UNIVERSITÄT SALZBURG

Generalization of the Dynamic Ordering for the One-Sided Block Jacobi SVD Algorithm: I. Analysis and Design

Martin Bečka[a] Marián Vajteršic

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

Technical Report 2008-01 June 2008

Department of Computer Sciences

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

Technical Report Series

Generalization of the Dynamic Ordering for the One-Sided Block Jacobi SVD Algorithm: I. Analysis and Design

Martin Bečka* and Marián Vajteršic[†]

Abstract. *The efficiency of the one-sided parallel block-Jacobi algorithm for computation of the singular value decomposition (SVD) of a general matrix $A \in \mathbb{R}^{m \times n}$, $m \geq n$, depends–besides some numerical tricks that speed-up the convergence–crucially on the parallel ordering of subproblems, which are to be solved in each parallel iteration step. We discuss in detail possible generalizations of the so-called dynamic ordering of subproblems that was originally designed for the two-sided parallel block-Jacobi SVD algorithm. It turns out that the straightforward generalization leads to the algorithm that requires too much communication in each parallel iteration step, so that the Jacobi method would spend too much time in communication and its efficiency due to the clever usage of some numerical tricks will be lost. Therefore, a modification of the dynamic ordering is proposed based on the use of the maximum principal angle between two subspaces of the same dimension. This approach can be described as a heuristics. Its efficiency should be tested with using a wide set of random matrices and compared with the (row or column) cyclic ordering.*

1 Introduction

This report is the third 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. In the second part [22], the parallel implementation and overall data layout was discussed in some detail with the emphasis on changing the data distribution needed for the preprocessing (the QR decomposition with column pivoting) and subsequent SVD computation.

For the two-sided parallel block-Jacobi SVD algorithm we have designed, implemented and successfully tested in [4] the so-called *dynamic* ordering that defines, in the case of p processors,

*Mathematical Institute, Department of Informatics, Slovak Academy of Sciences, Bratislava, Slovak Republic, email: Martin.Becka@savba.sk.

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

p pairs of off-diagonal blocks, which are nullified and, at the same time, the decrease of the off-diagonal Frobenius norm is maximal. This strategy leads to a substantial decrease of the number of parallel iteration steps needed for the convergence of the whole algorithm at given precision. For the OSBJA, no such strategy is known. It is the purpose of this report to discuss the possible application of dynamic ordering (with some modifications) to the one-sided case.

This report should be read in close relationship with [21] and [22]. 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 the ordering strategy, which should be related to the dynamic ordering in the two-sided case.

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 for 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, $1 \leq i \leq r$, 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 $\mathcal{F} : \{0, 1, \ldots\} \rightarrow \mathbf{P}_r = \{(l, m) : 1 \leq l < m \leq r\}$. 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 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 the updating of $A^{(k)}$. 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. $\quad \hat{A}_{ii}^{(0)} = Q_{ii}^{(0)} \Gamma_i^{(0)} Q_{ii}^{(0)T}$; (eigenvalue 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:

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

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 $\Gamma^{(k)}$. 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),

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

and

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

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:

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

(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 a simple copy 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 Ordering of Subproblems

3.1 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 (B_i, B_j) 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 and the 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 of 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. 1. In the two-sided block Jacobi

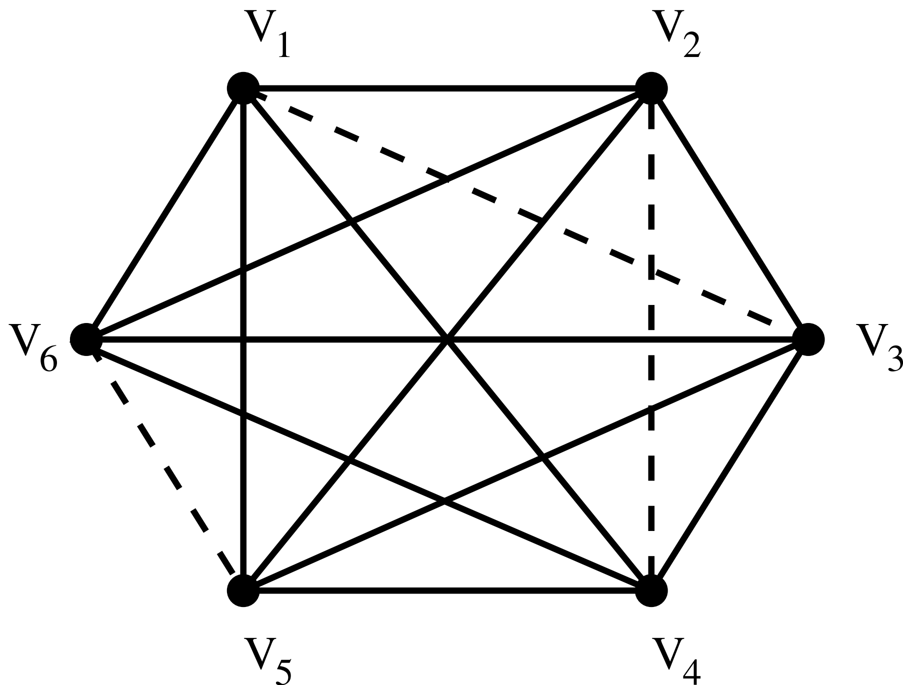

Figure 1: Maximum-weight perfect matching on a complete graph for $r = 6$. The chosen edges are dashed.

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. 1). 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 Frobenius 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 B_i 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 (B_i, B_j), we can define the departure from their mutual orthogonality by

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

where $b_t^{(k)}$ is the t-th column of the matrix block B_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 (B_i, B_j) requires $O((n/r)^2)$ scalar products, each of length n. 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 B_i must meet each matrix block column B_j, $j \neq i$, in some processor, in which the updated weight w_{ij} is computed according to (10). Since there are exactly two block columns in each processor (each with $n/2p$ columns for the blocking factor $r = 2p$), the update of weights can be achieved by organizing the processors into a ring and using the

neighbor communication pattern, where each processor sends its the 'left' block column to its left neighbor and receives one block column from its right neighbor. In the second round, the local 'right' block column is sent to the right neighbor and one block column is received from the left neighbor. Hence, in each of two rounds each processor sends and receives exactly one block column per one communication start-up. Two rounds are needed to ensure that each pair of block columns is met in some processor exactly ones. Having p processors, two rounds can be done using $2(p-1)$ sends and receives with respect to each processor. The overall transferred data volume is n^2 double real values, i.e., the complete matrix A. 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. Also, the complexity of computing all scalar products between all columns of all blocks is $\approx 2pn^3/4p^2 = n^3/2p$, which is, for $p > 4$, even larger that the computational complexity $\approx 10n^3/p^3$ of one iteration step of the accelerated OSBJA (see the estimate (13) in [22]).

One possibility to decrease the computational and communication complexity would be to choose randomly only $s \ll n/2p$ columns from each block column and to send/receive only this restricted set of columns. This approach decreases the amount of scalar products and the amount of transferred data, but it does not decrease the number of start-ups needed at the beginning of each data transfer. Also, the random choice of s columns leads to some sort of heuristics, and it is not clear if this is better than , e.g., the row cyclic ordering of subproblems.

Can we somehow decrease the communication complexity when looking for the ordering of p parallel subproblems at the beginning of each iteration step? The solution might be in the consideration of block columns as the *orthogonal bases* of certain linear subspaces of dimension $n/(2p)$ in the original linear subspace $\mathbb{R}^n$ and in the description of the *geometric position* of such subspaces with respect to the canonical basis of $\mathbb{R}^n$. We will describe our ideas in the next subsection.

3.2 Mutual position of block columns

After the initialization, the block columns contain mutually orthogonal columns. Suppose that each processor contains exactly two block columns (this is not substantial for the following discussion). Moreover, suppose that the columns in each block column are *normalized* so that each has the unit Euclidean norm. Each processor then stores two vectors of dimension $n/2p \equiv k$. (Recall that these norms are estimations of singular values of matrix A.) Hence, each column block is the *orthonormal basis* of the k-dimensional subspace which is spanned by the column vectors of a given block column.

Now take two block columns X, Y which should be orthogonalized in a given parallel iteration step. Having p processors, our goal is to choose p pairs of those block columns that are maximally *inclined* to each other, i.e., their mutual position differs maximally from the orthogonal one.

This vague description can be made mathematically correct using the notion of *principal angles*

between two k-dimensional subspaces spanned by two block columns X, Y. Since X and Y are orthonormal bases of two subspaces with an equal dimension, the cosines of principal angles are defined as the singular values of the matrix X^TY. Let $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_k$ be k singular values of the $k \times k$ matrix X^TY. Then the principal angles $\theta_1 \leq \theta_2 \leq \ldots \leq \theta_k$, $\theta_i \in [0, \pi/2]$, $1 \leq i \leq k$, are defined as:

$$\theta_i = \arccos(\sigma_i), \;\; 1 \leq i \leq k. \tag{11}$$

Since X and Y have orthonormal columns, all singular values of X^TY are in the interval $[0, 1]$, so that the relation (11) is well defined.

We are interested in the *smallest* principal angle θ_1, i.e., in the *largest* cosine (largest singular value) σ_1. When $\sigma_1 = 0$, then all $\sigma_i = 0$, $2 \leq i \leq k$, and two block columns X and Y are perfectly orthogonal; we do not need to orthogonalize them explicitly. On the other hand, when $\sigma_1 \approx 1$, we can not say too much about the values of other cosines (without their explicit computation), but we know for sure that at least one dimension is so to say 'common' for both subspaces, so that we need to explicitly orthogonalize them, i.e., we choose such a pair for the parallel ordering. In other words, the weights of the complete graph in Fig. 1 will be maximal singular value of the matrix X^TY for each pair of block columns (X, Y).

However, this approach means that we must explicitly compute the matrix X^TY. When two block columns X and Y are placed in two different processors, we can either compute this matrix product in parallel (but for each pair of block columns), or store both blocks in one processor and compute the matrix product locally using the LAPACK library. Afterwards, we must compute (or at least somehow estimate) the largest singular value to get our weights for the maximal perfect matching. In both cases we need again too much communication at the beginning of each parallel iteration step to construct the actual parallel ordering for that step.

Is it possible to reliably estimate σ_1 of X^TY for a pair (X, Y) of block columns based *only on local information* gained by processing the block columns *separately*? If so, this approach would solve the communication (and computational) bottleneck at the beginning of each parallel iteration step.

To estimate the largest singular value σ_1 of the $k \times k$ matrix X^TY, one can use, for example, the power method applied to the Jordan-Wielandt matrix C,

$$C \equiv \begin{pmatrix} 0 & X^TY \\ Y^TX & 0 \end{pmatrix}.$$

It is well known that the eigenvalues of the $2k \times 2k$ matrix C are $\pm\sigma_1, \pm\sigma_2, \ldots, \pm\sigma_k$. Notice that there are k pairs of eigenvalues with the same absolute value. However, we are interested neither in estimating the multiplicity of eigenvalues, nor in the approximations of eigenspaces; we need only an estimate of $\sigma_1 > 0$. Therefore, we will use the power method in its simplest form, discarding the estimate of the eigenvector.

The power method requires matrix-vector products of type X^TYu and Y^TXv where the composed vector $(u^T, v^T)^T$ is of unit length. We do not want to form matrices X^TY and Y^TX explicitly (since the column blocks X and Y are in different processors). Instead, the matrix-vector multiplications can be computed sequentially as $X^T(Yu)$ and $Y^T(Xv)$ by sending only two k-dimensional vectors between processors. For example, let the block column X and Y reside in processor 0 and 1, respectively. Then processor 0 generates (or normalizes) the unit

vector $(u^T, v^T)^T$, computes Xv and sends u and Xv to processor 1. Processor 1 computes Yu and $Y^T(Xv)$ and sends both vectors back to processor 0. Processor 0 finishes the current step of the power method by computing $X^T(Yu)$, collecting both parts into one vector $w = ((Y^T(Xv))^T, (X^T(Yu))^T)^T$, finding $\alpha = \max_{1 \le i \le 2k} |w_i|$ (which is the estimate of σ_1), dividing w/α and scaling w to unit length. Hence, each step of the power method requires exchange of two vectors of length k between two processors that store the required block columns. This work can be organized in parallel for all pairs of block columns residing in two different processors, where it needs to be defined, for each pair of processors, who is the 'master' (doing the job of processors 0 in the above example) and who is the 'slave' (doing only two matrix-vector multiplications as processor 1). The number of steps of the power method can be *fixed* (depending on the width k of block columns), so that no computation of residuals and no testing of the convergence criterion will be needed.

At the end, all estimates of all σ_1's for all pairs of different blocks will be sent to processor 0. These estimates are the weights for the complete graph, and processor 0 will compute the maximum perfect matching on this graph, which will give those pairs of block columns that *may be* mostly inclined to each other (i.e., their minimal angle is close to 0, but we cannot say anything about other angles). On the other hand, when σ_1 of is close to 0, we know for sure that *all* principal angles are close to $\pi/2$ and it is not efficient to orthogonalize those two block columns in the next parallel iteration step. In this sense, the weights based on estimates of σ_1 lead to the heuristics that should be tested in numerical experiments. Finally, processor 0 will broadcast the computed ordering to all processors, appropriate column blocks will be collected in processors and the SVD computations of the next parallel iteration step can begin.

Another approach to this problem is based on description of each block column with respect to the canonical basis of $\mathbb{R}^n$. The local information about an $n \times k$ block column X is the geometric position of the k-dimensional vector subspace it defines in the whole space $\mathbb{R}^n$. This position can be described by means of n angles between span(X) and canonical vectors e_i, $1 \le i \le n$. (Recall that the canonical vector e_i in $\mathbb{R}^n$ has dimension n, its ith component is 1 and all other components are 0.) Since X has k orthonormal columns, the orthogonal projection P_X on the subspace span(X) is defined by the $n \times n$ idempotent matrix $P_X = XX^T$. The image of e_i under this projection is

$$P_X e_i = X(X^T e_i) = XX(i,:)^T,$$

where $X(i,:)$ is the ith row of X. Then the cosine α_i of the acute angle $\angle(e_i, X)$ between e_i and span(X) is defined by

$$\alpha_i \equiv \cos \angle(e_i, X) = \|P_X e_i\| = \|(X(i,:)\|. \tag{12}$$

Thus, to compute all α_i, $1 \le i \le n$, we need to compute all n Euclidean row norms of the block column X, and this can be done *locally* in a processor storing X. Since each row is of dimension k, the computational complexity is $O(nk^2)$, which is low for $k \ll n$.

Hence, the position of each column block in $\mathbb{R}^n$ is defined by the n-tuple $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ that describes its position with respect to the canonical basis $[e_1, e_2, \ldots, e_n]$. This information can be computed *locally* in each processor for all block columns it stores. Taking two such n-tuples, say, $\{\alpha_i\}$ and $\{\beta_i\}$, for two different block columns X and Y stored in two different processors, can we get an estimate of some principal angle between them?

We can use a well-known relationship between the *largest* principal angle θ_n and the spectral norm of the difference of orthogonal projectors P_X and P_Y:

$$\sin\theta_n = \|P_X - P_Y\|.$$

Now we can bound $\|P_X - P_Y\|$ from below using the definition of the spectral norm and (12):

$$\begin{aligned}\|P_X - P_Y\| &= \max_{\|z\|=1} \|P_X z - P_Y z\| \geq \max_{1\leq i\leq n}\{\|P_X e_i - P_Y e_i\|\} \\ &\geq \max_{1\leq i\leq n}\{|\|P_X e_i\| - \|P_Y e_i\||\} = \max_{1\leq i\leq n}\{|\alpha_i - \beta_i|\}.\end{aligned}$$

Consequently, we obtain the lower bound for $\sin\theta_n$ using only *local* information from block columns X and Y in the form of two n-tuples $\{\alpha_i\}$ and $\{\beta_i\}$:

$$\sin\theta_n \geq \max_{1\leq i\leq n}\{|\alpha_i - \beta_i|\}. \tag{13}$$

At the beginning of a parallel iteration step, each processor computes the n-tuples $\{\alpha_i\}$ for all block columns it stores and sends them to processor 0. For all different pairs of block columns (X, Y), processor 0 computes the lower bound on the right-hand side of (13), which is the weight in the complete graph with r nodes (ℓ is the blocking factor, usually $r = 2p$, where p is the number of processors). Then processor 0 computes *the minimal weight perfect matching* that will give $r/2 = p$ pairs of block columns that *can be* minimally inclined to each other (their maximal principal angle can be close to zero, so that all other angles could be close to zero). Thus, the ordering of subproblems is defined in processor 0 that is broadcast to all processors and the computation of p parallel EVDs can start.

Notice that we can speak only about the *possibility* of minimal inclination when the weight is small, because the weights are only *lower* bounds for $\sin\theta_n$. When the weight between block columns X and Y is large, say, $1-\delta$ where $0 < \delta \ll 1$, then we know that the maximum angle is near $\pi/2$ (but we have no information about smaller principal angles). However, when the weight is small, the maximum principal angle θ_1 *can* be small (but it can be also large). In other words, the large weights defined by the right-hand side of relation (13) allow us to find those pairs of column blocks that are nearly orthogonal with respect to the largest principal angle, and we will not orthogonalize them in a given parallel iteration step. But the small weights carry the risk that respective pairs of column blocks *may* be orthogonal to each other and we choose them for our parallel ordering. Now, this risk is very small at the beginning of the iterative process, because it can be expected that the mutual position of pairs of block columns will be random and only very seldom will be two blocks exactly orthogonal to each other. As the iterative computation proceeds, any two column blocks will be more and more orthogonal to each other with respect to *all* principal angles.

It is clear from above, that because we work only with the lower estimate for the largest principal angle θ_n, the proposed approach can be regarded as *a heuristics*. It should be tested by numerical experiments and compared with the (row or column) cyclic ordering.

3.3 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

The main contribution of this report is the design of two variants of dynamic ordering for the one-sided parallel block-Jacobi SVD algorithm using only local information from distributed block columns at the beginning of each parallel iteration step. The first method is based on estimating the cosine of the smallest principal angle by the power method (implemented in parallel), while the second one uses the local information for estimating the sine of the largest principal angle. Both approaches can be regarded as heuristics, but the first approach seems to give more accurate estimates. However, only numerical experiments will show their efficiency as compared to some classic parallel block orderings (e.g., block row/column cyclic).

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] G. Okša and M. Vajteršic, *Parallel one-sided block Jacobi SVD algorithm: II. Implementation*, Technical report, Salzburg University, Salzburg, Austria, December 2007.

[23] 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.

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