and block parallel ordering

In this section 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.

Since the cross-product matrix $\hat{A}_{ij}^{(k)}$ in (4) is to be computed at the beginning of the OSBJA using two block columns, to keep the interprocessor communication at minimum in this stage of computation it is natural to assume that this cross-product is made in one processor (so that no distributed matrix-matrix multiplication is necessary, which would slow-down the algorithm considerably). Hence, having p processors, the natural data layout consists of two block columns per processor, i.e. the blocking factor (the number of block columns) is $r = 2p$. At the beginning, we can assign two consecutive block columns to consecutive processors. Notice that the similar data layout was used in [2, 3, 4] for the *two-sided* block-Jacobi method.

At the beginning, we have a distributed original matrix A with two block columns per processor. 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's routine `PDGEQPF` and `PDGELQF`, respectively.

The second preprocessing consists of the initialization that consists of 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 3.1.2. 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—see remarks in section 3.3.

Having diagonalized the diagonal blocks of the cross-product matrix $\hat{A}^{(0)}$, next we need to choose r pivot pairs (i, j) that define r subtasks, which 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. This data movement must be repeated at the beginning of each parallel step. 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 minimalized. 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 somehow adapt our *dynamic* ordering that was designed for the parallel two-sided block-Jacobi algorithm in [4].

4.2 Individual phases of the algorithm

When two block columns are assigned to each processor, all computations in the modified algorithm (see section 3.3) 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. It is therefore useful to minimize the amount of transferred data by choosing a suitable block parallel ordering, which , for example, would transfer only one half of data (i.e., essentially only one block column).

In contrast to local computations in the modified algorithm, the implementation of the stopping criterion requires some sort of global communication between processors. The update of ω^2 in (16) and of ν in (18) 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`).

5 Conclusions

We have summarized and analyzed in some detail recent new ideas for accelerating the serial one-sided block-Jacobi method. The most important features of the modified algorithm are: the preprocessing of an original matrix by the QRF with column pivoting (with the optional LQF of R-factor), working with matrix blocks rather than with matrix elements, the special initialization, the block-matrix recursion and the CS decomposition. The analysis of a possible parallelization strategy is also provided. In the near future, we plan to implement the parallel one-sided block-Jacobi algorithm with all accelerating features on a parallel computer with distributed memory.

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] 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.

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