UNIVERSITÄT SALZBURG

Generalization of the Dynamic Ordering for the One-Sided Block Jacobi SVD Algorithm: II. Implementation

Martin Bečka[a] Marián Vajteršic

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

Technical Report 2008-03 December 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: II. Implementation

Martin Bečka* and Marián Vajteršic†

Abstract. *We have designed, implemented and tested (by simulation on a serial computer) the new dynamic ordering for the parallel one-sided block-Jacobi SVD algorithm. Our idea is based on the estimation of the cosines of principal angles between two block columns X and Y of the same width without explicitly forming the matrix product X^TY (or Y^TX) and computing its SVD. Instead, we propose to use a fixed number $2q$ of iterations in the Lanczos algorithm applied to the symmetric 2x2 block Jordan-Wielandt matrix with zero diagonal blocks, 21-block X^TY and 12-block Y^TX; the order of the Jordan-Wielandt matrix is the sum of the block column widths. However, the matrix blocks X^TY and Y^TX are never formed explicitly; the needed matrix-vector multiplications are computed exchanging intermediate product vectors between two processors that host the block column X and Y. After computing $2q$ iterations, the Frobenius norm of an auxiliary tridiagonal matrix of order $2q$ estimates the square root of twice the sum of squares of q largest cosines (representing q smallest principal angles) between X and Y. In the parallel algorithm using p processors, these weights can be used for choosing p pairs of block columns, which are far from orthogonality with respect to those q smallest angles. We show how to implement this new parallel ordering in the distributed paradigm of parallel computing using the Message Passing Interface (MPI). First numerical results obtained by simulation show that the one-sided parallel dynamic ordering can lead to a substantial decrease of the number of parallel iteration steps needed for the convergence as compared to a cyclic ordering.*

1 Introduction

This report is the fourth 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 [23] 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 [24], the parallel implementation and overall

*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.

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. Finally, the third part [6] contains the discussion about possible extensions of the dynamic ordering for the two-sided block-Jacobi parallel SVD algorithm(see [4]) to the case of the one-sided block-Jacobi parallel SVD algorithm.

When a matrix A of order $m \times n$ is cut columnwise with blocking factor $l = 2p$, where p is the number of processors, exactly two block columns (each of width n/l) are stored in each processor (here we assume that l divides n). The one-sided block-Jacobi SVD algorithm then mutually orthogonalizes any two block columns (including the mutual orthogonalization of matrix columns inside the block columns). The aim is to achieve orthogonality between any two matrix columns to a given accuracy. Then the norms of such columns are singular values, the columns after normalization are left singular vectors, i.e. they are computed easily *a posteriori*.

Having p processors, we need p pairs of block columns that will be mutually orthogonalized. The main question is, how to choose those p pairs. Usually, some fixed, prescribed, *cyclic* strategy is chosen in the form of a list of block columns that need to meet in some processor. In this case, a *sweep* can be defined, during which a given block column is orthogonalized against all remaining block columns exactly once. Many such parallel orderings exist – see [2, 3].

A big disadvantage of any fixed ordering is the fact that the actual status of orthogonality is usually checked only after a whole sweep and one has no information about the quality of this process at the beginning of a parallel iteration step. In other words, in a given parallel iteration step one can try to orthogonalize some mutually 'almost orthogonal' block columns while neglecting pairs with small principal angles. It is clear, at least intuitively, that orthogonalizing block columns with small principal angles first would mean to eliminate the 'worst' pairs first, and this would mean (hopefully) the faster convergence of the whole algorithm as compared with any fixed, cyclic ordering.

Hence, the main question is how to choose p pairs of block columns with smallest principal angles among all $l(l-1)/2 = p(2p-1)$ pairs. The obvious, but very naive way is to compute, for each column block X, all possible matrix products X^TY, then to compute the SVD of X^TY and look at the singular values, which are the cosines of acute principal angles (the smaller angle, the larger cosine). When the block columns are distributed in processors, to compute matrix products X^TY for each two different block columns X and Y means to move block columns across processors, i.e., it leads to heavy communication at the beginning of each parallel iteration step. Besides that, one needs to compute many matrix products and SVDs. And, at the end of the day, when p pairs of column blocks with smallest principal angles are chosen, they must meet in processors, which means yet another communication.

Our idea is different. At the beginning of each parallel iteration step, the column blocks remain stationary in processors. The largest cosines of, say, q smallest principal angles between any two column blocks X and Y are *estimated* by the Lanczos algorithm applied on the symmetric Jordan-Wielandt matrix *without* explicitly forming X^TY and Y^TX. After, say, $2q$ iterations the square the Frobenius norm of the auxiliary tridiagonal matrix of order $2q$ provides a good estimate of the square root of twice the sum of squares of q largest cosines (representing q smallest principal angles) between X and Y. When the blocking factor is $l = 2p$, we can use

these weights in the maximum perfect matching problem on a complete graph with $2p$ vertices to find p pairs of block columns that are 'least' mutually orthogonal. This is a direct extension of the dynamic ordering for the parallel two-sided block-Jacobi SVD algorithm; see [4].

The report is organized as follows. In section 2 we shortly repeat the main ideas about the OSBJA from [23]. Section 4 describes in details the new parallel ordering strategy, which can be called as the dynamic ordering for the one-sided case. We describe in detail its parallel implementation under the distributed paradigm of parallel computing using the Message Passing Interface (MPI) library.

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 for the complex case.

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

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

where the width of A_i is $n_i,\ 1 \leq i \leq l$, so that $n_1 + n_2 + \cdots + n_l = n$. In order to provide the maximal balanced width of columns, the first $(n \bmod l)$ block columns have width $\lceil n/l \rceil$ and the remaining block columns have width $\lfloor n/l \rfloor$.

The OSBJA can be written as an iterative process:

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

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

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

where the unidentified matrix blocks are zero. The purpose of matrix multiplication $A^{(k)} R^{(k)}$ in (1) is to mutually orthogonalize the columns between column-blocks i and j of $A^{(k)}$. The matrix blocks $R_{ii}^{(k)}$ and $R_{jj}^{(k)}$ 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 $\sum_{s=j+1}^{l} n_s$, respectively. The orthogonal matrix

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

of order $n_i + n_j$ is called the *pivot submatrix* of $R^{(k)}$ at step k. During the iterative process (1), two index functions are defined: $i = i(k),\ j = j(k)$ whereby $1 \leq i < j \leq l$. 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 $\mathcal{F} : \{0, 1, \ldots\} \to \mathbf{P}_r = \{(i, j) : 1 \leq i < j \leq l\}$. If $\mathbf{O} = \{(i_1, j_1), (i_2, j_2), \ldots, (i_{N(l)}, j_{N(l)})\}$ is some ordering of $\mathbf{P}_r$ with $N(l) = l\,(l-1)/2$, then the *cyclic* strategy is defined by:

If $k \equiv l - 1 \mod N(l)$ then $(i(k), j(k)) = (i_s, j_s)$ for $1 \leq s \leq N(l)$.

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 $N(l)$ 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:

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

   and the eigenvector matrix $\hat{R}^{(k)}$ is partitioned according to (3). The matrix $\hat{R}^{(k)}$ defines the orthogonal transformation $R^{(k)}$ 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 [9, 10, 11, 22]. 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). Mathematical details regarding the preprocessing step can be found in [23].

3 Dynamic ordering in the two-sided algorithm

At the beginning of each parallel iteration step it is necessary to choose $p = l/2$ pivot pairs (i, j) that define, for p processors, p subtasks (A_i, A_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 [18]. 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 [18] 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 [19], 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 $l = 2p$ 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 [15], 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 $l = 2p$ vertices - hence the number of vertices is equal to the blocking factor; see Fig. 1. In the two-sided block Jacobi method, each edge is weighted by the non-negative weight $\|A_{ij}\|_{\mathrm{F}}^2 + \|A_{ji}\|_{\mathrm{F}}^2$, where $\|A_{uv}\|_{\mathrm{F}}^2$ 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

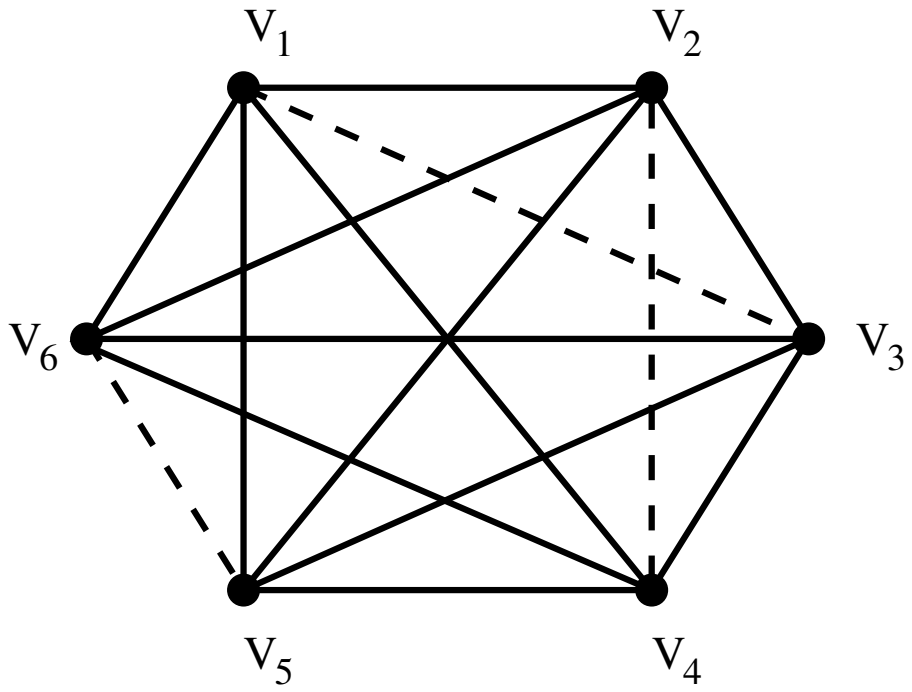

Figure 1: Maximum-weight perfect matching on a complete graph for $r = 6$. The chosen edges are dashed.

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 $l/2$ 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 $2l$ 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-weight perfect matching costs only one `MPI_ALLGATHER`.

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. (We note that after the first iteration the columns *within* each block column A_i remain mutually orthogonal.) Now, the principle of the maximum-weight perfect matching can be easily extended also to the paradigm of mutual orthogonality of block columns. An ideal case is the mutual orthogonality of *all* pairs of block columns. The departure from this ideal case can be measured either by a sum of squares of cosines of angles between all pairs of columns in two given block columns, or by the maximum cosine of these angles. Hence, for each pair of block

columns (A_i, A_j), we can define the departure from their mutual orthogonality by

$$w_{ij} = \sum_{u,v=1}^{n/l} \cos^2 \angle(a_u^{(i)}, a_v^{(j)}) \quad \text{or} \quad w_{ij} = \max_{1 \le u,v \le n/l} \{\cos^2 \angle(a_u^{(i)}, a_v^{(j)})\}, \tag{6}$$

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

But, in contrast to the two-sided block Jacobi method, the weights defined in (6) 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 (6) 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 (6) in such a way that each matrix block column A_i must meet each matrix block column A_j, $j \neq i$, in some processor, in which the updated weight w_{ij} is computed according to (6). Since there are exactly two block columns in each processor (each with $n/2p$ columns for the blocking factor $l = 2p$), 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 $\approx 2pn^3/4p^2 = n^3/2p$ (considering $m = n$), 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 [24]).

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, the amount of transferred data as well as 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 $\mathbb{R}^m$. Having two block columns, i.e., two orthogonal bases of two linear subspaces of the same dimension k, the mutual position of them is defined by so called *principal angles*. In the next subsection we show that the twice the sum of squares of q largest cosines of these angles can be estimated by the Frobenius norm of certain tridiagonal, symmetric matrix T_{2q}, which comes out from the Lanczos process applied to the special Jordan-Wielandt matrix.

4 Dynamic ordering in the one-sided algorithm

After the first iteration, 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 A_i, A_j 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 A_i, A_j. Since A_i and A_j are orthonormal bases of two subspaces with the equal dimension, the cosines of principal angles are defined as the singular values of the matrix $A_i^T A_j$. Let $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_k$ be k singular values of the $k \times k$ matrix $A_i^T A_j$. 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:

$$\theta_i = \arccos(\sigma_i), \ 1 \leq i \leq k. \tag{7}$$

Since A_i and A_j have orthonormal columns, all singular values of $A_i^T A_j$ are in the interval $[0, 1]$, so that the relation (7) is well defined.

We are interested in, say, q *smallest* principal angles, i.e., in q *largest* cosines (largest singular values) $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_q$. When $\sigma_1 = 0$, then all $\sigma_i = 0$, $2 \leq i \leq k$, and two block columns A_i and A_j are perfectly orthogonal; we do not need to orthogonalize them explicitly. On the other hand, when all σ_k are significantly greater than 0, column blocks A_i and A_j are certainly far from the mutual orthogonality. Now, for large matrices A of order, say, 10^3 - 10^4, and modest number of processors, of order, say, 10^1, the width k of column blocks can be quite large, say, of order 10^2. Hence, we can choose s much smaller - say, of order 10^0 - 10^1.

However, it seems this approach means that we must explicitly compute the matrix $A_i^T A_j$. When two block columns A_i and A_j 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 m singular values and afterwards compute some function of them (e.g., the sum of their squares) to get our weight w_{ij} for the maximum-weight 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.

To estimate q largest singular value of the $k \times k$ matrix $A_i^T A_j$, we suggest to use the Lanczos process applied to the symmetric Jordan-Wielandt matrix C,

$$C \equiv \begin{pmatrix} 0 & A_i^T A_j \\ A_j^T A_i & 0 \end{pmatrix}. \tag{8}$$

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.

It follows from the theory of Krylov space methods that the Lanzcos algorithm applied to a symmetric matrix is the good iterative method for estimating its largest (in absolute value) eigenvalues. This algorithm, applied to the symmetric Jordan-Wielandt matrix C, is listed below as Algorithm A1 for a *fixed* number of iteration steps $2q$.

A1: Lanczos algorithm for the symmetric Jordan-Wielandt matrix C

1. Choose integer $2q$ and the vector x_0 of length $2k$, and compute: $\beta_1 = \|x_0\|;\ v_1 = x_0/\beta$;
2. **for** $(\ell = 1;\ \ell < 2q;\ \ell++)$ {
3. $\quad w_\ell = Cv_\ell$;
4. $\quad$ **if** $(\ell\,!= 1)\ w_\ell = w_\ell - \beta_\ell v_{\ell-1}$;
5. $\quad \alpha_\ell = w_\ell^T v_\ell$;
6. $\quad w_\ell = w_\ell - \alpha_\ell v_\ell$;
7. $\quad \beta_{\ell+1} = \|w_\ell\|$;
8. $\quad$ **if** $(\beta_{\ell+1}\,!= 0)\ v_{\ell+1} = w_\ell/\beta_{\ell+1}$;
9. $\quad$ **if** $(\beta_{\ell+1}\ == 0)\ \ell = 2q;\quad$ }
10. Set: $T_{2q} = \mathrm{tridiag}(\beta_{s+1}, \alpha_s, \beta_{s+1}), \quad s = 1, \ldots, 2q$.
11. Compute the Frobenius norm of T_{2q}. □

Steps 2-9 constitute an adaptation of the Arnoldi method for a symmetric matrix. Due to the special structure of C (see Eq. (8)), the matrix-vector product in step 2 is applied in two substeps: $w_\ell^1 = A_i^T A_j v_\ell^1,\ w_\ell^2 = A_j^T A_i v_\ell^2$, where $v_\ell = (v_\ell^{1T}, v_\ell^{2T})^T$ and $w_\ell = (w_\ell^{1T}, w_\ell^{2T})^T$.

The result is the orthonormal basis of the Krylov subspace $\mathcal{K}_m(C, x_0)$ formed by vectors v_ℓ, $1 \le \ell \le 2q$. Besides that, the coefficients α_ℓ and β_ℓ are computed that are stored in the symmetric, tri-diagonal matrix T_{2q} (step no. 10).

In our application, the orthonormal vectors v_ℓ are not important (they are used, for example, in the solution of a linear system of equations). What is most important, is the square of the Frobenius norm of T_{2q} written in terms of its eigenvalues ω_ℓ, $1 \le \ell \le 2q$ (they are known as Ritz values):

$$\|T_{2q}\|_F^2 = \sum_{\ell=1}^{2q} \omega_\ell^2.$$

As already mentioned, the $2q$ Ritz values approximate reasonably well $2q$ largest (in the absolute value) eigenvalues λ_ℓ of the Jordan-Wielandt matrix C. However, in our application, there are exactly two eigenvalues of C with the same absolute value (with opposite signs) and they are related to the squares of singular values of $A_i^T A_j$. Therefore,

$$\|T_{2q}\|_F^2 = \sum_{\ell=1}^{2q} \omega_\ell^2 \approx \sum_{\ell=1}^{2q} \lambda_\ell^2 = 2\sum_{\ell=1}^{q} \sigma_\ell^2 = 2\sum_{\ell=1}^{q} \cos^2(\theta_\ell),$$

i.e., the Frobenius norm of T_{2q} can be used as the (good) approximation for the sum of q largest cosines defining q *smallest* principal angles between subspaces $\mathrm{span}(A_i)$ and $\mathrm{span}(A_j)$. In other words, we have found a relatively easily computable weight w_{ij} for the maximum perfect matching in the one-sided block-Jacobi method. We stress that we do *not* need to compute the Ritz values (i.e., the eigendecomposition of T_m) - the Frobenius norm squared is enough!

Moreover, note that in our application T_{2q} is not needed in its explicit form. All that is needed is the square of its Frobenius norm. Since

$$w_{ij} = \|T_{2q}\|_F^2 = \sum_{\ell=1}^{2q} \alpha_\ell^2 + 2\sum_{\ell=2}^{2q} \beta_\ell^2,$$

$\|T_{2q}\|_F^2$ can be computed recursively immediately after computing α_ℓ and $\beta_{\ell+1}$ in the ℓth iteration step of the Lanczos algorithm. Indeed, this is a very simple computation!

Note that the weight w_{ij} takes into account the *actual* mutual position of two subspaces $\mathrm{span}(A_i)$ and $\mathrm{span}(A_j)$. Therefore, we can simply choose 'worst' p pairs of column blocks for their parallel orthogonalization by choosing the pairs with highest values of w_{ij}. This is an analogy to the two-sided dynamic ordering where the actual Frobenius norm of the off-diagonal blocks was taken into account. Therefore, the above described ordering can be defined as the *one-sided dynamic ordering*. To choose p 'worst' block columns for the parallel orthogonalization, the same maximum-weight perfect matching algorithm on the complete graph with r vertices and weights w_{ij} can be used as in the two-sided case - see [4].

We have just described, how we can quite cheaply compute the weight w_{ij} that is the function of q (estimated) largest cosines of principal angles between subspaces $\mathrm{span}(A_i)$ and $\mathrm{span}(A_j)$. The larger is the weight, the lower is the degree of mutual orthogonality between these two subspaces. However, at the beginning of each parallel iteration step we have to compute those weights for *all* $l(l-1)/2$ pairs of block columns of matrix A. In the next subsection we describe how this computation can be done in parallel *without* sending/receiving whole block columns and *without* computing explicitly the matrix products $A_i^T A_j$.

4.1 Parallel implementation of the one-sided dynamic ordering

Having p processors and $l = 2p$ block columns of matrix A, each processor stores 2 block columns. This means that there are $l(l-1)/2 = p(2p-1)$ pairs of block columns. Suppose that p pairs of block columns stored currently in processors have been just mutually orthogonalized.

Then there are $2p(p-1)$ pairs that are not mutually orthogonalized. From this number we need to choose p pairs at the beginning of each parallel iteration step; these pairs should be the 'worst' ones, i.e., those that differ at most from the mutual orthogonality. In other words, at the beginning of each parallel iteration step it is necessary to run $2p(p-1)$ Lanczos processes for a fixed number $2q$ of iterations to get weights w_{ij} defined above.

It is possible to organize the computation of Lanczos processes in parallel as follows. Let us number the processors from P_0 to P_{p-1} in a process row. Let us denote 2 block columns in each processor by left (L) and right (R). Pairings of block columns are defined in a systematic way. The block L from a given processor P_i is paired with all left blocks from all processors P_j to the right of P_i (i.e., $j > i$), and with all right blocks from all processors P_j to the left of P_i (i.e., $j < i$). Similarly, the block R from a given processor P_i is paired with all right blocks from all processors P_j to the right of P_i (i.e., $j > i$), and with all left blocks from all processors P_j to the left of P_i (i.e., $j < i$). It is easy to see that such an organization defines exactly $2(p-1)$ pairings for each processor that contain either block L or R from a given processor. Moreover, all block columns are covered and the covering is optimally *balanced* in the sense that each processors has to do the same computational work. We say that each processor is the *master* for its own $2(p-1)$ Lanczos processes.

For one pair of block columns (X, Y), the corresponding Lanczos process 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 by sending only m and k-dimensional vectors between processors. For example, let the block column X and Y reside in processor P_0 and P_1, respectively. Then processor P_0 (which is the master) generates (or normalizes) the unit vector $(u^T, v^T)^T$, computes Xu and sends and Xu and v to processor P_1. Processor P_1 computes Yv and $Y^T(Xu)$ and sends both vectors back to processor P_0. Processor P_0 finishes the current step of the Lanczos method by computing $X^T(Yv)$, scalars α and β and by updating the Frobenius norm of the matrix T_{2q}. Hence, $2q$ iterations of one Lanczos process requires two point-to point communications for sending/receiving m and k-dimensional vectors (these two vectors can be sent in one message).

This is the work for *one* Lanczos process. In fact, these computations must be done for all $2(p-1)$ Lanczos processes for which processor P_0 is the master and this work is serialized inside each processor. However, all processors serve as the master exactly for $2(p-1)$ different Lanczos processes each and they compute these tasks in parallel. Moreover, in our implementation, one data structure is used for all Lanczos processes and each processor receives the whole data structure (not only the relevant part of it). At the same time, each processor stores the information about two block columns that it currently stores, and about all Lanczos processes for which it serves as the master. Therefore, each processor can read/write from/to the data structure the data/results of its own computations for all Lanczos processes for which it is the master (matrix-vector products, updates of Frobenius norms). To communicate the data structure to all processors, the MPI collective communication `ALLTOALL` is used. Two such communications are needed per one (parallel) iteration, i.e., together $4q$ collective communications are needed. These communications serve also like the global *synchronization* steps in the whole computation. Finally, one `MPI_ALLGATHER` is used to gather all computed weights on each processor.

At the end of computation with Lanczos processes, all processors contain all weights w_{ij} for all block column pairs (via one `MPI_ALLGATHER` call), which are simply the squares of Frobenius norms of all matrices T_{2q} produced in all Lanczos processes. Therefore, each processor can compute the maximum-weight perfect matching and the resulting parallel ordering; the algorithm is the same as for the parallel two-sided block-Jacobi method (see [4]). For transferring the chosen pairs in processors, the optimal parallel scheduling is used in order to send/receive at most one block column (see [5]) for any processor.

The stopping criterion of the iteration process is based on the maximum value of currently computed weights w_{ij}. When using the double precision with the machine precision ϵ, the convergence is reached when

$$\max_{i,j} w_{ij} < m\, 2q\, \epsilon, \tag{9}$$

where m is the matrix order and q is the number of angles computed in Lanczos processes. In other words, the computation is finished when the cosines of q largest principal angles between all column blocks are 'sufficiently' small. This is the *global* stopping criterion and it seems that it works very well (with one exception - see the next section). The *local* stopping criterion (which decide when it is *not* necessary to orthogonalize a given pair of block columns) is similar to the global stopping criterion

$$w_{ij} < m\, 2q\, \epsilon. \tag{10}$$

5 First numerical results: Simulation on a serial computer

Since the cluster of personal computers at the Salzburg University is not available at the moment, we have simulated the parallel one-sided block-Jacobi algorithm with the new dynamic ordering on a serial computer. However, our serial simulation enables to define and use all parameters normally used in 'true' parallel processing - namely, the number of processors p. All computations were performed using the IEEE standard double precision floating point arithmetic with the machine precision $\epsilon_M \approx 2.22 \times 10^{-16}$. The global stopping criterion was of size 10^{-13}. Of course, the total execution time of the simulated parallel algorithm is of a questionable value and we do not include it here. But the number of parallel iteration steps n_{it}, needed for the convergence, is a reliable measure of efficiency (or inefficiency) and should be preserved also in computations on a 'true' parallel architecture.

In our simulations, we have used 6 various distributions of singular values (SVs) of randomly generated matrices. A distribution of SVs is described by the parameter `mode`. For a condition number κ, the SVs were always contained in the interval $[\kappa^{-1}, 1]$. The value `mode` $= 1$ corresponds to a multiple minimal SV, `mode` $= 2$ defines a multiple maximal SV, `mode` $= 3$ describes a geometric sequence of SVs. Additional distributions of SVs are used with `mode` $= 4$ defining an arithmetic sequence of SVs, `mode` $= 5$ defining the SVs as random numbers such that their logarithms are uniformly distributed, and, finally, `mode` $= 6$ setting the SVs to random numbers from the same distribution as the rest of a matrix (i.e., in our case they were normally distributed).

Table 1: Performance for $n = 2000,\ p = 4,\ \kappa = 10^1$ and $2q = 4$. For the one-sided block Jacobi algorithm, the computations were simulated on a serial computer. For the two-sided method, the computations were run on the cluster 'Gaisberg' in Salzburg.

`mode`	1	2	3	4	5	6
n_{it} (one-sided)	3	3	43	40	42	–
n_{it} (two-sided)	130	127	41	42	44	43

Table 1 compares the results of simulation (for the one-sided method with dynamic ordering) and true parallel computation (for the two-sided method with dynamic ordering, see [4]) for a random matrix A of order $n = 2000$, $p = 4$ processors. The one-sided dynamic ordering uses only 4 iterations in Lanczos method so that only 2 maximal principal cosines are approximated. The global stopping criterion was the same for both methods.

When looking at n_{it}, the one-sided method with dynamic ordering clearly outperforms the two-sided one for `mode` $= 1$ and 2 by two orders of magnitude! Recall that here we do not use any pre-processing of the matrix before its SVD. For the two-sided method, $O(1)$ parallel iterations for `mode` $= 1,\ 2$ can be achieved only by computing the QR factorization with column pivoting of a matrix prior to its SVD and then applying the SVD to the R-factor - see [23].

For other modes the values of n_{it} are almost the same. With respect to the computational complexities of the one-sided and the two-sided method, the efficient computation of weights for ordering and almost equal number of iterations for both, one can expect the one-sided method is almost two times faster than its two-sided counterpart. We note that 42 parallel iteration means $42/(l-1) = 6$ sweeps of the classical cyclic one-sided Jacobi method.

Currently, we have some problems with `mode` $= 6$ for the one-sided method. The convergence stagnates, it helps to decrease q to 1 when the weights are getting very small. The problem is probably related to the computation with values closed to zero (we compute cosine squared, what can be very small value for almost right angles).

The quality of simulated computations for the one-sided method with dynamic ordering can be documented by defining three 'quality indices' $Q_1,\ Q_2$ and Q_3:

$$Q_1 = \frac{\|A - U\Sigma V^T\|_F}{\|A\|_F}, \quad Q_2 = \frac{\|I - U^T U\|_F}{\sqrt{n}}, \quad Q_3 = \frac{\|I - V^T V\|_F}{\sqrt{n}}. \tag{11}$$

Here, the matrices U, Σ and V are the *computed* left singular vectors, singular values and right singular vectors, respectively. Recall that the one-sided method computes iteratively only the singular values and right singular vectors. Left singular vectors u_i are computed *a posteriori* by

$$u_i = \sigma_i^{-1} A v_i,\ 1 \le i \le n. \tag{12}$$

Thus, the index Q_1 measures the overall relative accuracy of the SVD computation, whereas the indices Q_2 and Q_3 give the information about the relative loss of orthogonality of the computed left and right singular vectors, respectively.

Table 2 brings the values of $Q_1,\ Q_2$ and Q_3 for the simulated parallel one-sided block-Jacobi

Table 2: Numerical accuracy of simulated computations for the one-sided method with dynamic ordering.

`mode`	1	2	3	4	5	6
Q_1	1.43×10^{-15}	1.56×10^{-15}	1.71×10^{-15}	1.31×10^{-15}	1.67×10^{-15}	–
Q_2	9.97×10^{-15}	9.40×10^{-15}	8.11×10^{-14}	8.23×10^{-14}	2.49×10^{-14}	–
Q_3	6.58×10^{-15}	6.89×10^{-15}	3.41×10^{-14}	3.41×10^{-14}	3.30×10^{-14}	–

method with dynamic ordering. The simulation parameters are the same as mentioned in the caption of Table 1.

In general, the accuracy of our new one-sided method seems to be excellent. The matrix A can be recovered from its computed SVD to the relative accuracy of 10^{-15} *regardless* to a value of `mode`. With respect to the loss of orthogonality, one can perhaps distinguish between 'easy' cases (`mode` $= 1,\ 2$) and 'hard' cases (`mode` ≥ 3). In the first group, the orthogonality of both sets of singular vectors is maintained to the relative accuracy of 10^{-15}, whereas in the second group it drops to 10^{-14}. It is also interesting to note, that the relative loss of orthogonality is always larger for the *a posteriori* computed left singular vectors (except for `mode` $= 5$). But more numerical experiments must be done to better understand this behavior.

6 Conclusions

We have designed, implemented and tested (albeit only by simulation on a serial computer) the so-called one-sided dynamic ordering. As far as we know, our ordering is completely new, based on the original ideas of applying the Lanczos algorithm (in fact, a set of Lanczos processes run in parallel) to the set of Jordan-Wielandt matrices and of using the squares of Frobenius norms of tridiagonal, symmetric Lanczos matrices as the weights in the maximum-weight perfect matching algorithm for finding 'worst' p pairs of block columns for their orthogonalization at the beginning of each parallel iteration step. First simulated numerical results show that it is possible to substantially decrease the number of parallel iteration steps needed for the convergence as compared with the usual cyclic ordering. In particular, there is no notion of the 'sweep' in the one-sided dynamic ordering. A lot of work must be done in testing the new ordering for matrices with various distributions of singular values. Also, the question of a reliable local stopping criterion remains open. However, the new one-sided dynamic ordering seems to be quite promising.

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] M. Bečka and G. Okša, *Variable blocking factor ... for a parallel block-Jacobi SVD algorithm*, Parallel Computing ...

[6] M. Bečka and M. Vajteršic, *Generalization of the Dynamic Ordering for the One-Sided Block Jacobi SVD Algorithm: I. Analysis and Design*, Technical report, Salzburg University, Salzburg, Austria, June 2008.

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

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

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

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

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

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

[13] 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.

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

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

[16] 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.

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

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

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

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

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

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

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

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

[25] 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.

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