Efficient parallel implementations of eigenproblems reduction.

Dissertation

Bergische Universität Wuppertal
Fakultät für Mathematik und Naturwissenschaften

eingereicht von Valeriy Manin, M. Sc.
zur Erlangung des Grades eines Doktors der Naturwissenschaften

Betreut durch Prof. Dr. Bruno Lang

Wuppertal, 24.02.2022

Contents

1 Introduction

This doctoral thesis is dedicated to extension of the `ELPA` library: a library designed for solving generalized eigenvalue problems (GEPs) $\mathsf{H}\mathsf{z} = \lambda\mathsf{S}\mathsf{z}$ and standard eigenvalue problems (SEPs) $\mathsf{A}\mathsf{x} = \lambda\mathsf{x}$ efficiently on parallel computers. The presented thesis is based on the two papers: [7] [8] and follows the text of these publications in many aspects.

The GEPs originate often from electronic structure computations [17, 30, 23] with a significant portion of the overall calculations contained in so-called self consistent field (SCF) cycles, which involve solving a sequence of generalized hermitian positive definite (HPD) eigenproblems.

A usual way to solve a generalized hermitian eigenproblem includes the following steps:

- reduction to a standard form
- solving the obtained SEP for eigenvalues and eigenvectors
- back transformation of the eigenvectors in order to get eigenvectors of the initial GEP.

To solve a SEP with a direct method, firstly a reduction of the problem to a tridiagonal form should be done. Sometimes, the initial GEP or SEP is not full, but has a banded form with a semi-bandwidth of order of hundreds or thousands. In this case a reduction to a narrower band with the subsequent tridiagonalization may be beneficial from the performance point of view.

This work is devoted to the two following aspects of the solution process:

- a reduction from the generalized form to a standard one and the following back transformation of the computed eigenvectors.
- A reduction of the banded SEP to a task with narrower band and restore of the eigenvectors of the initial problem.

An attempt to develop as efficient and scalable routines as possible was made for the two above-mentioned tasks. The main emphasis of the presented work was done on the implementation of methods and tricks to achieve the best performance on the modern supercomputer systems. In the first chapter the results for the generalized to standard eigenproblem reduction are presented and in the second chapter the bandwidth reduction is discussed.

2 Reduction of a generalized eigenproblem to a standard one

The object of investigation in this work is represented by a generalized eigenproblem of a kind $A^{(i)}X^{(i)} = BX^{(i)}\Lambda^{(i)}$, with $n \times n$ matrices $A^{(i)}$ and B being hermitian (the real symmetric case is similar). B in addition is positive definite. $\Lambda^{(i)}$ is a $k \times k$ diagonal matrix with the $k \leq n$ eigenvalues of interest on the diagonal, and $X^{(i)}$ is an $n \times k$ matrix of associated B-orthogonal eigenvectors. Although the presented in this work approach is efficient for a large variety of different GEPs, a one important specificity of the eigenproblems arising in the

electronic structure computations is taken into account. Namely, the "overlap matrix" B remains unchanged during all the iterations inside of a one SCF cycle, whereas the "Hamiltonian" $A^{(i)}$ depends on the solution $(X^{(i-1)}, \Lambda^{(i-1)})$ of the previous problem. Thus, hundreds of the solutions with the same matrix B but with different matrices A are required on each step of a simulation.

A reduction of a generalized eigenproblem to a standard one assumes calculation of representations of $\widetilde{A}$ and $\widetilde{X}$ such that: $AX = BX\Lambda \Rightarrow \widetilde{A}\widetilde{X} = \widetilde{X}\Lambda$.

The formulas for $\widetilde{A}$ and $\widetilde{X}$ can be obtained through the following transformations:

- start with $AX = BX\Lambda$; do Cholesky decomposition of $B = U^H U$ (U is upper triangular, and U^H denotes the conjugate transpose of U) $\Rightarrow$
- $AX = U^H U X\Lambda$: multiply the both sides by U^{-H} $\Rightarrow$
- $U^{-H}AX = UX\Lambda$: input and identity matrix in the form of $U^{-1}U$ between A and X on the left side $\Rightarrow$
- $U^{-H}AU^{-1}UX = UX\Lambda$: now denote $\widetilde{A} = U^{-H}AU^{-1}$ and $\widetilde{X} = UX$ and obtain the desired SEP.

Thus, the GEP to SEP reduction and the subsequent solving of the SEP and back transformation of the eigenvectors can be done as follows:

1. $B \Rightarrow U^H U$
2. Optional: $U \Rightarrow U^{-1}$ (explicit triangular matrix inversion)
3. $A \Rightarrow \widetilde{A} = U^{-H}AU^{-1}$
4. Solve the standard hermitian eigenproblem $\widetilde{A}\widetilde{X} = \widetilde{X}\Lambda$ for $\widetilde{X}$ and Λ
5. $X = U^{-1}\widetilde{X}$ (back transformation of the eigenvectors).

Step 3 can be implemented without explicitly building the upper triangular matrix U^{-1} (see [26] for an overview of possible realizations), and this is done in the function `TwoSidedTrsm` in the ELEMENTAL library [27] and `PDSYNGST` in ScaLAPACK [6]. Then a triangular solve must be done for the back transformation (step 5). This approach is particularly attractive if only a single eigenproblem with matrices A and B must be solved.

However, for a sequence of eigenproblems with the same matrix B, steps 1 and 2 must be performed only once. Then, with the explicitly built U^{-1} steps 3 and 5 can be represented as multiplications by a triangular matrix. An efficient implementation of such routines may be beneficial in comparison to the versions without explicit calculation of U^{-1}. This approach is followed in the ELPA library [25] and in this work.

This section presents implementations of scalable routines to realize steps 3 and 5 based on Cannon's algorithm for matrix multiplication. Cannon's idea was optimized for triangular matrices and an extension of Cannon's algorithm to the case of rectangular process grids was developed.

It will be shown, that such an approach may even be beneficial for solving a single eigenproblem, because the gain obtained from the optimized steps 3 and 5 may outweigh the time spent for the explicit computation of U^{-1}.

Note that the matrix U^{-1} in steps 3 and 5 is again upper triangular. To simplify the notation it is assumed that U has been overwritten with U^{-1} and simply a multiplication with U is considered, i.e., the reduction and back transformation take the form $\widetilde{A} = U^H A U$ and $X = U\widetilde{X}$, respectively.

The amount of operations in step 3 can be reduced by using the symmetry properties of the initial matrix A and of the output $\widetilde{A}$. Since A is hermitian, so is $\widetilde{A} = U^H A U$, and therefore it is sufficient to calculate only one triangle of $\widetilde{A}$. Step 3 may be implemented as a sequence of four substeps as follows:

(i) Compute the upper triangle M_u of $M = AU$ ("multiplication 1" in following); this takes approximately $\frac{2}{3}n^3$ arithmetic operations.

(ii) Transpose M_u to obtain the lower triangle M_l of $M^H = U^H A^H = U^H A$.

(iii) Compute the lower triangle of $\widetilde{A} = M_l U = U^H A U$ ("multiplication 2" in following; $\approx \frac{1}{3}n^3$ operations).

(iv) If the whole matrix $\widetilde{A}$ is needed for further computations, then reflect its lower triangle across the diagonal.

Transposition [11] is a cheap operation in comparison to the multiplications, and therefore an emphasis on efficient parallel routines for the triangular matrix multiplications is done: multiplication 1 for computing the upper triangular part of a product "hermitian full times upper triangular", and multiplication 2 to obtain the lower triangle of a product "lower triangular times upper triangular."

For the back transformation to restore eigenvectors of the initial problem (step 5) the function to compute a full result of the "upper triangular times rectangular full" multiplication was implemented.

2.1 Cannon's algorithm for full matrices

Firstly the general idea of the Cannon's algorithm for matrix multiplications should be presented. It is easier to explain the idea of the algorithm for square matrices on square process grids. And then an extension of this approach to the case of rectangular process grids will be demonstrated.

2.1.1 Cannon's algorithm for square process grids

Cannon's method [9] for a parallel matrix–matrix product $A \cdot B$ on a square grid of processes is summarized in Algorithm 1. As in most textbook descriptions (e.g., [20]) it is at first assumed, that the matrices are partitioned into size-$(n_b \times n_b)$ blocks on a $p_c \times p_c$ process grid, where $n_b = n/p_c$. Then each process $P_{i,j}$, where $0 \le i, j < p_c$, initially holds exactly one block of the matrices, namely $A_{\mathsf{loc}} \equiv A_{i,j} \equiv A(in_b+1 : (i+1)n_b, jn_b+1 : (j+1)n_b)$ and $B_{\mathsf{loc}} \equiv B_{i,j} \equiv B(in_b + 1 : (i+1)n_b, jn_b + 1 : (j+1)n_b)$.

The first two lines of the algorithm represent an initial "skewing" of the matrices, i.e., each process $P_{i,j}$ in the ith process row circularly sends its A_{loc} to the process i positions to its left, $P_{i,(j-i+p_c) \mod p_c}$, and receives a new A_{loc} from the process i positions to its right, $P_{i,(j+i) \mod p_c}$. Similarly, the jth block column of B is shifted by j positions upward in the process grid, such that $P_{i,j}$ now holds $A_{i,(j+i) \mod p_c}$ and $B_{(j+i) \mod p_c, j}$; see Figure 1. Further, on the k-th

Algorithm 1: Cannon's algorithm for a square $p_c \times p_c$ process grid

```
1 Circular shift for matrix A (by i positions to the left in process row i);
2 Circular shift for matrix B (by j positions upward in process column j);
3 C_loc := 0;
4 for k = 0 to p_c − 1 do
5 |   Local multiplication: C_loc := C_loc + A_loc · B_loc;
6 |   Circular shift for A: Send current A_loc to left neighbour and receive
  |     new A_loc from right neighbour;
7 |   Circular shift for B: Send current B_loc to upper neighbour and
  |     receive new B_loc from lower neighbour;
8 end
```

iteration of the loop, $P_{i,j}$ will hold $A_{i,(j+i+k) \mod p_c}$ and $B_{(j+i+k) \mod p_c,j}$, and therefore at the end of the algorithm this process will have computed the (i, j) block of the product $A \cdot B$

$$C_{\mathsf{loc}} = \sum_{k=0}^{p_c-1} A_{i,j+i+k} B_{j+i+k,j} = \sum_{m=0}^{p_c-1} A_{i,m} B_{m,j} = (A \cdot B)_{i,j}. \tag{1}$$

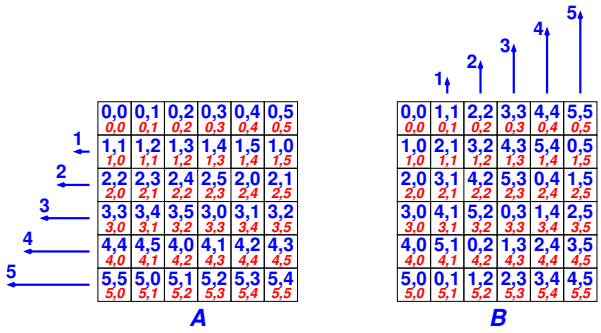

Figure 1: Distribution of the blocks of the matrices A and B after the initial skewing for the case that every process holds exactly one block of each matrix. For each block, the block number is shown with upright font, followed by the process coordinates in *slanted* font. The numbers next to the arrows indicate by how many positions the blocks in the respective block row (block column) have been shifted to the left (upward).

In fact Algorithm 1 is applicable in a more general setting when every process has more than one local block. Let A and B be distributed over a $p_c \times p_c$ process grid in a 2D block cyclic way into size-$(n_b \times n_b)$ blocks with arbitrary block size $n_b \geq 1$, i.e., $P_{i,j}$ holds exactly those blocks $A_{\ell,m}$ and $B_{\ell,m}$ such that $i \equiv \ell \mod p_c$ and $j \equiv m \mod p_c$ ("2D block cyclic" is the standard distribution for matrices in `ScaLAPACK` and `ELPA` libraries). Then, for the example with $p_c = 6$ shown in Figure 2, process $P_{2,4}$ gets A from $P_{2,(4+2)mod6} \equiv P_{2,0}$' and B from $P_{(2+4)mod6,4} \equiv P_{0,4}$ during the skewing. As a result of the initial skewing $P_{2,4}$ will hold $A_{\mathsf{loc}} = \begin{bmatrix} A_{2,0} & A_{2,6} \\ A_{8,0} & A_{8,6} \end{bmatrix}$ and $B_{\mathsf{loc}} = \begin{bmatrix} B_{0,4} & B_{0,10} \\ B_{6,4} & B_{6,10} \end{bmatrix}$. Note that the local

block rows/columns are not shifted individually (as might be assumed from Figure 2), but A_{loc} and B_{loc} are shifted as a whole. This keeps the local blocks sorted by increasing row/column number. It is not hard to verify that during the course of Algorithm 1, $P_{2,4}$ computes its portion $C_{\mathsf{loc}} = \begin{bmatrix} C_{2,4} & C_{2,10} \\ C_{8,4} & C_{8,10} \end{bmatrix}$ of the product, and similarly for the other processes.

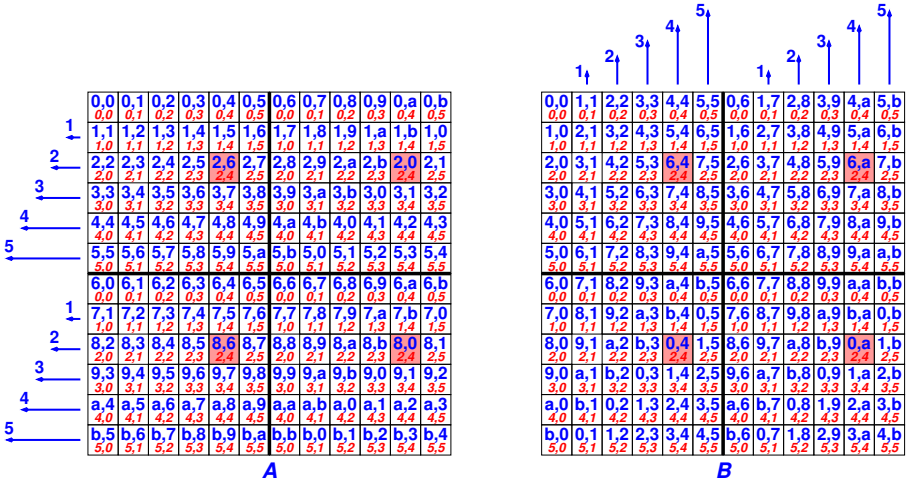

Figure 2: Initial skewing of matrices A and B for a 2D block cyclic distribution. The matrices A and B have 12 block rows and block columns, the process grid is of size 6-by-6 (shown by thick black lines). For each block, the block number is shown with upright font, followed by the process coordinates in *slanted* font. The block indices are hexadecimal, i.e., a $\equiv 10$ and b $\equiv 11$. The numbers next to the arrows indicate by how many positions the blocks in the respective block row (block column) have been shifted to the left (upward). The blocks ending up in $P_{2,4}$ are shaded.

2.1.2 Cannon's algorithm for rectangular process grids

In [24], Cannon's algorithm has been generalized to rectangular $p_r \times p_c$ process grids. Essentially, the process grid is emulating a virtual $p \times p$ grid, where p is the least common multiple of p_r and p_c. Thus, p steps are needed in the algorithm, each involving two shifts.

Consequently, the Cannon's algorithm will not be efficient on the grids such as "5×7". However, usually there is no need to use such grids in practice. In the following, the rectangular grids with the number of process columns being a multiple of the number of rows are considered, with an aspect ratio $r = p_c/p_r \in \mathbb{N}$.

In this situation the algorithm from [24] would take p_c steps. By contrast, the method described below requires only p_r steps and therefore reduces the number of communication operations by a factor of r (with the same overall communication volume), at the price of increased memory usage for keeping r copies of A.

Since the rows of a matrix are distributed over p_r process rows and columns of the matrix are distributed over $p_c = r \cdot p_r$ process columns, every process

locally stores r times more rows than columns. In order to make a use of all the locally available rows, the corresponding columns from r processes must be combined for every process. That implies the additional memory usage for r copies of A.

The A_{loc} of r processes will have to be combined during the skewing as it is shown on Figure 3 for a case of $p_r = 3$, $p_c = 6$, i.e., $r = 2$, and matrices with 12 block columns and rows. A 2D block cyclic distribution of the matrices is assumed. Since a block column of B_{loc} contains r times more blocks than a block row of A_{loc}, the $P_{i,j}$ will in addition to the A blocks from $P_{i,j+i}$ get the A blocks of $P_{i,j+i+p_r}$, $P_{i,j+i+2p_r}$, ..., $P_{i,j+i+(r-1)p_r}$, where all column indices must be taken modulo p_c. This in turn implies that the r processes $P_{i,j}$, $P_{i,j+p_r}$, ..., $P_{i,j+(r-1)p_r}$ (with "stride" p_r) will hold the same blocks of A, and these correspond to the locally available block columns of B.

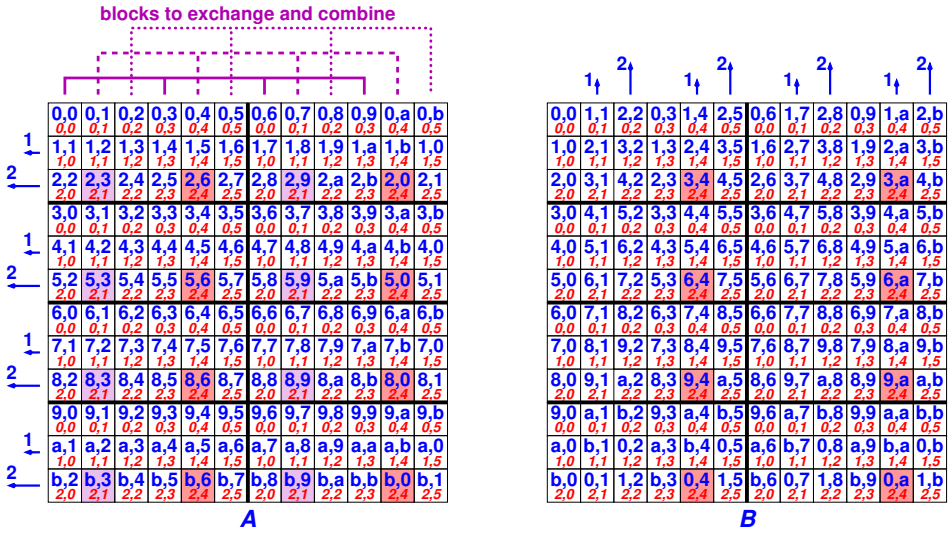

Figure 3: 2D block cyclic distribution of the matrices A and B on a rectangular process grid after the skewing. A and B have 12 block rows and block columns each, and the process grid is of size 3×6 (shown by thick black lines). Block and process numbers, as well as communication distance (next to the arrows), are denoted as in Figure 2. The blocks ending up in $P_{2,4}$ are shaded.

Since the A_{loc} and B_{loc} are exchanged as a whole (all the local blocks together), the ascending order of the block rows in each B_{loc} is preserved, and therefore the block columns of the combined A_{loc} must also be in ascending order, thus making it possible to implement the local multiplication in a one function call. This is achieved by interleaving the block columns of the A_{loc} from the processes $P_{i,i+j}$, $P_{i,i+j+p_r}$, ..., $P_{i,i+j+(r-1)p_r}$. Starting with the block column that has the lowest global number, the first block of the first process is copied at first, then the first block of the second process and so on till the first block of the $r-th$ process. Then the same for the second blocks of each process is done and so on. For the 3×6 grid shown in Figure 3, the skewing and com-

bination will lead to process $P_{2,4}$ holding $A_{\mathsf{loc}} = \begin{bmatrix} A_{2,0} & A_{2,3} & A_{2,6} & A_{2,9} \\ A_{5,0} & A_{5,3} & A_{5,6} & A_{5,9} \\ A_{8,0} & A_{8,3} & A_{8,6} & A_{8,9} \\ A_{11,0} & A_{11,3} & A_{11,6} & A_{11,9} \end{bmatrix}$ and $B_{\mathsf{loc}} = \begin{bmatrix} B_{0,4} & B_{0,10} \\ B_{3,4} & B_{3,10} \\ B_{6,4} & B_{6,10} \\ B_{9,4} & B_{9,10} \end{bmatrix}$.
$P_{2,1}$ will hold the same A_{loc}, but a different B_{loc}.

For the matrix B the initial skewing proceeds in a standard way as for the square process grids. If the rectangular grids with more process rows than columns were used, then blocks of B_{loc} from different processes would be assembled, whereas for the A_{loc} the skewing would be done in a standard way.

The resulting overall procedure is given in Algorithm 2. After skewing and combining A as just described (lines 1 and 2) and a "standard" skewing of B (line 3) the computations can be done as in Algorithm 1, in p_r steps. However, a more efficient scheme for the data transfers to left neighbours for A and to the top neighbours for B matrices can be implemented. Namely, the non-blocking send and receive operations should be used in order to overlap communications and computations. This requires two buffers for each local matrix: one pair of buffers is used for computation and sending ($A^{\mathsf{out}}_{\mathsf{loc}}$ and $B^{\mathsf{out}}_{\mathsf{loc}}$), and in the meantime the other couple of buffers can store the incoming data for the next step ($A^{\mathsf{in}}_{\mathsf{loc}}$ and $B^{\mathsf{in}}_{\mathsf{loc}}$). On the $i-th$ step of the algorithm a process firstly initiates sends of the $A^{\mathsf{out}}_{\mathsf{loc}}$ and $B^{\mathsf{out}}_{\mathsf{loc}}$ buffers to the neighbours and starts receives of data for the next $(i+1)-th$ iteration in the $A^{\mathsf{in}}_{\mathsf{loc}}$ and $B^{\mathsf{in}}_{\mathsf{loc}}$ buffers. Then the process does the local calculations with the current blocks $A^{\mathsf{out}}_{\mathsf{loc}}$ and $B^{\mathsf{out}}_{\mathsf{loc}}$. Hopefully, a significant part of the previously initiated receives of the buffers for the $(i+1)-th$ step will be completed before the local calculations are finished. Then a swap of pointers of the out and in buffers can be done, and the already received data can be utilized for the local computations on the $(i+1)-th$ step in parallel to receiving data for the next $(i+2)-th$ iteration in the receive buffers.

Using of such a scheme leads to optimized performance and scalability, however the benefits of such approach may differ and depend on the specific supercomputer configuration and interconnection scheme. This will be discussed in more details in Section 2.2.

Note that the shifts in the last step can be dropped if distribution after the initial skewing doesn't need to be restored. Then the whole last iteration $i = p_r - 1$ is replaced with $C_{\mathsf{loc}} := C_{\mathsf{loc}} + A^{\mathsf{out}}_{\mathsf{loc}} \cdot B^{\mathsf{out}}_{\mathsf{loc}}$.

In the presented approach, each process receives every p_rth block column of A, and therefore the amount of memory to store the local part of the matrix A depends only on the number of process rows. It doesn't make sense to increase the number of process columns in order to reduce memory consumption to store the matrix A: with the p_c increased every process will receive the same overall amount of blocks of A_{loc}, just from the increased number of sources. Thus, the amount of process rows must be increased making $r = p_c/p_r$ smaller if local memory size is an issue.

As it was already mentioned, a similar scheme can be derived for process grids where the number of rows is a multiple of the number of columns, $p_r/p_c \in \mathbb{N}$. Then B is replicated instead of A, and the B_{loc} are combined (and interleaved) among all "stride-p_c" processes within a column of the grid. In Sec-

Algorithm 2: Multiplication "full × full" on a $p_r \times p_c$ process grid with non-blocking communication; $(\mathsf{myRow}, \mathsf{myCol})$ are the coordinates of the current process

```
   /* Initial skewing; the communication for the shift of the
      A_loc and their combination (lines 1 and 2) can be done
      together */
```

1 Shift A_{loc} to the left by myRow positions;

2 Combine my A_{loc} with those of $P_{\mathsf{myRow},\,\mathsf{myCol}+p_r \bmod p_c}$, $\ldots$, $P_{\mathsf{myRow},\,\mathsf{myCol}+(r-1)p_r \bmod p_c}$ and arrange the block columns by increasing global number; this gives $A^{\mathsf{out}}_{\mathsf{loc}}$; `/* see main text for more details */`

3 Shift B_{loc} up by $\mathsf{myCol} \bmod p_r$ positions; this gives $B^{\mathsf{out}}_{\mathsf{loc}}$;

4 $C_{\mathsf{loc}} := 0$;

5 **for** $i = 0$ **to** $p_r - 1$ **do**

```
      /* Initiate shift for A */
```

6 `MPI_Isend(` $A^{\mathsf{out}}_{\mathsf{loc}}$ `)` to my left neighbour, $P_{\mathsf{myRow},\,\mathsf{myCol}-1 \bmod p_c}$;

7 `MPI_Irecv(` $A^{\mathsf{in}}_{\mathsf{loc}}$ `)` from my right neighbour, $P_{\mathsf{myRow},\,\mathsf{myCol}+1 \bmod p_c}$;

```
      /* Initiate shift for B */
```

8 `MPI_Isend(` $B^{\mathsf{out}}_{\mathsf{loc}}$ `)` to my upper neighbour, $P_{\mathsf{myRow}-1 \bmod p_r,\,\mathsf{myCol}}$;

9 `MPI_Irecv(` $B^{\mathsf{in}}_{\mathsf{loc}}$ `)` from my lower neighbour, $P_{\mathsf{myRow}+1 \bmod p_r,\,\mathsf{myCol}}$;

10 $C_{\mathsf{loc}} := C_{\mathsf{loc}} + A^{\mathsf{out}}_{\mathsf{loc}} \cdot B^{\mathsf{out}}_{\mathsf{loc}}$;

11 `MPI_Wait()` for A shift;

12 `MPI_Wait()` for B shift;

13 Swap pointers $A^{\mathsf{in}}_{\mathsf{loc}} \leftrightarrow A^{\mathsf{out}}_{\mathsf{loc}}$ and $B^{\mathsf{in}}_{\mathsf{loc}} \leftrightarrow B^{\mathsf{out}}_{\mathsf{loc}}$;

14 **end**

tion 2.2.4 the respective advantages of both variants related to the generalized to a standard eigenproblem reduction scheme will be presented.

Parallel matrix multiplication has been considered for decades, and several approaches have been proposed for 2D block cyclically distributed full matrices. They mainly differ in the way they bring together the corresponding blocks needed for the local computations and in the order of the products calculations in each process. Several approaches were realized in the `ScaLAPACK` library.

PUMMA (Parallel Universal Matrix Multiplication Algorithm) [12] extends an earlier algorithm [18] to this distribution. In each step, it shifts the matrix A along rows of the process grid, and in each column of the grid one process broadcasts its local portion of B, and in our situation ($p_r \times p_c$ grid, $p_c = r \cdot p_r$) the algorithm would take p_c steps. It would also be possible to shift B and broadcast A, leading to p_r steps. SUMMA (Scalable Universal Matrix Multiplication Algorithm) [19] uses an outer product approach, broadcasting block columns of A and block rows of B (along rows and columns, resp., of the process grid). These $2 \times \lceil n/n_b \rceil$ broadcasts can be implemented in a pipelined fashion such that they do not lead to a factor $\log p$ in the overall communication time. DIMMA (Distribution-Independent Matrix Multiplication Algorithm) [10] essentially rearranges these broadcasts and aggregates the computation to optimize the local `GEMM` performance and the overlapping of computation with communication.

Holding several copies of the matrices A and B may reduce the communication costs, as done in the so-called 3D and 2.5D algorithms [14, 22, 32]. The former arranges the p processes in a three-dimensional $p^{1/3} \times p^{1/3} \times p^{1/3}$ grid, generates $p^{1/3}$ copies of A and B by broadcasting along two axes from two-dimensional "slices," and then reduces the C blocks along the third dimension. The latter allows to adjust the memory requirements to $c \in \{1, \ldots, p^{1/3}\}$ copies of A and B; they essentially run c "truncated" instances of Cannon's algorithm (not the full number of steps) on these copies and reduce the partial results at the end.

An approach, which is presented in this work, does not involve collective communications, thus making the synchronization points less strict. It also allows overlapping of communications and computations in a natural way. All together, it should lead to a better scalability and efficiency on large process grids.

2.2 A Cannon-type algorithm for multiplication 1

This routine performs step (i) from Section 2, i.e. it computes the upper triangular part M_u of $M := AU$, where U is and upper triangular matrix representing the inverse of the Cholesky factor of the matrix B from the initial generalized eigenproblem $AX = BX\Lambda$, A is a full symmetric matrix, and only the upper triangular part of their product is to be computed.

The upper triangular structure of U and sufficiency to compute only the upper triangular part of the result make it possible to save on the amount of calculations and on the communication volume.

Essentially, the Algorithm 2 for Cannon's matrix multiplication is applied, with B_{loc} replaced by a buffer U_{buf} (see discussion below) and suitable modifications to lines 3 and 10, realizing the initial skewing of the right operand (U) and the local calculations respectively.

2.2.1 Initial skewing for multiplication 1

Since A is full, the initial skewing and combination for this matrix is done exactly as described in Section 2.1.2 and Algorithm 2 with no changes. For U, the benefits of its triangular structure are used to reduce the communication volume. Every process has to find its local nonzero blocks from the upper triangular part of the matrix and pack them contiguously into a buffer U_{buf}, which is then used for the skewing and the ensuing shifts. Indeed, two buffers for each of the matrices are used, as it was described in Section 2.1.2. Because of the upper triangular structure of U, different processes may have different amount of the nonzero blocks. Since every process circularly receives all the blocks of U from the processes of its column, each of the processes must allocate U_{buf} large enough to store the largest amount of the nonzero blocks among all the processes of its column.

For the upper triangular matrices U and M_u the number of nonzero blocks for the local block column j_{loc} for any process $P_{\mathsf{row,col}}$ in an arbitrary $p_r \times p_c$ grid (not necessarily p_c a multiple of p_r) can be found as follows:

$$\mathsf{numBlocks} \;=\; \beta^u(j_{\mathsf{loc}}, \mathsf{row}, \mathsf{col}) \;:=\; \left\lceil \frac{\mathsf{col} + j_{\mathsf{loc}} \cdot p_c - \mathsf{row} + 1}{p_r} \right\rceil, \qquad (2)$$

where $\lceil \cdot \rceil$ denotes rounding upward to the nearest integer. For the illustration of this rule the matrix M_u in Figure 4 can be observed, as M_u is also upper triangular and non-skewed.

Here and in the following, a process is considered as belonging to the upper part of the process grid, if its column index is not smaller than the row one (i.e., $\mathsf{myCol} \geq \mathsf{myRow}$), and as belonging to the lower part of the grid otherwise. It can be seen, that the processes of the upper part of the grid copy their local blocks from the columns starting from the first one ($j_{\mathsf{loc}} = 0$), whereas the processes from the lower part skip the first local block column, because there are no nonzero blocks there in the upper part of a matrix ($\beta^u(0, \cdot, \cdot) = 0$), and start with $j_{\mathsf{loc}} = 1$ having r nonzero blocks with $r = p_c/p_r$.

The packing is summarized in Algorithm 3.

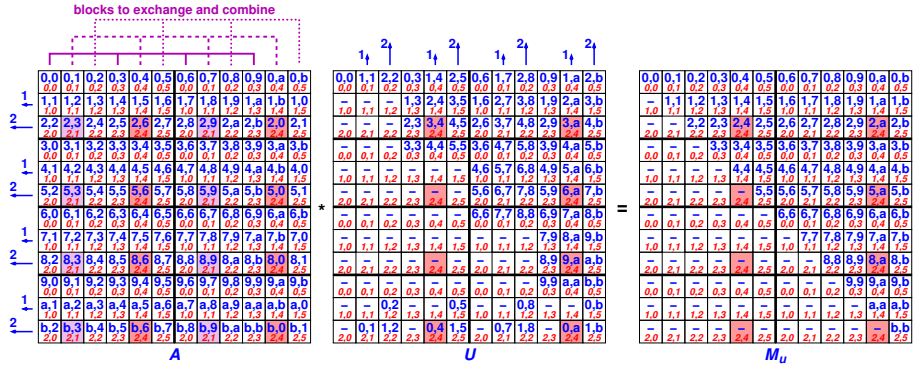

Figure 4: Initial skewing of the matrices A and U and distribution of the resulting matrix M_u (upper triangle of M) on a 3×6 process grid (shown by thick black lines). Block and process numbers, as well as communication distance (next to the arrows), are denoted as in Figure 2. The blocks ending up in $P_{2,4}$ are shaded. "–" marks a zero block, which is not touched.

Algorithm 3: Initial skewing of U for multiplication 1; $(\mathsf{myRow}, \mathsf{myCol})$ are the coordinates of the current process. This algorithm replaces line 3 in Algorithm 2.

```
   /* Loop to form U_buf block column by block column */
1  myBlockColsInU = ⌈ (⌈n/n_b⌉ − myCol) / p_c ⌉ ;  /* number of block columns in
     my U_loc */
2  U_buf = ∅;
3  for j_loc = 0 to myBlockColsInU − 1 do
4  |  numBlocks = β^u(j_loc, myRow, myCol) according to (2);
5  |  if ( numBlocks > 0 ) then
6  |  |  append block column j of U_loc to the buffer U_buf ;
   |  |   /* numBlocks · n_b^2 elements */
7  |  end
8  end
9  Shift U_buf up by (myCol mod p_r) positions; this gives U_buf^out;
```

The skewing also bears potential for asynchronous communication. One might just initiate the shift for A_{loc} and then overlap the actual communication with packing of local parts of U into U_{buf}. However, the benefits of such implementation are not significant. Communication time is an important issue only for large process grids, but the initial skewing is almost negligible on such grids, as compared to as many as p_r shifts between the updates, with same communication volume each.

2.2.2 Local update for multiplication 1

Benefits of the triangular structure of U and M_u in order to reduce amount of arithmetic operations can not be used, if the local updates are done in a single matrix product as in line 10 of Algorithm 2.

That is why it is needed to proceed by block columns, updating only the upper triangle of M_u and using the corresponding block rows of A.

The processes from the upper part of the grid update M_u starting from the first local block column, whereas the processes from the lower part compute M_u starting from the second local block column, since there are no nonzero blocks of M_u in their local data M_{loc}. In a similar way, the processes from the upper part of the grid initially (before skewing) have blocks of U needed for the calculation of the first local block columns of M_u, whereas the processes of the lower part of the grid have no useful blocks of U needed to make an update of the first local block columns M_u (it can be seen on the Figure 4 again).

That is why a process needs to determine the process row index of the initial source of U_{loc} for each iteration of the algorithm (for each local update) and compare it with its row index, taking into consideration the four following cases:

- if both me and the source are from the upper part of the grid, then I update M_{loc} starting from its first block column (since I am from the upper part of the grid and the received U_{loc} also contains the needed blocks of U for the first block column of M_u update, because the source is also from the upper part of the grid); use all the block columns of the received U_{loc}.

- if both me and the source are from the lower part of the grid, then update M_{loc} starting from its second block column (since I am from the lower part of the grid); use all the block columns of the received U_{loc} (there are no useless blocks there, because the source has skipped the first local block column while copying U_{loc} to a buffer).

- if I am from the upper part of the grid but the source is from the lower part, then update M_{loc} starting from its second block column (although I am from the upper part of the grid, but the source has no relevant blocks in U_{loc} to update the first local block column of M_u); use all the block columns of the received U_{loc}.

- if I am from the lower part of the grid but the source is from the upper part, then update M_{loc} starting from its second local block column (since I am from the lower part of the grid); skip the first block column of the received U_{loc} (the source is from the upper part, but I do not need to update the first local block column of M_u). However, I can not throw the first block column away from U_{loc} because it may be needed for the later iterations, when this part of U reaches a process from the upper part of the grid after some number of the shifts.

To illustrate this, the matrices from Figure 4 and two processes, $P_{0,1}$ (from the upper part of the grid) and $P_{2,0}$ (from the lower part) are considered in order to explain the four possible situations.

Since $P_{0,1}$ belongs to the "upper part" of the grid ($\mathsf{myRow} \leq \mathsf{myCol}$), according to (2) the first column in M_{loc} is nonempty in this process; see Figure 5.a). After the skewing, that is, in iteration $i = 0$ of Algorithm 2, $P_{0,1}$ holds the $A^{\mathsf{out}}_{\mathsf{loc}}$ (after combination; cf. Section 2.1.2) and U_{buf} that originally came from $P_{0,1}$ (itself) and $P_{1,1}$, respectively, and since $P_{1,1}$ is also in the upper part of the grid, U_{buf} contains a block $U_{1,1}$ from the first block column in $P_{1,1}$'s U_{loc}; see Figure 5.a1). Then the local update is performed in two steps,

$$M_{0,1} = M_{0,1} + A_{0,1} \cdot U_{1,1}$$

for the first block column in M_{loc}, and

$$\begin{bmatrix} M_{0,7} \\ M_{3,7} \\ M_{6,7} \end{bmatrix} = \begin{bmatrix} M_{0,7} \\ M_{3,7} \\ M_{6,7} \end{bmatrix} + \begin{bmatrix} A_{0,1} & A_{0,4} & A_{0,7} \\ A_{3,1} & A_{3,4} & A_{3,7} \\ A_{6,1} & A_{6,4} & A_{6,7} \end{bmatrix} \cdot \begin{bmatrix} U_{1,7} \\ U_{4,7} \\ U_{7,7} \end{bmatrix}$$

for the second block column. In the final iteration, $i = 2$, the situation is similar; see Figure 5.a3), whereas the U_{buf} for $i = 1$ comes from $P_{2,1}$, which is in the lower part of the grid, and thus U_{buf} does not contain a block from $P_{2,1}$'s first block column; see Figure 5.a2). Therefore the first block column of M_{loc} is not updated in this case.

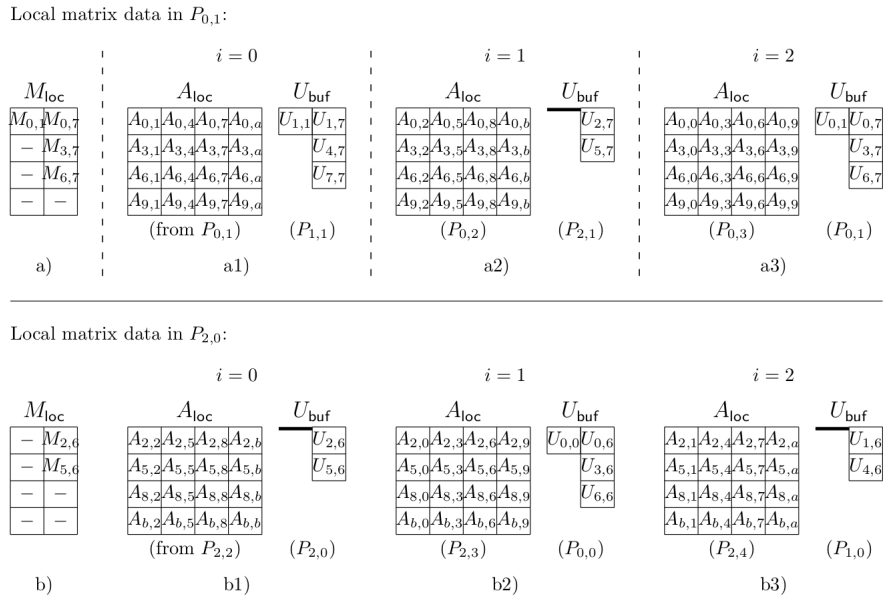

Figure 5: Local matrix data during Algorithm 2 after the initial skewing for two processes, $P_{0,1}$ in the "upper part" of the grid (i.e., $\mathsf{myRow} \leq \mathsf{myCol}$), and $P_{2,0}$ in the "lower part" ($\mathsf{myRow} > \mathsf{myCol}$). Pictures a) and b) show the local parts of the matrix M_u that are held and updated by these processes, and pictures a1)–a3) and b1)–b3) indicate the current contents of the buffers A_{loc} and U_{loc} (more precisely, the $A_{\mathsf{loc}}^{\mathsf{out}}$ and $U_{\mathsf{buf}}^{\mathsf{out}}$ used for the update) available in the three iterations $i = 0$ to 2. Below each of these buffers the process, from which it came originally, is indicated. Thick horizontal lines are used when no block from a block column of U had been packed into a U_{buf}. Matrix sizes and process grid are as in Figure 4. The block indices are hexadecimal, i.e., $a \equiv 10$ and $b \equiv 11$.

For $P_{2,0}$ the situation is slightly different. Since $P_{2,0}$ is in the lower part of the grid, the first block column in its M_{loc} is empty and therefore is never updated; see Figure 5.b). For $i = 0$ and $i = 2$ this matches the current U_{buf}s, which also come from processes in the lower part of the grid and therefore do not contain blocks from the first block column of U_{loc}; see Figure 5.b1) and b3). By contrast, the U_{buf} in iteration $i = 1$ comes from a process in the upper part, and therefore it contains a block $U_{0,0}$ from block column 0; see Figure 5.b2). This block is ignored in the local update, but it cannot be deleted from U_{buf}, because it might be needed in later local updates in other processes.

Algorithm 4 summarizes the local update.

2.2.3 Results for Multiplication 1

In this section the results obtained on the high performance computing systems of the two generations at the Max Planck Computing and Data Facility (MPCDF) in Garching are presented. The discussed approach showed better performance on the both configurations.

The older one was Hydra system with two 10-core Intel Ivy Bridge 2.4 GHz

Algorithm 4: One local update during multiplication 1; this algorithm replaces line 10 in Algorithm 2. (myRow, myCol) are the coordinates of the current process, and i is the loop counter from Algorithm 2.

```
  /* Loop over the block columns of M_loc for the local update
     */
1 myBlockColsInM = ⌈ (⌈n/n_b⌉ − myCol) / p_c ⌉ ; /* number of block columns in
   my M_loc */
2 for j_loc = 0 to myBlockColsInM − 1 do
3 |   blocksToUpdate = β^u(j_loc, myRow, myCol) ;     /* blocks in block
  |    column j_loc of M_loc */
4 |   origRowU = (myRow + myCol + i) mod p_r ;       /* row number of
  |    origin process of U_buf^out */
5 |   blocksInUBuf = β^u(j_loc, origRowU, myCol) ; /* blocks in U_buf^out for
  |    this block column */
6 |   if ( (blocksToUpdate > 0) and (blocksInUBuf > 0) ) then
7 |   |   M_act = M_act + A_act · U_act;
  |   |   /* The ''active'' submatrices are
  |   |       M_act ≡ M_loc(0 : blocksToUpdate − 1, j_loc),
  |   |       A_act ≡ A_loc^out(0 : blocksToUpdate − 1, 0 : blocksInUBuf − 1),
  |   |       U_act ≡ next blocksInUBuf blocks in U_buf^out,
  |   |       and indices refer to blocks, not to individual
  |   |       entries */
8 |   end
9 end
```

processors on each node. The nodes are interconnected with the fast InfiniBand FDR14 interconnect.

The new one was COBRA system. Each COBRA node contains two 20-core 2.4 GHz Intel Skylake processors, running a SUSE Linux Enterprise Server 12 SP3 operating system. As the usually used process numbers were represented by powers of 2, 32 cores of each of the involved nodes were utilized. MPI from the Intel Parallel Studio 2018.4 was applied, as well as the MKL from the same release for local `BLAS` and `ScaLAPACK` functionality. The nodes are interconnected with the fast OmniPath network.

The new algorithm was compared with two other functions, providing similar functionality,

- `PDTRMM` from `ScaLAPACK` for triangular matrix multiplication, and
- `elpa_mult_at_b_real_double` from the `ELPA` package, calculating the lower triangular part of an "upper triangular transposed × full" matrix product (with implicit transpositions).

All matrices were double precision real, the routines used only MPI parallelization.

For most of the runs the process grid was chosen to be "as square as possible", i.e., $p_r \times p_r$, if the process amount was a perfect square, and with two times more process columns than rows, $p_r \times 2p_r$, for the remaining values of p (all p were powers of 2). However, some runs with the grids of the other configurations

were made. Sometimes the grids of the form of $p_r \times 4p_r$ may provide better efficiency than the square ones for the new implementation. Also a comparison of the "tall" grids (with p_r larger than p_c) with the "wide" grids was interesting. This will be discussed later.

The optimal block size, n_b, for a given overall number of processes, p, and a matrix size, n, may depend on many parameters, e.g., performance of the node-local BLAS operations with respect to the shape and size of the local matrices involved in them, the speed of collective and point-to-point communication operations, cache size of the involved processors, etc.

For each p the fastest of three runs was reported. For some cases three block sizes, $n_b = 32, 64, 128$ were used and the best results among these runs (9 runs in total) were chosen.

Figure 6 presents the results for matrix sizes of 15000 and 30000 on Cobra machine.

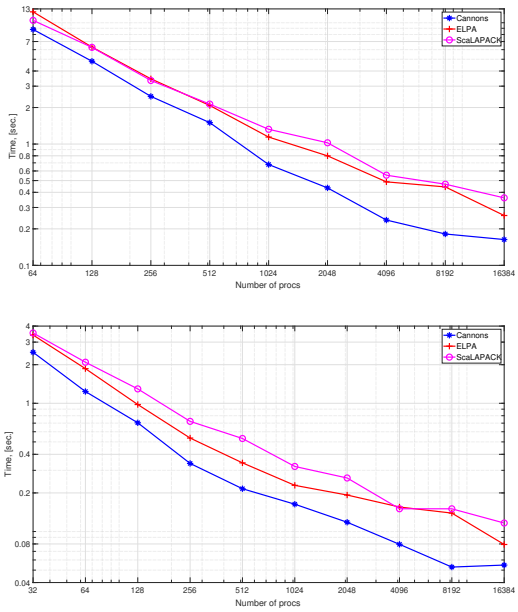

Figure 6: Timings for multiplication 1 on Cobra. Upper: n = 30,000, n_b is taken the best among 32, 64 and 128. Lower: n = 15,000, $n_b = 64$

Figure 7 presents the results for matrix size of 30000 on the previous Hydra computing system.

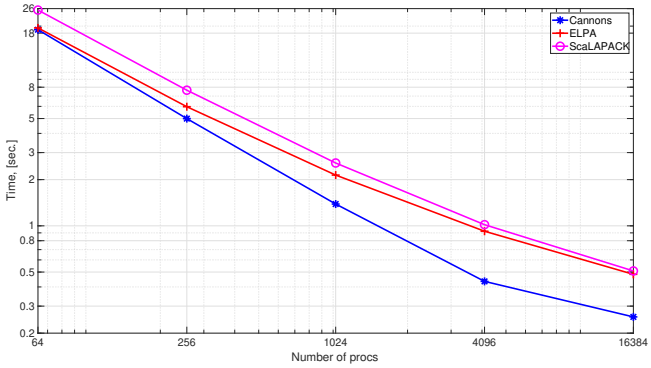

Figure 7: Timings for multiplication 1 on Hydra. $n = 30{,}000$, $n_b = 64$

It can be seen, that the Cannon's version scales significantly better than the other two implementations thanks to less strict synchronizations and asynchronous communications. It delivered up to 2 times better performance for the large process grids than the other two approaches.

2.2.4 Setting up the process grid

The speed of data transfers along process columns and rows of the grid typically depends on the process ordering while the grid setup.

Let an amount of processes per node be denoted as p_{node}. Usually values of $p_{\text{node}} = 32$ for Cobra and $p_{\text{node}} = 16$ for Hydra were used.

For the presented runs, a process distribution among nodes by contiguous chunks was utilized, thus mapping $P_{k \cdot p_{\text{node}}}, \ldots, P_{(k+1) \cdot p_{\text{node}} - 1}$ to the same node k. Since the largest fraction of data transfers in the new implementation is represented by communications of a process with its neighbour, that helps to reduce the inter nodes communication volume: a significant part of communications takes place in the shared memory of a node or between physically closer nodes. The last may be not always true if the system can not allocate the requested number of nodes consecutively. However, for the majority of the experiments the chunks of adjacent nodes were allocated to execute the programs.

A process grid was build using the routine `BLACS_GRIDINIT`: the $p = p^* \cdot p_{\text{node}}$ processes $P_0, \ldots P_{p-1}$ are configured as a $p_r \times p_c$ grid. With the presented assumption, a column-wise grid configuration, $P_{i,j} \equiv P_{i+j \cdot p_r}$, leads to faster communication along columns of the grid, since the processes in a column are mapped to physically closer nodes or to the same node. Similarly, a row-wise ordering of the processes, $P_{i,j} \equiv P_{i \cdot p_c + j}$, makes data transfers along rows to be faster.

This is not important, if the communication volume along the process rows and columns is the same. However, during "multiplication 1" blocks of a full matrix A are transferred along process rows, whereas only an upper triangular matrix U is moved within the columns. Thus, the communication volume along grid rows is significantly larger than along grid columns, and therefore it is reasonable to set up the grid with a row-wise ordering. In such a way, the larger amount of data will be transferred along the faster direction. The impact

may be negligible for small grids, however with the process number increase the right choice of the grid organization will have a significant influence on the performance. Table 1 and Figure 8 show, that the row-wise variant of the grid leads to 2.5 times better performance closer to the end of scaling.

Table 1: Timings (in seconds) for row-wise and column-wise ordering of the processes in multiplication 1 on Cobra (matrix size $n = 30{,}000$, $n_b = 64$).

Grid size	row-wise	column-wise
8×8	8.68	9.11
16×16	2.56	2.48
32×32	0.69	0.92
64×64	0.35	0.63
128×128	0.17	0.42

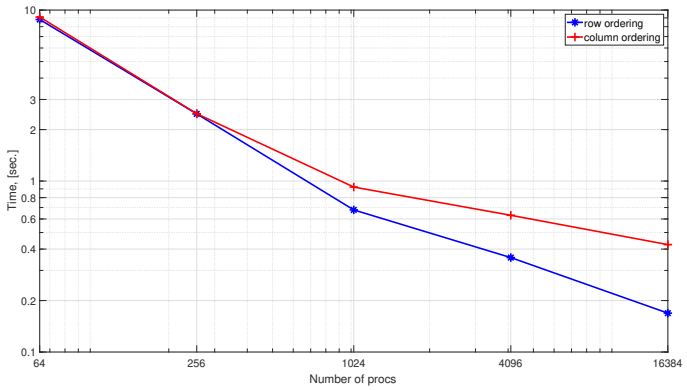

Figure 8: Timings for multiplication 1 with Cannon's algorithm on Cobra. $n = 30{,}000$, $n_b = 64$. Row and column ordering of processes.

If a user wants to double the number of processes to utilze, he has a choice whether the amount of process rows or columns to be increased. By increasing number of processes for one of the grids dimensions, the communication volume per process along this dimension will be proportionally decreased. For example, if a grid gets two time more process rows, then every process will have two times less local rows of a matrix with the unchanged amount of local columns. Thus, the communication volume per process along rows will be decreased in two times, whereas amount of data to be sent along process columns will remain the same.

With the process grid being organized in the row-wise manner, it is better to save on the column communications, because they are much slower. Consequently, it is more efficient to double the number of process columns. That is exactly an explanation why the "flat" grids with $p_r \leq p_c$ are preferred over "tall" ones.

Figure 9 shows the timings for the "flat" and "tall" rectangular grids with aspect ratios 2 in each case ($p_r \times 2p_r$ and $2p_c \times p_c$). For example, runs on the 16×32 and 32×16 grids are compared for the case of 512 processes. It can be seen, that the flat grids are superior in comparison to the tall ones for large process numbers. If communication within grid columns is faster, then tall grids may be superior.

The flat grids provide better performance despite the fact, that by using the tall grids the average communication volume per process would be reduced more intensely than in a case of the flat grids. As it was already mentioned, the flat and tall grids reduce communication volume per process along the process columns and rows respectively. The full matrix A is being shifted along rows, whereas the upper triangular matrix U is transferred along columns. Thus, halving the communication volume per process for A could be more desirable than halving data transfers for U in case of identical communication rates along process rows and columns. However, the benefits of the communication reduction along slow column dimension outweigh the potential advantages of the process rows increase.

This becomes even more obvious for multiplication 2, where the left operand is also a triangular matrix, and the communication volumes along rows and columns are identical.

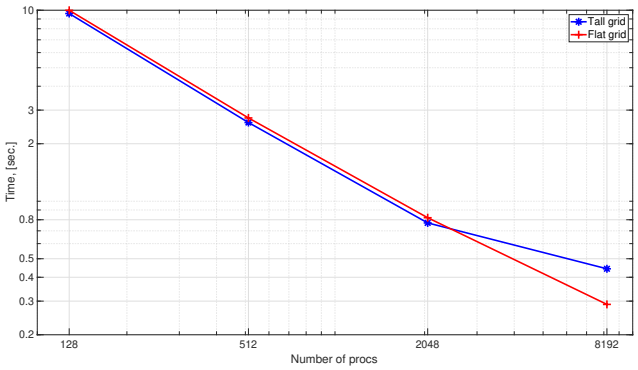

Figure 9: Timings for multiplication 1 with Cannon's algorithm on Hydra. $n = 30{,}000$, $n_b = 64$. Flat and tall grids.

2.3 Multiplication 2

This function performs step (iii) from Section 2, i.e., it calculates the lower triangular part of $\widetilde{A} = M_l U$ with M_l being lower triangular and U is the upper triangular inverse of B's Cholesky factor. For simplicity, just L instead of M_l will be written in the following.

2.3.1 Initial skewing for multiplication 2

Since the second factor U in multiplication 2 is the same as in multiplication 1, its skewing is done with Algorithm 3, first packing the local blocks tightly into a buffer U_{buf} and then shifting within the process column, yielding the initial $U_{\mathsf{buf}}^{\mathsf{out}}$

for the local updates. (The fact that U is the same in the both multiplications can be used to partly or completely avoid communications for U during the second multiplication. This idea will be presented later in Section 2.4.)

Now the other factor is also (lower) triangular, and the communication volume for the shifts can be reduced for the left operand also. Similarly to the matrix U, each local portion L_{loc} of L is also packed into a buffer L_{buf} before the skewing along process rows; see lines 1 through 8 in Algorithm 5. As it was shown in (2), for a grid with integer aspect ratio $r = p_c/p_r$ the number of blocks in the j_{loc}th column of $P_{\mathsf{row},\mathsf{col}}$'s L_{loc} is given by:

$$\beta^l(j_{\mathsf{loc}}, \mathsf{row}, \mathsf{col}) \;=\; \left\lceil \frac{\left\lceil \frac{n}{n_b} \right\rceil - \mathsf{row}}{p_r} \right\rceil - \left\lceil \frac{\mathsf{col} - \mathsf{row}}{p_r} \right\rceil - j_{\mathsf{loc}} \cdot r, \tag{3}$$

where the first term represents the maximum length of an L_{loc} column among all processes of the current grid row, the second term takes a value from $\{0, \ldots, r-1\}$ and accounts for missing blocks due to the lower triangular structure, and the third term reflects the fact, that each local block column is r blocks shorter than the previous one.

As described in Section 2.1.2, for the flat rectangular grids every process has to combine blocks of L from r processes, and therefore also sends its L_{buf} to r processes. This is done in lines 9 through 28 of Algorithm 5, together with the interleaving of the received block columns into a buffer $L^{\mathsf{out}}_{\mathsf{buf}}$ (to preserve the global order of the received blocks). For the computation of $\mathsf{pos}^{\mathsf{out}}$ in lines 23 and 10 note that in the buffer $L^{\mathsf{out}}_{\mathsf{buf}}$, each column of L has one block less than the preceding one, and the longest among these columns is the first one ($j_{\mathsf{loc}} = 0$) received from that "skewing partner" $P_{\mathsf{myRow},\mathsf{myCol}+\mathsf{myRow}}, \ldots, P_{\mathsf{myRow},\mathsf{myCol}+\mathsf{myRow}+(r-1)p_r}$ which has the lowest column number; cf. also Figures 10 and 11. Thus the first term in line 23 is the overall length of the buffer (in blocks), and the second term gives the offset of the current block column from the end.

2.3.2 Local update for multiplication 2

The local update is similar to Algorithm 4 and works on $\widetilde{A}_{\mathsf{loc}}$ block column by block column, with one important difference. The left factor L_{act} in each update $\widetilde{A}_{\mathsf{act}} = \widetilde{A}_{\mathsf{act}} + L_{\mathsf{act}} \cdot U_{\mathsf{act}}$ comes from a packed buffer with different leading dimension for each block column in L_{act}. That is why an update for a block column typically cannot be done in a single matrix multiplication, as in line 7 of Algorithm 4.

To give an example, consider the updates in iteration $i = 1$ in process $P_{1,1}$; see also Figures 10 and 11. While the first block column can be updated in a single multiplication,

$$\begin{bmatrix} \widetilde{A}_{1,1} \\ \widetilde{A}_{4,1} \\ \widetilde{A}_{7,1} \\ \widetilde{A}_{10,1} \end{bmatrix} = \begin{bmatrix} \widetilde{A}_{1,1} \\ \widetilde{A}_{4,1} \\ \widetilde{A}_{7,1} \\ \widetilde{A}_{10,1} \end{bmatrix} + \begin{bmatrix} L_{1,0} \\ L_{4,0} \\ L_{7,0} \\ L_{10,0} \end{bmatrix} \cdot \begin{bmatrix} U_{0,1} \end{bmatrix},$$

the update of the second block column (only the lower triangle of the result is

Algorithm 5: Initial skewing of L for multiplication 2; $(\mathsf{myRow}, \mathsf{myCol})$ are the coordinates of the current process, and $r = p_c/p_r$ is the grid's aspect ratio.

/* Pack my own L_{loc} into the buffer L_{buf} block column by block column */

1 $\mathsf{myBlockColsInL} = \left\lceil \frac{\lceil n/n_b \rceil - \mathsf{myCol}}{p_c} \right\rceil$; /* number of block columns in my L_{loc} */

2 $L_{\mathsf{buf}} = \emptyset$;

3 **for** $j_{\mathsf{loc}} = 0$ **to** $\mathsf{myBlockColsInL} - 1$ **do**

4 $\quad \mathsf{numBlocks} = \beta^l(j_{\mathsf{loc}}, \mathsf{myRow}, \mathsf{myCol})$ according to (3);

5 $\quad$ **if** ($\mathsf{numBlocks} > 0$) **then**

6 $\quad\quad$ append block column j_{loc} of L_{loc} to the buffer L_{buf} ; /* $\mathsf{numBlocks} \cdot n_b^2$ elements */

7 $\quad$ **end**

8 **end**

/* Shift the L_{buf}s by myRow positions to the left and combine this skewing with interleaving data from r processes with ''stride'' p_r; collect all received data in a buffer $L_{\mathsf{buf}}^{\mathsf{out}}$ */

9 $L_{\mathsf{buf}}^{\mathsf{out}} = \emptyset$; /* will hold $\frac{\mathsf{maxBlocks} \cdot (\mathsf{maxBlocks}+1)}{2}$ blocks, each containing n_b^2 elements */

10 $\mathsf{maxBlocks} = \beta^l(0, \mathsf{myRow}, (\mathsf{myCol} + \mathsf{myRow}) \bmod p_r))$ according to (3) ; /* longest column in $L_{\mathsf{buf}}^{\mathsf{out}}$ */

11 **for** $i = 0$ **to** $r - 1$ **do**

$\quad$ /* Communication partners are at a distance myRow (accounting for the shift) $+ i \cdot p_r$ (for combination) */

12 $\quad \mathsf{whereToSend} = (\mathsf{myCol} - \mathsf{myRow} - i \cdot p_r + p_c) \bmod p_c$;

13 $\quad \mathsf{fromWhereToReceive} = (\mathsf{myCol} + \mathsf{myRow} + i \cdot p_r) \bmod p_c$;

14 $\quad$ **if** ($\mathsf{whereToSend} \neq \mathsf{myCol}$) **then**

15 $\quad\quad$ Send L_{buf} to $P_{\mathsf{myRow},\mathsf{whereToSend}}$ and receive $L_{\mathsf{buf}}^{\mathsf{in}}$ from $P_{\mathsf{myRow},\mathsf{fromWhereToReceive}}$;

16 $\quad$ **else**

$\quad\quad$ /* Then also $\mathsf{fromWhereToReceive} = \mathsf{myCol}$, i.e., I use my own data */

17 $\quad\quad$ $L_{\mathsf{buf}}^{\mathsf{in}} = L_{\mathsf{buf}}$; /* L_{buf} contains my portion of L before shift; copy only pointer */

18 $\quad$ **end**

$\quad$ /* Append the contents of received $L_{\mathsf{buf}}^{\mathsf{in}}$ to $L_{\mathsf{buf}}^{\mathsf{out}}$; $\mathsf{pos}^{\mathsf{in}}$ points to the next (j_{loc}th) block column to extract from $L_{\mathsf{buf}}^{\mathsf{in}}$; $\mathsf{pos}^{\mathsf{in/out}}$ refer to $n_b \times n_b$ blocks, not individual elements */

19 $\quad j_{\mathsf{loc}} = 0$; $\mathsf{pos}^{\mathsf{in}} = 0$;

20 $\quad$ **while** ($\mathsf{pos}^{\mathsf{in}}$ has not reached the end of $L_{\mathsf{buf}}^{\mathsf{in}}$) **do**

21 $\quad\quad \mathsf{numBlocks} = \beta^l(j_{\mathsf{loc}}, \mathsf{myRow}, \mathsf{fromWhereToReceive} \bmod p_c)$ according to (3);

22 $\quad\quad$ **if** ($\mathsf{numBlocks} > 0$) **then**

23 $\quad\quad\quad \mathsf{pos}^{\mathsf{out}} = \frac{\mathsf{maxBlocks} \cdot (\mathsf{maxBlocks}+1)}{2} - \frac{\mathsf{numBlocks} \cdot (\mathsf{numBlocks}+1)}{2}$; /* see main text for details */

24 $\quad\quad\quad$ Copy $\mathsf{numBlocks}$ blocks from $L_{\mathsf{buf}}^{\mathsf{in}}$ (starting at block position $\mathsf{pos}^{\mathsf{in}}$) to $L_{\mathsf{buf}}^{\mathsf{out}}$ (starting at $\mathsf{pos}^{\mathsf{out}}$);

25 $\quad\quad\quad j_{\mathsf{loc}} = j_{\mathsf{loc}} + 1$; $\mathsf{pos}^{\mathsf{in}} = \mathsf{pos}^{\mathsf{in}} + \mathsf{numBlocks}$

26 $\quad\quad$ **end**

27 $\quad$ **end**

28 **end**

needed, so only the last two rows of L_{buf} are used),

$$\begin{bmatrix} \widetilde{A}_{7,7} \\ \widetilde{A}_{10,7} \end{bmatrix} = \begin{bmatrix} \widetilde{A}_{7,7} \\ \widetilde{A}_{10,7} \end{bmatrix} + \underbrace{\begin{bmatrix} L_{7,0} & L_{7,3} & L_{7,6} \\ L_{10,0} & L_{10,3} & L_{10,6} \end{bmatrix}}_{L_{\mathsf{act}}} \cdot \begin{bmatrix} U_{0,7} \\ U_{3,7} \\ U_{6,7} \end{bmatrix} =$$

$$= \begin{bmatrix} \widetilde{A}_{7,7} \\ \widetilde{A}_{10,7} \end{bmatrix} + \begin{bmatrix} L_{7,0} \\ L_{10,0} \end{bmatrix} \cdot \begin{bmatrix} U_{0,7} \end{bmatrix} + \begin{bmatrix} L_{7,3} \\ L_{10,3} \end{bmatrix} \cdot \begin{bmatrix} U_{3,7} \end{bmatrix} + \begin{bmatrix} L_{7,6} \\ L_{10,6} \end{bmatrix} \cdot \begin{bmatrix} U_{6,7} \end{bmatrix}$$

requires three multiplications, because even if all three block columns of L_{act} comprise the same number of blocks (two), the distance from $L_{7,0}$ to $L_{7,3}$ in $L_{\mathsf{buf}}^{\mathsf{out}}$ is three blocks, and from $L_{7,3}$ to $L_{7,6}$ it is only two blocks.

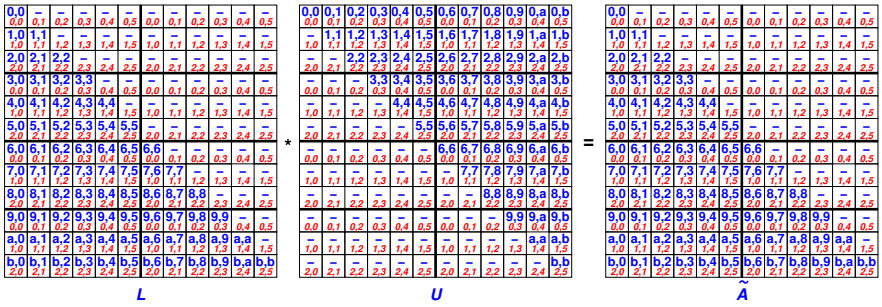

Figure 10: Distribution of the matrices L, U, and the lower triangle of $\widetilde{A}$ before the skewing for multiplication 2 on a 3×6 process grid (shown by thick black lines). Block and process numbers are denoted as in Figure 2. "–" marks a zero block, which is not touched.

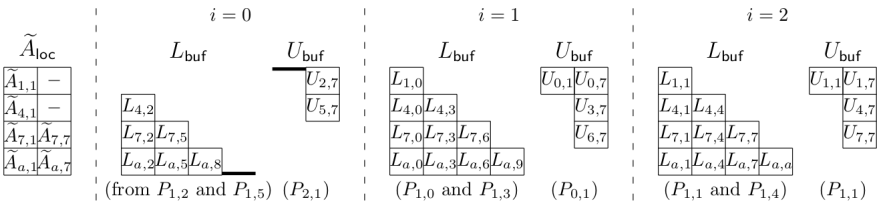

Figure 11: Local matrix data during Algorithm 2 after the initial skewing for process, $P_{1,1}$. The left picture shows the local part of the matrix $\widetilde{A}$ that is held and updated by this process, and the remaining pictures indicate the current contents of the buffers L_{buf} and U_{buf} (more precisely, the $L_{\mathsf{buf}}^{\mathsf{out}}$ and $U_{\mathsf{buf}}^{\mathsf{out}}$ used for the update) available in the three iterations $i = 0$ to 2. Below each of these buffers we indicate the process from which it came originally, and thick horizontal lines are used when no block from a block column of L or U had been packed into the buffer. The block indices are hexadecimal, i.e., $a \equiv 10$ and $b \equiv 11$.

In general, blocksInUBuf multiplications for each of the block columns are needed, where blocksInUBuf is the number of blocks in the j_{loc}th column of $U_{\mathsf{buf}}^{\mathsf{out}}$;

cf. line 5 of Algorithm 4. Therefore, line 7 of that algorithm must be replaced with a straight-forward loop taking $\mathsf{blocksInUBuf}$ iterations. There is also a possibility of $\mathsf{blocksInUBuf} = 0$: the first block column in M_{loc} is not updated during iteration $i = 0$ since there is no matching block in $U^{\mathsf{out}}_{\mathsf{buf}}$. In general, the value $\mathsf{blocksInUBuf}$ can be easily found, if the global row index of the U_{buf} origin process is known. The block columns of L_{act} are readily accessed once the quantity $\mathsf{maxBlocks}$, the length of the first (and longest) block column in the current $L^{\mathsf{out}}_{\mathsf{buf}}$, is known. This quantity must be determined w.r.t. the process that built this buffer before the skewing, that is,

$$\mathsf{maxBlocks} = \beta^l(0, \mathsf{myRow}, (\mathsf{myCol} + \mathsf{myRow} + i) \bmod p_r)$$

in iteration i.

Since only blocks in the lower triangle of $\widetilde{A}$ are being updated, all currently available U blocks will be used in the update. By contrast, in most cases not all blocks of $L^{\mathsf{out}}_{\mathsf{buf}}$ are used; again, these cannot be removed from the buffer because they may be needed in later updates in other processes.

2.3.3 Results for Multiplication 2

The results obtained on the COBRA machine are presented on Figure 12 and Figure 13.

There is no a specific function for a multiplication of two triangular matrices with computing of only one triangle in `ScaLAPACK`. That is why the `PDTRMM` function was used, as for the multiplication 1. Therefore a complexity of calculations is twice as larger for the `ScaLAPACK` version than for the `ELPA` and Cannon's variants. This two times difference can be observed in the timings for small process numbers.

The Cannon's implementation delivers the best performance for all the grid configurations. However, a drop of efficiency for the square grids of large sizes can be seen. The detailed timings show, that there is a slowdown because of high costs of data transfers along columns for the second matrix. That gives us an idea to try the rectangular grids instead of the square ones. Namely, for the 4,096 processes a 32×128 process grid was used instead of the 64×64 one, and for 16,384 processes the 64×256 grid instead of a 128×128 configuration was applied. The corresponding results are presented on the lower plots of Figure 12 and Figure 13.

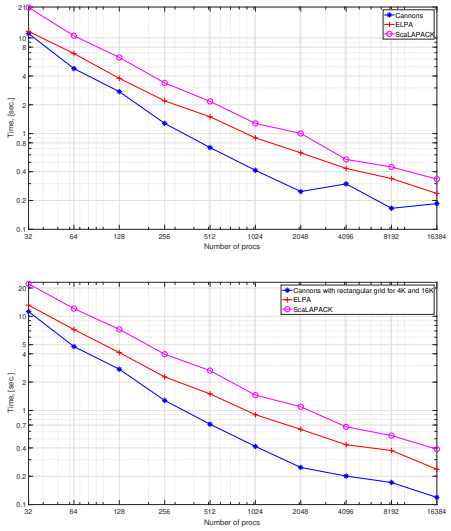

Figure 12: Timings for multiplication 2 on Cobra. $n = 30{,}000$. Upper: n_b is taken the best among 32, 64 and 128. Use square grids where it is possible. Lower: $n_b = 64$, use rectangular grids for 4096 and 16384 processes.

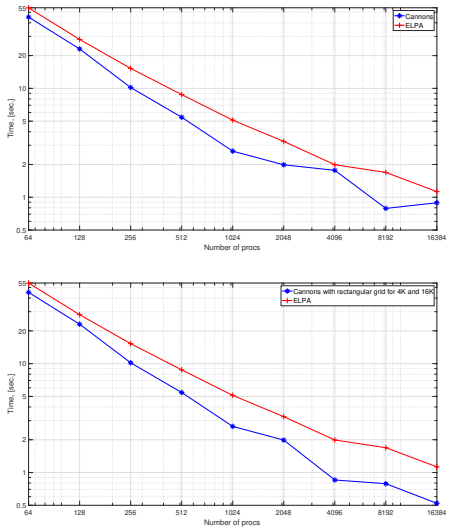

Figure 13: Timings for multiplication 2 on Cobra. $n = 60{,}000$, $n_b = 64$. Upper: use square grids where it is possible. Lower: use rectangular grids for 4096 and 16384 processes.

The results show a much better performance for the rectangular grids (Table 2 and Table 3 for the 30,000 and 60,000 matrices sizes respectively). For the other implementations a shift to the rectangular grids provides no advantage:

`ELPA` and `ScaLAPACK` benefit from the square grids mostly.

Table 2: Timings (in seconds) for rectangular and square grids for multiplication 2 (matrix size $n = 30{,}000$, block size $n_b = 64$).

Grid size	rectangular	square
4096	0.2	0.3
16384	0.12	0.2

Table 3: Timings (in seconds) for rectangular and square grids for multiplication 2 (matrix size $n = 60{,}000$, block size $n_b = 64$).

Grid size	rectangular	square
4096	0.85	1.77
16384	0.52	0.89

The benchmarks for the Hydra supercomputer system (Figure 14) for the case of 30,0000 matrix size and the block-size of 64 are also presented. The square grids were used where it was possible for runs on Hydra . It can be seen, that there is no such a significant performance drop for the large square grids on Hydra as it was on the Cobra machine. The reason for this may be that we are still not in a communication dominant regime for 4,096 processes in opposite to the situation observed of Cobra. That may happen due to the slower processors of the previous generation on Hydra alongside with the similar bandwidth of the interconnections among nodes. Obviously, a computer with the faster processors comes earlier to the end of scaling, if the speed of data transfers remains the same. A comparison of the timings for the different runs with the small processes numbers indicates, that the Cobra's processors are almost 2 times more powerful than the ones of Hydra.

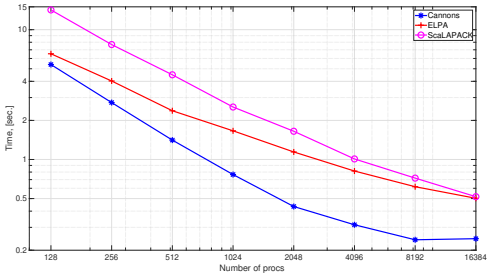

Figure 14: Timings for multiplication 2 on Hydra. $n = 30{,}000$, $n_b = 64$. Use square grids where it is possible.

2.4 Reduction in one function and back transformation

2.4.1 Combining both multiplications in one function

The two-sided triangular matrix multiplication $A \to U^H AU$ (with U being the inverse of B's Cholesky factor) includes the two already presented operations: multiplication 1 and multiplication 2. In the previous chapters their efficiency was analysed for the case, when the both operations were implemented as the two separate functions.

However, it is possible to benefit from implementation of the two-sided triangular matrix multiplication in a one function by significantly reducing the communication volume along process columns.

Note, that the right operand U is same in the both multiplications. That means, that exactly the same data transfers are carried out for U during the initial skewing and circular shifts in the both routines. In order not to duplicate these data exchange operations, the received blocks of U could be buffered during multiplication 1 in local storage and later used for multiplication 2.

This approach can shift the end of scaling further and is designed to enhance performance on large process grids with the communication time being dominant in comparison to the calculations.

Whether all the received blocks of U or just some part of them must be buffered, depends on the available memory size. The results for two versions of the function are presented:

(A) No buffering at all. Even in this case a small speedup may be possible thanks to an optimized memory allocation for buffers (only once for both multiplications) and thus perhaps more efficient cache usage.

(B) Full buffering. Here all U blocks, which were received during multiplication 1, are saved, and thus communications for U during the second multiplication are avoided completely.

Of course, intermediate levels of buffering are also possible. In the presented implementation it can be adjusted by a parameter, specifying how many of the circular shifts should be stored. Versions with buffering of the initially skewed matrices only and buffering of 50% of the shifts for matrix U were tried also during test runs. Keeping only the skewed matrices provides almost no benefits for large process grids, since the initial skewing represents relatively small part of the overall communication volume. For example, in a case of $p_r = 128$ there 128 shifts to be fulfilled with the same amount of data to be transferred as for the initial skewing. Thus, less than 1% of communications can be avoided in this case. The timings of the intermediate buffering runs were usually in between the extreme cases and have never shown the best performance in comparison to the two presented cases. That is why only the two extreme options are shown with an advice: if the buffering idea helps to improve the runtime of the routine, try to buffer as much shifts as it is allowed by the memory restriction.

In the Section 2.2.4 the process ordering to build a grid was discussed. A conclusion was made, that it was better to organize the process grid in a way making the data transfers along rows faster (thus making communications along columns slower). Since the communications for U are done along columns, the buffering idea helps to avoid exactly the slow communications, and only fast

data transfers along process rows are used for the second multiplication, what is a significant benefit of the idea.

2.4.2 Additional memory requirements for buffering

In this chapter an upper bound for the additional memory, required to store locally all the received blocks of U (full buffering), is estimated.

Firstly, the maximum number of blocks, that must be held in one process, should be determined. Note, that due to the upward shifts for U during multiplication 1, an involved process will get the non-zero blocks of U from all the processes of its process column. Thus, it will obtain the whole block-column of U.

Therefore the largest buffer must be allocated in the processes holding the longest block columns of the upper triangular matrix U, namely, the processes holding the last $(n^* = \lceil n/n_b \rceil)$th block column, which contains n^* blocks. Due to the block cyclic distribution over the $p_r \times p_c$ grid, these processes also must buffer every p_cth block column to the left, containing $n^* - p_c$, $n^* - 2p_c$, ..., blocks. In total, they hold $k^* = \lceil n^*/p_c \rceil$ nonempty block columns with a total of

$$\sum_{k=0}^{k^*-1} (n^* - k \cdot p_c) = k^* n^* - \frac{(k^*-1)k^*}{2} \cdot p_c \tag{4}$$

blocks, each containing n_b^2 elements. This is just slightly more than the $n(n+1)/(2p_c)$ elements that a perfectly balanced distribution of U's entries would take.

The size of additional memory depends on the p_c but not on the p_r. It is not possible to decrease the memory requirement by increasing the process rows amount, because every process has to store the whole block columns of U in the full buffering case anyway. Thus, in order to reduce the memory consumption the process columns number must be enlarged, what also corresponds to the idea of using the flat rectangular grids instead of the tall ones (see Section 2.2.4).

Table 4 presents the memory requirements in MB for the full buffering case. Since buffering is a method to avoid data transfers, it helps to increase performance in the communication dominant regimes mostly and is beneficial on relatively large process grids only. In this case there should be enough memory to run the algorithm. For example, for matrices of size $n = 30{,}000$ full buffering is noticeably faster starting from 4,096 processes, corresponding to a 64×64 grid. Then at most 61.3 MB (1 MB = 1024^2 bytes) per process are required to completely avoid communications along columns in the second multiplication, and thus with 32 processes per node less than 2 out of the 96 GB (1 MB = 1024^3 bytes) of a COBRA node's memory are consumed.

However, as it was already mentioned, it is also possible to use intermediate levels of buffering, storing only some of the shifts of U. As a rough estimate, buffering of a one shift requires amount of memory to store the local data of the matrix U, what corresponds to p_r "additional copies" of U for the full buffering case. Sharper bounds similar to (4) may be used to adjust the buffering level to the amount of available memory.

Table 4: Size of the required buffer for full buffering of U according to (4) in MB per process (1 MB = 1024^2 bytes) for different numbers of process columns, p_c, and matrix sizes, n, with double precision real data, i.e., 8 bytes per element, and block size $n_b = 64$.

p_c	$n = 30{,}000$	$n = 60{,}000$	$n = 100{,}000$
8	437	1733	4796
16	222	874	2410
32	115	444	1217
64	61.3	230	621
128	34.6	123	323

2.4.3 Back transformation of eigenvectors

A Cannon-type algorithm can be also applied in step 5 from Section 2, the back transformation $X = U^{-1}\widetilde{X}$ of the eigenvectors. Here, X is a matrix of the desired eigenvectors of the initial generalized eigenproblem $AX = BX\Lambda$, $\widetilde{X}$ are the computed eigenvectors of the standard eigenproblem $\widetilde{A}\widetilde{X} = \widetilde{X}\Lambda$, and U is the Cholesky factor of B.

As it was mentioned in Section 2, U^{-1} is calculated explicitly, and thus the back transformation represents just a matrix product instead of a triangular solve. Here again a multiplication of an upper triangular matrix by a full one must be computed, what makes the task similar to the multiplication 1 algorithm from Section 2.2. However, there are three differences. Firstly, for the back transformation we have the upper triangular multiplier to the left and the full matrix to the right in multiplication (it was vice versa for the multiplication 1 case). Secondly, the right operand is not necessarily a square matrix: if less than 100 % of the eigenvectors are needed to be computed, then X and $\widetilde{X}$ are tall rectangular matrices. And finally, the full matrix must be computed as a result, in contrast to a one triangle in multiplication 1.

The implementation is similar to multiplication 1 with the above mentioned specificities taken into account. Note, that for the efficiency reasons it is better to have a matrix with the smaller amount of elements to the right. As it was mentioned in Section 2.2.4, a process grid is organized in a way to make communications along the process rows faster than along the columns. Data transfers (initial skewing and circular shifts) are carried out along rows for the left matrix and along columns for the right operand. That is why it is beneficial to have a smaller matrix to right.

For multiplication 1 it is fulfilled, since the left operand is represented by a full square matrix and the right matrix is upper triangular. For the back transformation it is also true, if the rectangular matrix X (right operand) has less entries that the upper triangular factor U (left operand). Since there are $\frac{n\cdot(n+1)}{2}$ nonzero elements in U, the condition is fulfilled, if less than $\frac{n+1}{2}$ eigenvectors are needed to be restored (slightly more than 50% of the eigenvectors). As only 33% of them are usually needed in the target applications, the condition for the efficient communications will be commonly satisfied on practice.

2.4.4 Results for a reduction to a standard form and for back transformation.

The presented implementation of the generalized to standard eigenproblem reduction is compared with the `PDSYNGST` routine from `ScaLAPACK` and the multiplication functions from `ELPA`. The `PDSYNGST` routine is invoked with the `UPLO` parameter equal to L in order to utilize the fast $2k$ rank updates.

The new routine for the eigenvectors back transformation is compared with a triangular solve (routine `PDTRTRS`) from the `ScaLAPACK` library and with explicit multiplication functions from the `ELPA` library. For the `ScaLAPACK` version the triangular solve is applied, since no inverse of U is being computed explicitly by the `PDSYNGST` function.

The Figure 15 shows the results for $n = 15,\ 000$, and $n_b = 64$ on Cobra supercomputer with 33% of eigenvectors to be restored. It can be seen, that the Cannon's algorithm delivers the best performance for all the process numbers. The buffering and no-buffering versions of the reduction provide a similar efficiency for small amounts of processes (the buffering variant is slightly slower due to a necessity to write some data in a buffer). However, at the end of scaling the buffered version performs significantly faster (for 4,096 processes). The results for a back transformation also show the superiority of the Cannon's implementation thanks to much better scalability after 256 processes already.

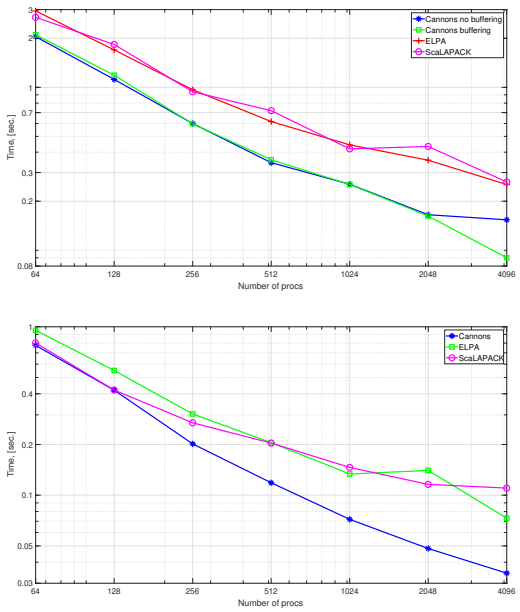

Figure 15: Timings in a case of $n = 15{,}000$, $n_b = 64$ on Cobra. Upper: timings for reduction to a standard form. Lower: timings for a back transformation, restore 33% of eigenvectors.

The corresponding results for $n = 30{,}000$, are shown on Figure 16. One can observe a significant advantage of the Cannon's variant. The buffered version

of Cannon's outperforms the no-buffered one starting from 4,096 processes and allows to utilize more processes efficiently. The block size is taken the best among 32, 64 and 128. However, the picture remains more or less the same for all the tested block sizes.

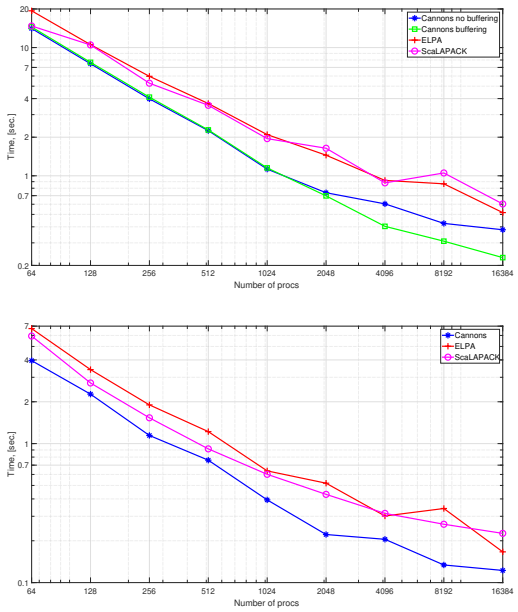

Figure 16: Timings in a case of $n = 30{,}000$ on Cobra. n_b is taken the best among 32, 64 and 128. Upper: timings for reduction to a standard form. Lower: timings for a back transformation, restore 33% of eigenvectors.

In Section 2.3.3 it was observed, that the multiplication results are better for rectangular process grids in case of a large amount of processes, since it helped to reduce the per-process communication volume in a slow direction, namely along the columns. That makes it reasonable to try the reduction functions on the rectangular grids for the cases of 4,096 and 16,384 processes. Since the buffered version of the algorithm efficiently avoids data transfers along columns, it is possible to expect an improved performance for the no-buffered variant only. Table 5 shows the results for the no-buffered Cannon's implementation with the matrix size $n = 30{,}000$ and block size $n_b = 64$.

Table 5: Timings (in seconds) for rectangular and square grids for reduction to a standard form with the no-buffered Cannon's algorithm (matrix size $N = 30{,}000$, $n_b = 64$).

Grid size	rectangular	square
4096	0.54	0.64
16384	0.36	0.38

Sometimes a back transformation of 100% eigenvectors is required. Remind, that the presented implementation of the Cannon's algorithm enjoys the efficient data transfers in the case, when the left matrix contains more elements than the right operand. That leads to a larger data transfer volume along process rows than along the columns, what is beneficial for the row-ordered grid configurations which are used.

However, with 100% of eigenvectors to be restored, the right factor is a full square matrix of size $n \times n$, whereas the left matrix is an upper triangular matrix having almost 2 times entries less. In this case a drop of performance can be expected for the large grids. That is exactly the picture that can be seen on the Figure 17, upper part: although the Cannon's implementation is the fastest one, it performs not enough efficiently closer to the end of scaling. It could be possible to use the column-ordered grids to increase performance in such a case, but that would lead to a sacrifice with the efficiency of the reduction to a standard form, which benefits from the row-ordered grids. That is why in order to reduce the along-columns communication costs, the rectangular grids instead of the square ones could be used for the cases of 4,096 and 16,384 processes. The Table 6 shows a superiority of the rectangular grids in this case: 10% and 13% faster for 4,096 and 16,384 processes respectively. The corresponding plot with the rectangular 4,096 and 16,384 grids is presented on Figure 17, lower part. The `ScaLAPACK` and `ELPA` implementations do not benefit from the rectangular grids because of symmetry of collective communications along process rows and columns.

Alternatively, one might implement an optimized routine for right-multiplying with a triangular matrix (similarly to the presented multiplication 1) and use that routine to compute $(\widetilde{X}^H U^{-H})^H$ in the case with more that 50% of eigenvalues to be restored.

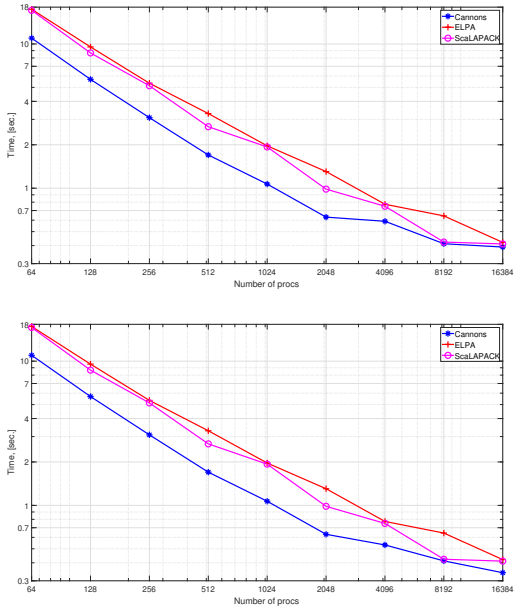

Figure 17: Back transformation of 100% of eigenvectors on Cobra, $n = 30{,}000$. Upper: n_b taken the best among 32, 64 and 128, use square grids where it is possible. Lower: use rectangular grids for 4096 and 16384 processes with $n_b = 64$.

Table 6: Timings (in seconds) for rectangular and square grids for back transformation of 100% of eigenvectors with the Cannon's algorithm (matrix size $N = 30{,}000$, block size $n_b = 64$).

Grid size	rectangular	square
4096	0.53	0.59
16384	0.34	0.39

The results for larger matrices ($n = 60{,}000$) are shown on Figure 18. Again, the Cannon's implementations performs better than the other ones, both for the eigenproblem reduction and eigenvectors back transformation. It should be also mentioned, that the back transformation even of the 33% of eigenvectors is slightly faster on the rectangular grids closer to the end of scaling.

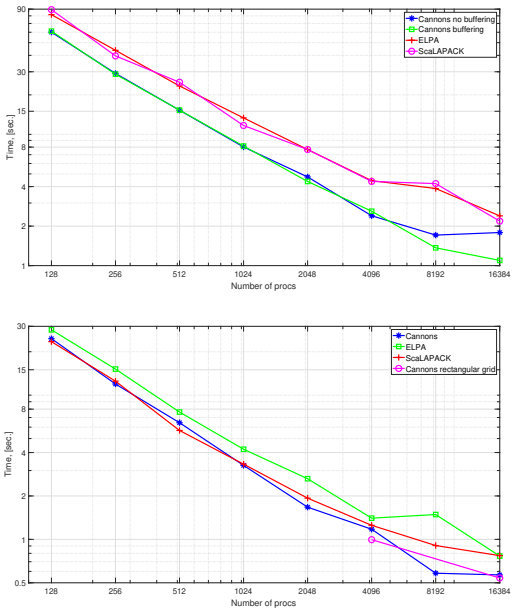

Figure 18: For a case of $n = 60{,}000$, $n_b = 64$ on Cobra. Upper: reduction to a standard form. Lower: a back transformation of 33% of eigenvectors.

Timings for a complex case are also presented. Since every complex multiplication requires 4 times more operations, but the communication volume is increased in 2 times only, a scalability in this case should be better for all the implementations. Figure 19 presents the corresponding results for $n = 30{,}000$ and block size of 64. The Cannon's version significantly outperforms all the other algorithms. The buffered variant of reduction is beneficial starting from 8,192 processes instead of 4,096 for the real case.

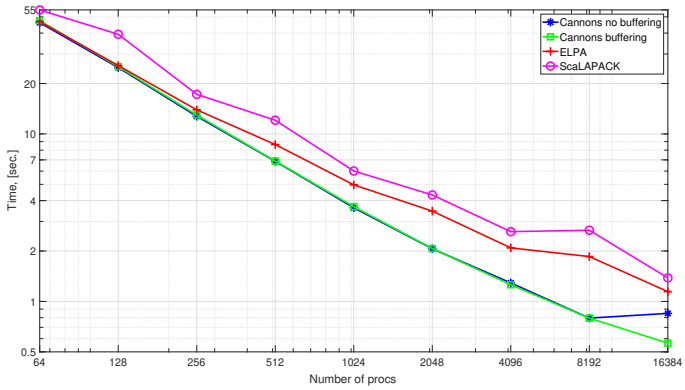

Figure 19: Timings for reduction to a standard form in a complex case on Cobra. $n = 30{,}000$, $n_b = 64$.

In addition to the Cobra machine, tests were done on the other supercomputers also. For example, for some of the runs the previous Hydra machine of the Garching computer center was used. Each Hydra node consists of 2 Intel Ivy Bridge (Xeon E5-2680v2) processors with 10 cores (2.8 GHz) each. The interconnection between nodes is a fast InfiniBand FDR14 network.

The Cannon's reduction routine showed the best performance (Figure 20). The fully buffered Cannon's version was almost 3 times faster than the `ELPA` routines and almost 6 times faster than the `ScaLAPACK` implementation at the end of scaling. The complex versions of the algorithms scaled predictably better than the real ones.

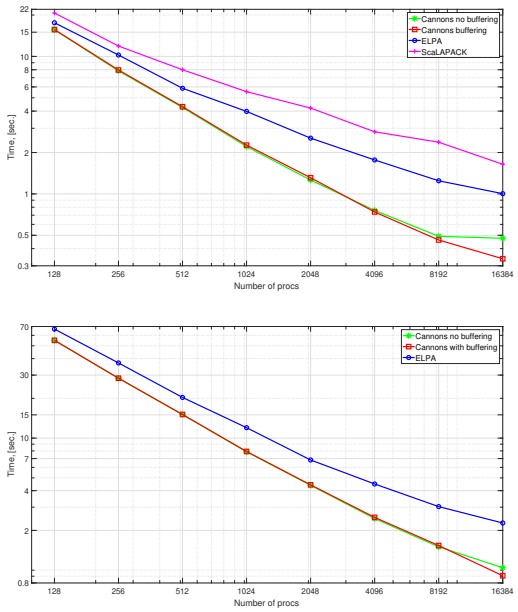

Figure 20: Timings for reduction to a standard form on Hydra. $n = 30{,}000$, $n_b = 64$. Upper: double real case. Lower: double complex case.

Again, the row-wise ordering of processes delivered a better efficiency than the column-wise one (Figure 21).

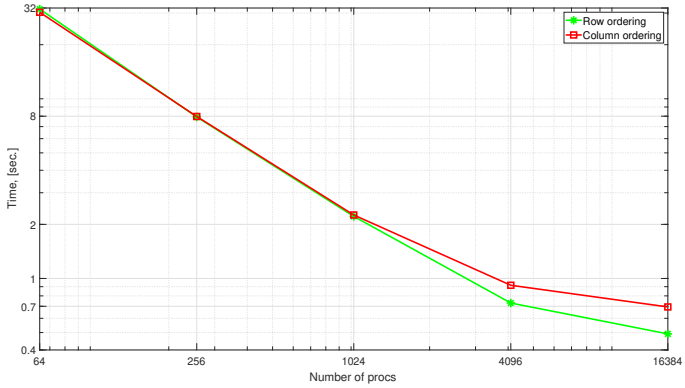

Figure 21: Hydra: reduction to a standard form times. $n = 30{,}000$, $n_b = 64$.

Remind, that the main application of the reduction routine is the SCF cycles solutions with the hundreds of the general eigenvalue problems to the standard ones transformations with the same matrix U. That is why the timings for the inverse of U can be neglected since only one inverse for hundreds of the reductions is done. However, the new approach may be competitive even for a case of a single reduction. Figure 22 presents the generalized to standard reduction plus inverse of U timings. For Cannon's and `ELPA` these two steps are done separately, whereas the `ScaLAPACK` approach does everything inside of a one routine.

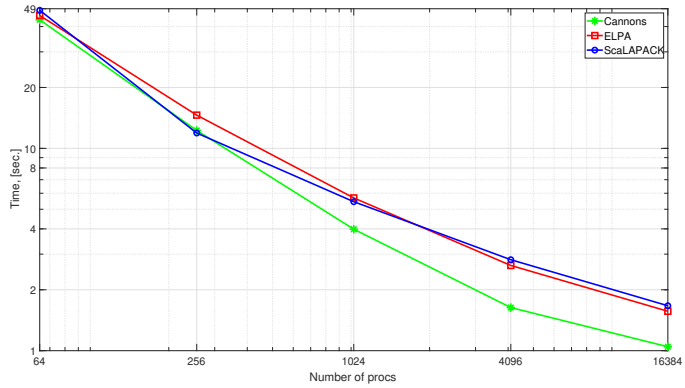

Figure 22: Timings for reduction to a standard form plus inverse of U on Hydra. $n = 30{,}000$, $n_b = 64$.

The measurements were also done for matrix sizes of 30,000 and 60,000 on the K-computer of the Riken Advanced Institute for Computational Science (Figure 23 and 24). Each node of the K-computer consists of a single 2.0 GHz eight-core SPARC64 VIIIfx processor. The nodes are interconnected with the Fujitsu's proprietary torus fusion network. The idea of using the K-comp was

to utilize more processes for the large matrix size in order to reach the end of scaling.

The new implementation delivered the best performance for all the runs on K-computer also.

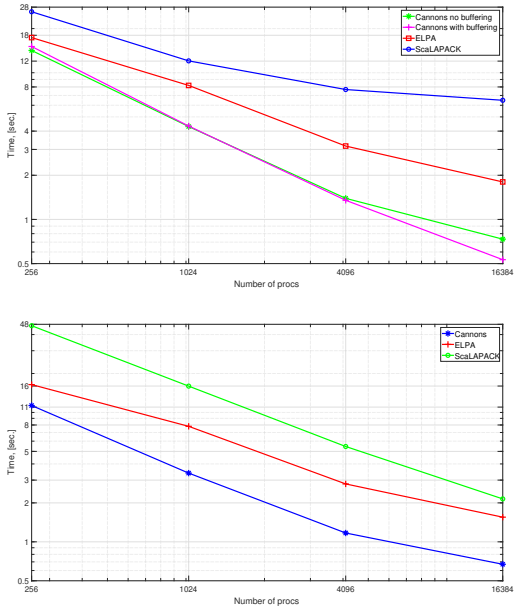

Figure 23: K-comp: timings for n = 30,000, n_b = 64. Upper: reduction to a standard form. Lower: back transformation of 100% of eigenvectors.

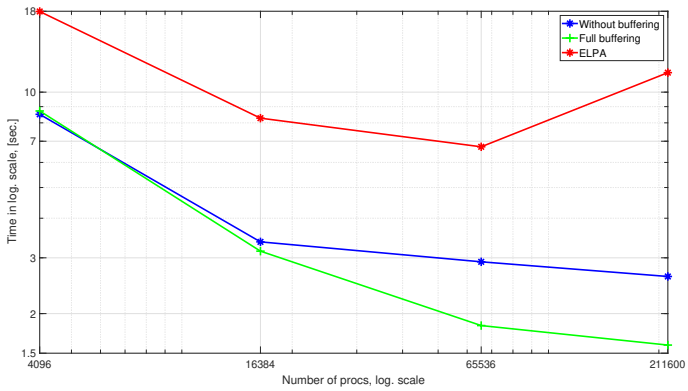

Figure 24: K-comp: timings for reduction to a standard form. n = 60,000, n_b = 64.

2.4.5 Quality of the computed eigensystems

To assess the quality of the computed eigensystems, generalized eigenproblems $AX = BX\Lambda$ with different sizes and condition numbers have been solved. The entries of the test matrices were chosen as follows: $a_{i,j} = \cos i \cos j + \sin j \sin i$, and $B = B^0 + \sigma I$, where $b^0_{i,j} = \sin j \sin i$ and $\sigma > 0$. Thus, cond(B) can be controlled by varying σ. Table 2.4.5 reports some of the results indicating that the two methods with explicit inversion (`ELPA` and the new implementation) perform comparably to `ScaLAPACK` (using implicit inversion). Note, that the maximum computed eigenvalue $\lambda_{\max}(A, B)$ is almost identical to cond(B) for these matrices.

n	$\lambda_{\max}(A,B)$	Residual $\max_j \|\|Ax_j - Bx_j\lambda_j\|\|_2$			B-orthonormality $\max_{i,j} \|x_i^T Bx_j - \delta_{i,j}\|$		
		ScaLAPACK	ELPA	new	ScaLAPACK	ELPA	new
1,000	5.01E+02	2.19E-12	3.28E-12	2.60E-12	1.35E-13	1.02E-14	1.24E-14
	5.01E+05	6.44E-08	7.05E-08	6.21E-08	3.56E-12	4.06E-12	3.87E-12
	5.01E+08	2.00E-03	3.60E-03	3.95E-03	3.39E-09	4.16E-09	3.27E-09
30,000	1.50E+04	8.15E-10	3.55E-10	4.07E-10	3.83E-11	2.78E-13	2.88E-13
	1.50E+07	1.25E-05	2.52E-05	2.17E-05	3.97E-11	7.39E-12	7.54E-12
	1.50E+10	3.57E-01	6.21E-01	1.35E+00	7.38E-09	7.54E-09	8.01E-09

Table 7: Maximum residuals and deviation from B-orthonormality for varying dimensions n and condition numbers cond$(B) \approx \lambda_{\max}(A, B)$.

3 Reducing the bandwidth

A symmetric standard eigenproblem

$$\mathsf{A}\,\mathsf{X} = \mathsf{X}\,\Lambda$$

is considered in this section with an additional assumption, that the matrix A is banded with semi-bandwidth b.

Such matrices naturally arise in simulations describing interactions, which vanish as the distance between partners grows. Usually a suitable reordering of the elements is required to obtain the banded form of a matrix to be solved ([13]).

The bandwidth of a matrix in such calculations usually depends on the dimensionality of a problem (e.g., 2D vs.or 3D), on the effective distance of interaction, the way to reorder the elements. On practice, the target values of the bandwidth are in range of hundreds or thousands.

An emphasis on direct eigensolvers is done in this work to solve the banded symmetric eigenvalue problems. This approach is beneficial, if a significant portion of the eigenvalues and eigenvectors is required. For example, in electronic structure computations often from 10% to 30% of the eigenvectors are needed.

Direct eigensolvers [20] typically reduce the symmetric matrix to a tridiagonal form firstly by application of a unitary/orthogonal transformation: $\mathsf{T} = \mathsf{Q}_{\mathsf{AT}}^{H}\,\mathsf{A}\,\mathsf{Q}_{\mathsf{AT}}$. After, the eigenproblem for the tridiagonal matrix is solved $\mathsf{T}\,\mathsf{X}_{\mathsf{T}} = \mathsf{X}_{\mathsf{T}}\,\Lambda$. Finally, the eigenvectors of the tridiagonal problem must be transformed to the ones of the initial problem: $\mathsf{X}_{\mathsf{A}} = \mathsf{Q}_{\mathsf{AT}}\,\mathsf{X}_{\mathsf{T}}$. A very narrow-banded (e.g., pentadiagonal) intermediate matrix might be used instead of a tridiagonal one [21].

The `LAPACK` [1], `ScaLAPACK` [6], and `ELPA` [25] libraries and the `SBR` toolbox [4, 3] (among others) contain routines for performing these steps. `LAPACK` and `SBR` represent the serial implementations of the algorithms (however, a shared memory parallelism via multi-threaded BLAS can be utilized), `ScaLAPACK` and `ELPA` contain the parallel functions to exploit distributed memory parallelism of modern supercomputers.

`ELPA` originally focused on full standard and generalized eigenvalue problems. With the ELPA-AEO project, support for banded problems has been, and continues to be, added. The efficient parallel algorithms were implemented to reduce banded generalized eigenproblems to equivalent standard ones with keeping the banded structure (see [28, 29] for a description). Also a reduction of a banded matrix to narrower-banded form was realised. This is the topic of the present chapter: efficient parallel reduction of a banded matrix to the one with the smaller bandwidth with eigenvalues being preserved. Also a fast routine to restore eigenvectors of the initial banded matrix was implemented.

`ELPA` already provides an efficient routine for tridiagonalizing banded matrices. However, it is optimized for small bandwidths only. A banded to tridiagonal reduction algorithm allows only application of the level-2 BLAS operations [15, 16]. Thus, it cannot fully exploit the computing capabilities of today's supercomputers. The degree of parallelism is $\mathcal{O}(n/b)$ for such a routine, what is also a problem.

Until recently it was not a significant drawback, since banded matrices in `ELPA` applications were presented as the intermediate result in the two-step

reduction of full matrices [25], with a small bandwidth $b \sim 32$ to 64. This two-step procedure includes a full to the banded matrix reduction as the first step (with a small bandwidth) with a subsequent banded to tridiagonal reduction on the second step. Consequently, a vast majority of operations was fulfilled during the first stage, where the level-3 BLAS operations are applied. And only the negligibly small portion of calculations is done on the second step, where the performance is not high.

However, if the initial matrix has a banded form with a bandwidth of order of hundreds or thousands, an appropriate algorithm should be implemented to reduce the bandwidth to the values of $\sim$ 32 to 64, after that the routine for tridiagonalization can be applied. This reduction to the narrower band must exploit the level-3 BLAS operations and take the initial banded structure of a matrix into account.

3.1 Serial bandwidth reduction

At first the serial reduction algorithm must be presented [3], because it is essential for the following discussion on parallelization.

A serial algorithm for bandwidth reduction has been proposed with the `SBR` toolbox [3, 4]. `ELPA` also contains a prototype parallel implementation by T. Auckenthaler [2] featuring BLAS3 performance, but utilizing only one level of parallelism and without the option of transforming back eigenvectors. In this thesis techniques to improve the scalability and to utilize more processes during the reduction are demonstrated. Also an efficient parallel algorithm for transforming back the eigenvectors was developed.

As a demonstration example, the reduction scheme for a Hermitian banded matrix A of size $n = 40$ with (semi-)bandwidth $b_\mathsf{A} = 7$ to a narrower-banded matrix B with (semi-)bandwidth $b_\mathsf{B} = 3$ is presented. Since the both matrices are Hermitian, only their lower triangles are taken into account.

The reduction proceeds by "sweeps". During each sweep b_B columns are being reduced to bandwidth b_B. In fact, the number of columns to be reduced per sweep might be chosen more freely in an interval $1 \leq n_b \leq b_\mathsf{B}$. However, for a simplicity reasons, and because small n_b tend to diminish compute performance, $n_b = b_\mathsf{B}$ is used both, for the idea presentation and for performance measurements.

For the first sweep, the matrix is considered as a block tridiagonal matrix with diagonal blocks A_{ii} and subdiagonal blocks $\mathsf{A}_{i+1,i}$. All the blocks are of size $b_\mathsf{A} \times b_\mathsf{A}$, except of the $b_\mathsf{B} \times b_\mathsf{B}$ block A_{00}, the $b_\mathsf{A} \times b_\mathsf{B}$ block A_{10}, and the blocks $\mathsf{A}_{N,N-1}$ and A_{NN} at the end of the band; cf. Figure 25.

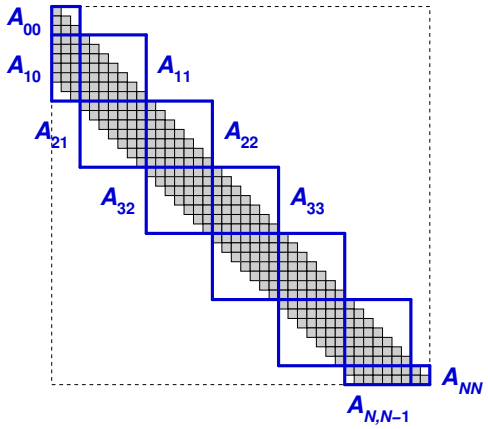

Figure 25: Block partition of the (lower part of) the band on the first sweep of the reduction. All blocks are $b_\mathsf{A} \times b_\mathsf{A}$, except for the first block column (width b_B) and block row N.

At first, a QR decomposition of the leading subdiagonal block is done: $\mathsf{A}_{10} = \mathsf{Q}_1^{(1)} \mathsf{R}_1^{(1)}$; cf. Figure 26(a). Then the orthogonal factor is applied to A from the both sides to preserve the eigenvalues of A. That means pre-multiplication of block row 1 with $\mathsf{Q}_1^{(1)^H}$ and post-multiplication of block column 1 with $\mathsf{Q}_1^{(1)}$ and leads to the structure shown in Figure 26(b). As a result, the subdiagonal block A_{21} fills almost completely with the nonzero elements. In order to avoid the fill-in spreading over the whole lower triangle of A, a critical part of the fill-in is removed with a "bulge-chasing" scheme before continuing with the bandwidth reduction.

More precisely, QR decomposition is applied to the first b_B columns of A_{21}, an orthogonal factor $\mathsf{Q}_2^{(1)}$ is obtained and is applied to block row and block column 2 of A, what restores the original bandwidth b_A in these b_B columns, but fills the next subdiagonal block A_{32}; cf. Figure 26(c). Then this procedure is repeated to remove the first b_B columns of the new fill-in, leading to the next subdiagonal block A_{43} to fill (cf. Figure 26(d)), and so on, until the end of the band is reached; cf. Figure 26(e).

As a result, the first b_B columns were reduced to the desired bandwidth value and the algorithm can proceed to the next columns by shifting the block decomposition by b_B rows and b_B columns (cf. Figure 26(f)). In a second sweep the next b_B columns of the matrix will be reduced to bandwidth b_B.

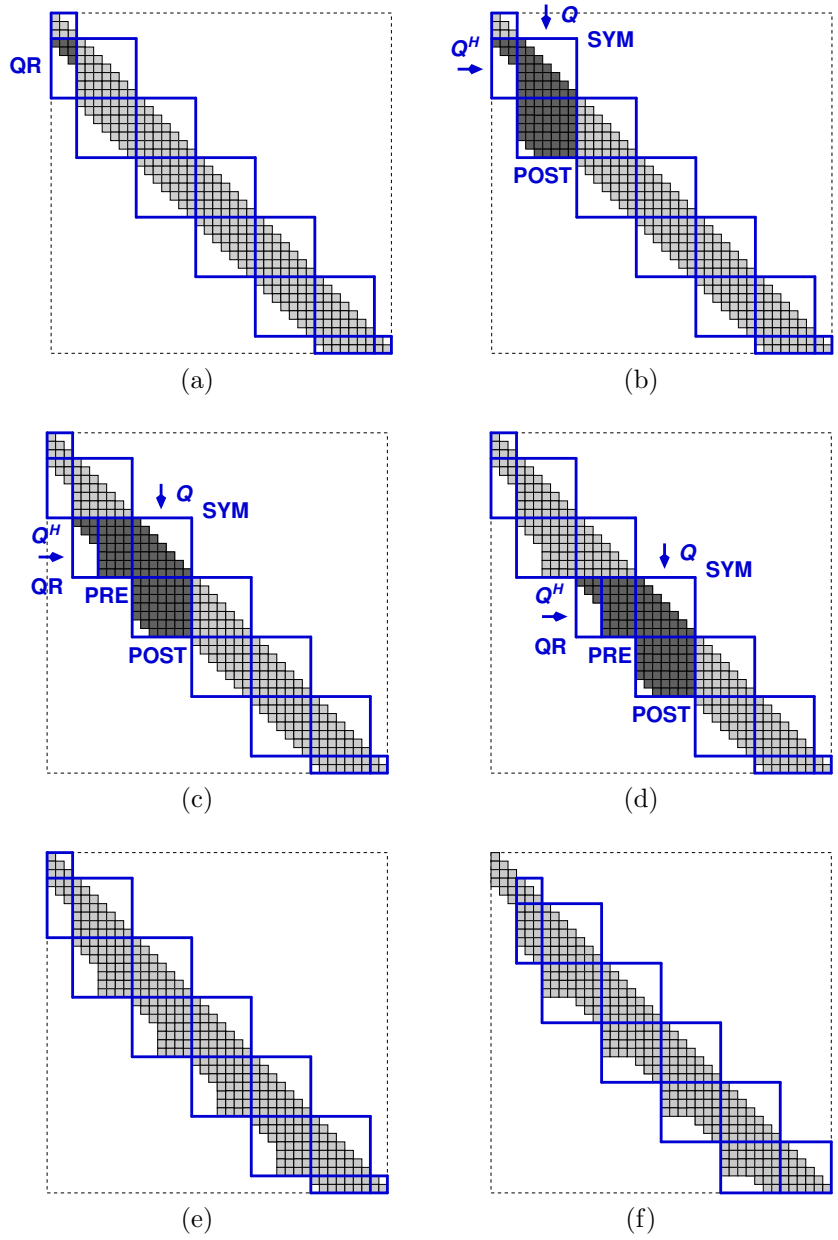

Figure 26: First sweep in the bandwidth reduction. (a) Initial QR decomposition of A_{10}. (b) Applying the resulting $\mathsf{Q}_1^{(1)}$ from both sides. (c), (d) Transformations 2 and 3 of the sweep, involving $\mathsf{Q}_2^{(1)}$ and $\mathsf{Q}_3^{(1)}$. (e) Structure of the band after the first sweep. (f) Shifting of the blocks for sweep 2. Dark grey indicates entries that are modified during the respective step, and light grey is used for the remaining nonzero entries.

Note that during the kth sweep a sequence $\mathsf{Q}_j^{(k)}$, $j = 1, 2, \ldots$, of orthogonal transformations is generated and applied from both sides. For each $\mathsf{Q}_j^{(k)}$ this

involves three or four steps:

QR QR decomposition of the first b_B columns of $\mathsf{A}_{j,j-1}$ (for $j = 1$ this is the whole block); this defines $\mathsf{Q}_j^{(k)}$.

PRE Pre-multiply the remaining $b_\mathsf{A} - b_\mathsf{B}$ columns of $\mathsf{A}_{j,j-1}$ with ${\mathsf{Q}_j^{(k)}}^H$ (for $j = 1$ this operation is void).

SYM Apply Q from both sides to the diagonal block: $\mathsf{A}_{jj} := {\mathsf{Q}_j^{(k)}}^H \mathsf{A}_{jj}\, \mathsf{Q}_j^{(k)}$.

POST Post-multiply the next subdiagonal block: $\mathsf{A}_{j+1,j} := \mathsf{A}_{j+1,j}\, \mathsf{Q}_j^{(k)}$.

3.2 Exploiting parallelism between blocks

Consider now the operations applied to a particular block column j during the kth sweep (cf. Figure 27): $\mathsf{A}_{jj} := \mathsf{Q}_{\text{old}}^H \mathsf{A}_{jj}\, \mathsf{Q}_{\text{old}}$ (SYM) and $\mathsf{A}_{j+1,j} := \mathsf{A}_{j+1,j}\, \mathsf{Q}_{\text{old}}$ (POST). The reflectors $\mathsf{Q}_{\text{old}} \equiv \mathsf{Q}_j^{(k)}$ for these operations had originated in the neighbouring block column to the left. Further, the next transformation $\mathsf{Q}_{\text{new}} \equiv \mathsf{Q}_{j+1}^{(k)}$ is determined from the first b_B columns of the subdiagonal block, $\mathsf{A}_{j+1,j}(:, 1 : b_\mathsf{B}) =: \mathsf{Q}_{\text{new}}\, \mathsf{R}_{j+1}^{(k)}$ (QR), and this new transformation is applied to the remainder of the block, $\mathsf{A}_{j+1,j}(:, b_\mathsf{B} + 1 : b_\mathsf{A}) := \mathsf{Q}_{\text{new}}^H\, \mathsf{A}_{j+1,j}(:, b_\mathsf{B} + 1 : b_\mathsf{A})$ (PRE). The relative order of POST, QR, and PRE is fixed: at first the whole $\mathsf{A}_{j+1,j}$ block must be updated with the POST operation, then the new reflectors Q_{new} must be generated based on the first b_B columns of the updated block, and finally the PRE operation must be applied to the last $b_\mathsf{A} - b_\mathsf{B}$ columns of the $\mathsf{A}_{j+1,j}$ block. However, the SYM might be done any time, even after PRE, since it operates with another block $\mathsf{A}_{j,j}$.

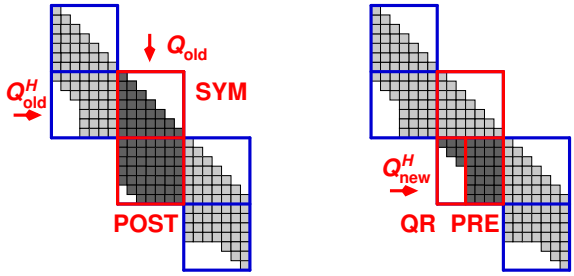

Figure 27: Operations applied to a particular block column j during the kth sweep (with $\mathsf{Q}_{\text{old}} \equiv \mathsf{Q}_j^{(k)}$ and $\mathsf{Q}_{\text{new}} \equiv \mathsf{Q}_{j+1}^{(k)}$).

To explain the parallel implementation of the algorithm, the first sweep of the algorithm is considered on a 1D process grid presented on Figure 28. Here again, $N = 40$, $b_\mathsf{A} = 7$ and $b_\mathsf{B} = 3$. Matrix is distributed over a grid having 4 processes from P_0 to P_3 using a block cyclic data distribution: with p processes, a process P_j holds block columns $j, j + p, j + 2p, \ldots$ of the band. In this case processes from P_0 to P_2 store 2 local block columns each, whereas process P_3 has only one. Of course, the zero values are not stored: only the nonzero $2 \cdot b_\mathsf{A}$ rows

are kept by each of the processes. Operations of the same colour are fulfilled with the same reflectors Q.

At first a naive and not the most efficient implementation of the algorithm is presented. In the following, some techniques to increase performance will be discussed. The first $3 = b_{\mathsf{B}}$ rows of the matrix already have the bandwidth of 3 and are not touched, thus the first sweep affects the rows from 4 to 40 with all the columns of the initial matrix. The sweep is started with the QR operation (red one) on process P_0. As a result, the rows from 4 to 6 of the matrix are also banded with the bandwidth of b_{B} and can be stored as an output. After this, process P_0 sends the reflectors to process P_1. Process P_1 receives the reflectors and does the SYM and POST updates (red ones). Then it generates the new reflectors with the QR operation (green one) and fulfills PRE operation (green one) by applying these new reflectors. Then process P_1 sends these reflectors to process P_2. Process P_2 receives the reflectors and starts with its updates. And so on till the end of the matrix.

When the end of the matrix is reached, then the first sweep is finished. On the second sweep the columns from 4 to 40 and rows from 7 to 40 will be touched (the whole picture is shifted by b_{B} rows and columns to the right-bottom). The second sweep starts with the QR update for block $\mathsf{A}(7:13, 4:6)$. Then the SYM operation for the block $\mathsf{A}(7:13, 7:13)$ must be done and so on similarly to the first sweep.

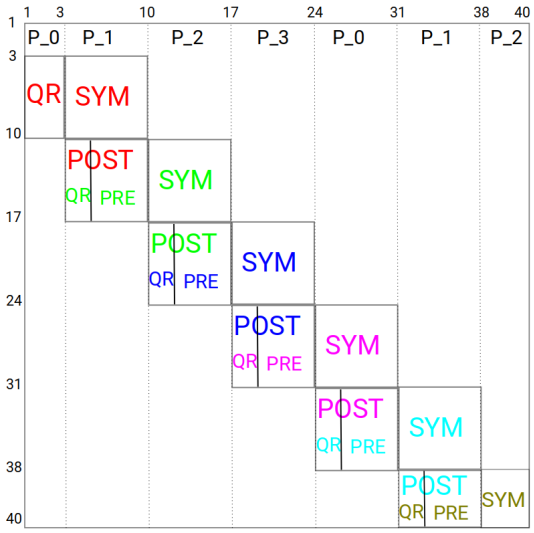

Figure 28: Data distribution for a one sweep.

However, SYM transformation for the $\mathsf{A}(7:13, 7:13)$ block cannot be applied as it is, because, as it was already mentioned, the matrix is not stored globally as it is shown on the Figure 28, but only the $2 \cdot b_{\mathsf{A}}$ rows of the matrix are kept. In addition, application of the transformations to the block, which is distributed over two different processes, would lead to unnecessary data transfers, thus reducing performance of the implementation. That is why data is firstly shifted by b_{B} rows and columns to left-up before the second sweep. That

means, that every process must send the first b_{B} columns of length $b_{\mathsf{A}} + b_{\mathsf{B}}$ for each of its local block columns to its left neighbour. After this, all the operations can be done by processes locally on the second sweep.

Thus, update of a block requires communication of a process with its left and right neighbours. If a j-th global block column must be updated on sweep k, then the process, which stores this block column locally, has to receive the following data before the update:

- reflectors Q from the left neighbour. These reflectors must originate from an update of global block column $j-1$ on sweep k. Process P_0 does not need to receive Q to update its first local block column (the 0-th global block column of a matrix)
- b_{B} columns from the right neighbour. These elements represent the first b_{B} columns after an update of the $j+1$-th global block column on the previous sweep $k-1$. This receive is not needed for the first sweep. Also, this receive is omitted, if the right neighbour doesn't need to update the $j+1$-th global block column on this sweep.

A process has to send the following data after the update of its local block column:

- reflectors Q to the right neighbour. This communication is required only if the right neighbour updates the $j+1$-th global block column on the current sweep k (because of the band shifts to the left, the processes have less local block columns to update as the sweep number grows)
- first b_{B} columns of length $b_{\mathsf{A}} + b_{\mathsf{B}}$ to the left neighbour, such that the left neighbour could make an update of the $j-1$-th global block column on the next sweep $k+1$. Of course, process P_0 doesn't need to fulfill this data transfer for its first local block column (there is nothing to the left) in the matrix.

A certain level of parallelism can be achieved by overlapping transformations from different sweeps in a pipelined manner. The next sweep can be initiated before the previous one is finished. In order to make an update on the $k+1$-th sweep for a block column, the process has to receive reflectors from the left neighbour and b_{B} columns from the right neighbour. The conditions for this are: the left neighbour has already fulfilled the QR operation on the $k+1$-th sweep for the corresponding block column and the right neighbour has updated the first b_{B} columns on the k-th sweep. Since every process may have many local bock columns to update, these already updated block columns must be adjacent to the current block column from left and right in the global matrix representation.

Order of the block columns updates could look as follows (see Figure 29, numbering is started from 0 for processes and from 1 for local block columnns):

1: Process P_0 makes QR for columns 1 to 3 (sweep 1 for the 1-st local block column) and sends reflectors to process P_1.

2: Process P_1 receives reflectors, updates columns from 4 to 10 (sweep 1 for the 1-st local block column), sends reflectors to process P_2 as well as the first b_{B} columns to process P_0.

3: Process P_2 receives reflectors, makes an update of columns from 11 to 17 (sweep 1 for the 1-st local block column), sends reflectors and first b_{B} columns of the updated block column to processes P_3 and P_1 respectively. At the same time, while process P_2 is working on its update, process P_0 receives columns from P_1 and makes QR operation (sweep 2 for the 1-st local block column).

4: now process P_3 can can join to the first sweep: it receives reflectors from P_2, makes update of columns from 18 to 24, sends reflectors to P_0 and the first b_{B} columns to P_2. At the same time, process P_1 can start with the update of its first local block column on the second sweep, because P_0 (left neighbour) has already done an update of its first local block column on sweep 2, and P_2 (right neighbour) has done an update of its first block column on sweep 1. So, P_1 makes an update of its first local block column for sweep 2 and sends the necessary data to processes P_0 and P_2.

5: since P_3 has done an update for its block column on sweep 1, process P_0 has reflectors to start an update of its second local block column (columns from 25 to 31). At this time, P_2 can make an update of its first local block column on sweep 2 (the left neighbouring process P_1 and right neighbouring process P_3 have done their updates on sweeps 2 and 1 respectively).

6: since P_0 has already done an update for its second local block column on sweep 1, process P_1 can do the same now: it updates its second local block column on sweep 1 (columns from 32 to 38). In parallel, process P_3 updates its local block column on sweep 2 (the needed updates on his neighbours have already been done: P_2 updated its first block column on sweep 2 and P_0 updated its second block column on sweep 1). And simultaneously, process P_0 can initiate sweep 3: make an update (just QR) for its first local block column on sweep 3. It can be done, because its right neighbour P_1 has already finished with updating of its first block column on sweep 2.

7: once P_1 finished with update of its second local block column on sweep 1, process P_2 can also update its second local block column. That will be the last update for sweep 1. At this time, P_0 can calculate for its second block column on sweep 2, because P_1 finished with update of its second local block column on sweep 1 and P_3 has updated its first block column on sweep 2. Simultaneously, process P_1 can start with updating of its first local block column on sweep 3.

8: and so on. Overall, the degree of this inter-block parallelism is $\mathcal{O}(n/b_{\mathsf{A}})$.

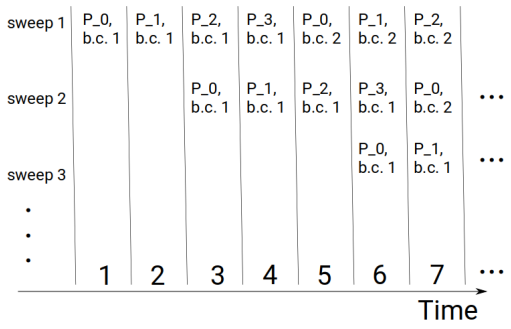

Figure 29: Timesteps for a pipelined parallel run of the bandwidth reduction. By b.c. the local block columns of the processes are denoted.

Note, that on timestep 5 process P_0 had a choice what of its local block columns to update. Since its left neighbour (P_3) has finished with update of its block column 1 on sweep 1, process P_0 can compute for its second block column on sweep 1. But P_0 can also update its first block column on sweep 3, because its right neighbour (P_1) has finished with its first block column on sweep 2.

In practice, the number of block columns usually exceeds amount of processes leading to many local block columns stored and updated by each process. With the number of local block columns per process increase, such situation becomes more probable, what gives choice to a process what block column to update. This provides additional freedom to update the local block columns not in some a prescribed order, but to start with an update of that block column, for which the required data (Q_{old} from the left and b_{B} columns from the right) is already available.

This approach is implemented as follows: every process initiates an asynchronous receive operation for each of its local block columns. Then it checks in a pooling loop, if it has received all the necessary data from the left and from the right for any of its local block columns (using the **test** operations). If data is available for some of the block columns, the process starts with an update of this block column immediately and initiates a new asynchronous receive for this block column for the next sweep. Thus, a process will not wait until data transfers required to update the i_1-th local block column on sweep j_1 is finished, if data for an update of another i_2-th local block column on sweep j_2 is already available.

The actual data transfers are asynchronous, with arrival of data directly being followed by issuing the next **receive**. A process can receive data for his local block columns updates in a background of computations: while updating a block column on sweep k, a process receives data for the update of this block column on the next sweep $k + 1$, as well as for the other its local block columns, if the corresponding receive operations have already been initiated for the other block columns. Thus, such approach essentially implements a distributed-memory task-based execution model without requiring an associated software framework.

Another improvement of the algorithm can be achieved by reordering of operations during a local update in such a way, that the data sends to the

neighbours are initiated as fast as possible. Thus, the other processes can start with their updates earlier. The optimal procedure is the following:

1 do POST operation

2 do QR and send the new reflectors to the right

3 do SYM operation using the old reflectors and send the first b_{B} columns of the block column to the left

4 do PRE operation by applying the new reflectors.

Algorithm 6 presents the whole procedure to update a block column. The checks if a process has to send or receive data have been omitted in the algorithm to keep it concise. As it was already mentioned, process P_0 doesn't need to receive reflectors from the left to update its first local block column. Also, processes need data from the left (reflectors) only and nothing from the right for the first sweep. A process needs to send data to the right only if the right neighbour has to update the corresponding block column on the next sweep after shifting the band to the left through the processes. For example, process P_2 (Figure 28) will not update its second local block column (global columns from 38 to 40) starting from the second sweep: only 37 columns will be processed on the second sweep, 34 columns will be updated on the third sweep and so on. In such a way, if the right neighbour does nothing for a specific block column any more, then the reflectors are not sent to him, as well as receives of b_{B} columns are not initiated by its left neighbour for the globally adjacent block column. In the presented case, P_1 will not call the receive function for its second local block column after the first sweep. However, it will continue receiving data for its first local block column. All these checks are not presented in the algorithm for a better readability.

Algorithm 6: Update of block column j according to sweep k, to be executed in process P_j. The corresponding receives of $\mathsf{Q}_{\text{old}} \equiv \mathsf{Q}_j^{(k)}$ and of b_{B} columns of A from P_{j-1} and P_{j+1} respectively are finished.

1 initiate **receive** for next $\mathsf{Q}_j^{(k+1)}$ from the left ;
2 initiate **receive** for next columns from the right ;
3 do POST with Q_{old} on subdiagonal block $\mathsf{A}_{j+1,j}$;
4 do QR on first b_{B} columns of subdiagonal block $\rightsquigarrow \mathsf{Q}_{\text{new}} \equiv \mathsf{Q}_{j+1}^{(k)}$;
5 **send** Q_{new} to right neighbour ;
6 do SYM with Q_{old} on diagonal block A_{jj} ;
7 **send** first b_{B} columns of A_{jj} and $\mathsf{A}_{j+1,j}$ to left neighbour ;
8 do PRE with Q_{new} on remaining $b_{\mathsf{A}} - b_{\mathsf{B}}$ columns of subdiagonal block ;

The numerical experiments were performed on the COBRA system at the Max Planck Computing and Data Facility (MPCDF), Garching. The COBRA nodes, that were used, feature two Intel Xeon Gold 6148 (Skylake) processors, each with 20 cores running at 2.4 GHz, and the nodes are connected with a 100 Gb/s OmniPath non-blocking, full fat tree interconnect. All computations were done with double precision real data.

Figure 30 shows the strong scaling of the implementation for reducing matrices of size n = 8Ki = 8,192, n = 16Ki = 16,384, and 32Ki = 32,768, and initial bandwidths $b_\mathsf{A} = 64, 128, 256$ to bandwidth $b_\mathsf{B} = 32$. The data reveal almost perfect scaling up to roughly $n/4b_\mathsf{A}$ processes and a maximum speedup close to $n/2b_\mathsf{A}$.

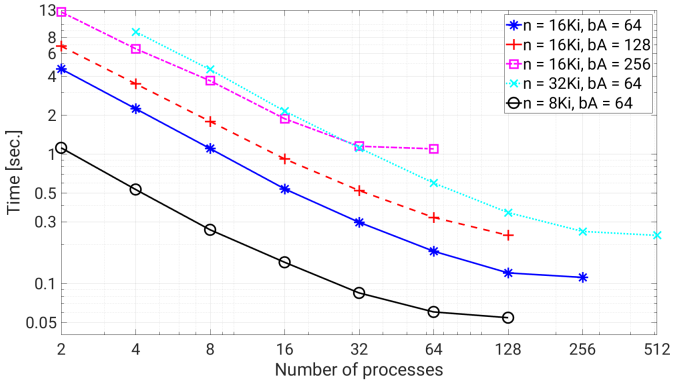

Figure 30: Strong scaling for the bandwidth reduction on COBRA (32 processes per node).

Except for end-of-scaling effects, the timings also reflect the arithmetic complexity of the reduction: doubling b_A roughly doubles the time T (keeping p fixed), whereas doubling n quadruples T, corresponding to the number of floating-point operations being of order $\mathcal{O}(b_\mathsf{A} n^2)$ [3].

3.3 Efficient block transformations

The SYM, POST, QR, and PRE operations for each block column can be realized in several ways [20].

Since every reflectors matrix $\mathsf{Q}_j^{(k)}$ represents result of a QR decomposition of a block with at most b_B columns, it is the product of at most b_B Householder matrices, $\mathsf{Q}_j^{(k)} = \mathsf{H}_1 \cdots \mathsf{H}_m$ with $m \leq b_\mathsf{B}$ and $\mathsf{H}_i = \mathsf{I} - \mathsf{y}_i \, \tau_i \, \mathsf{y}_i^H$, where $\tau_i = 2/\|\mathsf{y}_i\|^2 \in \mathbb{R}$.

One way to implement the updates is to apply these Householder transformations one-by-one, making use of the special structure of H_i. For example, the POST operation can be done as follows: H_i: $\mathsf{A}_{j+1,j} := \mathsf{A}_{j+1,j} \, \mathsf{H}_i = \mathsf{A}_{j+1,j} - \mathsf{z}_i \, \mathsf{y}_i^H$ for $i = 1, \ldots, n$, where $\mathsf{z}_i = \mathsf{A}_{j+1,j} \, \mathsf{y}_i \, \tau_i$. The procedure for PRE is similar. Each such update includes a $b_\mathsf{A} \times b_\mathsf{A}$ matrix-vector multiplication and a rank-1 update per Householder transformation, thus making a total arithmetic complexity equal to $m \cdot 4b_\mathsf{A}^2$ arithmetic operations.

The SYM operation involves a two-sided application of a Householder transformation and relies on a a Hermitian rank-2 update: $\mathsf{A}_{jj} := \mathsf{H}_i^H \, \mathsf{A}_{jj} \, \mathsf{H}_i = \mathsf{A}_{jj} - \mathsf{y}_i \, \mathsf{v}_i^H - \mathsf{v}_i \, \mathsf{y}_i^H$, with $\mathsf{v}_i = \mathsf{z}_i - (\tau_i \, (\mathsf{z}_i^H \, \mathsf{y}_i)/2) \cdot \mathsf{y}_i$ and $\mathsf{z}_i = \mathsf{A}_{jj} \, \mathsf{y}_i \, \tau_i$. Since the diagonal blocks before and after updates are Hermitian, this procedure can be implemented in a way to affect a one triangle (e.g. the lower one) of the A_{jj} block. With such implementation, SYM also takes $m \cdot 4b_\mathsf{A}^2$ arithmetic operations.

Application of the Householder transformations one-by-one allows only using level-2 BLAS operations [15, 16], with performance being far from the peak performance on the current supercomputer systems due to less efficient memory access. To utilize level-3 BLAS operations the WY representation can be used.

The compact WY representation [31] represents the product of Householder transformations as $\mathsf{H}_1 \cdots \mathsf{H}_m = \mathsf{I} - \mathsf{Y}\,\mathsf{T}\,\mathsf{Y}^H$, where $\mathsf{Y} = (\mathsf{y}_1, \ldots, \mathsf{y}_m)$ is $b_\mathsf{A} \times m$ and T is an $m \times m$ triangular matrix. At first, matrix T must be generated. Then, POST can be done as three matrix-matrix multiplications $\mathsf{A}_{j+1,j} := \mathsf{A}_{j+1,j} - \mathsf{Z}\,\mathsf{Y}^H$, where $\mathsf{Z} = \mathsf{V}\,\mathsf{T}$ and $\mathsf{V} = \mathsf{A}_{j+1,j}\,\mathsf{Y}$, and similarly for PRE.

The triangular factor T can be generated from a given set $\mathsf{y}_1, \tau_1, \ldots, \mathsf{y}_m, \tau_m$ of Householder transformations with a `_LARFT` routine of the `LAPACK` library [1]. To apply the compact WY representation to a matrix form left or right (PRE and POST operations) the function `_LARFB` can be used.

The `LAPACK` library contains no routine to realize the SYM update. Similarly to the non-blocked case, the SYM operation $\mathsf{A}_{jj} := (\mathsf{I}{-}\mathsf{Y}\,\mathsf{T}\,\mathsf{Y}^H)^H\,\mathsf{A}_{jj}\,(\mathsf{I}{-}\mathsf{Y}\,\mathsf{T}\,\mathsf{Y}^H)$ can be rewritten as a Hermitian rank-$2m$ update $\mathsf{A}_{jj} := \mathsf{A}_{jj} - \mathsf{Y}\,\mathsf{V}^H - \mathsf{V}\,\mathsf{Y}^H$, where $\mathsf{V} = \mathsf{Z} - \frac{1}{2}\,\mathsf{Y}\,\mathsf{S}$ with $\mathsf{S} = \mathsf{W}^H\,\mathsf{Z}$, $\mathsf{Z} = \mathsf{A}_{jj}\,\mathsf{W}$, and $\mathsf{W} = \mathsf{Y}\,\mathsf{T}$. That involves four matrix-matrix multiplications, with one triangular and one symmetric matrix product among them. The increased performance of the level-3 BLAS operations is achieved with a price of $\mathcal{O}(m^2 b_\mathsf{A})$ additional operations per block column. This is, however, negligible if $m \ll b_\mathsf{A}$.

In order to decrease amount of generations of matrix T from the Householder reflectors, a process can send its already generated T to the right neighbour with the reflectors together. Thus, the destination process can use the received matrix T for its computations. That doesn't increase the communication volume, because T is a $b_\mathsf{B} \times b_\mathsf{B}$ upper triangular matrix and the reflectors are stored as a $b_\mathsf{A} \times b_\mathsf{B}$ trapezoidal matrix. Consequently, it is possible to copy T to the upper part of the reflectors array without loss of data and send the whole chunk of data as a rectangular matrix to the right.

The timings presented on Figure 30 were obtained with the transformations done as it was just explained. For the QR decomposition the `LAPACK` routine `DGEQRF` was used. In the following we will call this a LAPACK-based version of the algorithm.

Alternatively, one might consider the "standard" WY representation [5] for a product of Householder transformations, $\mathsf{H}_1 \cdots \mathsf{H}_m = \mathsf{I} + \mathsf{W}\,\mathsf{Y}^H$, where again $\mathsf{Y} = (\mathsf{y}_1, \ldots, \mathsf{y}_m)$, and W is a $b_\mathsf{A} \times m$ matrix. This approach has been realized in the SBR toolbox for **S**uccessive **B**and **R**eduction [4, 3]. The SBR toolbox provides routines `_GEWYG` to generate the W factor, `_GEWY` to apply the blocked transform to a general matrix from the left or right (PRE and POST operations) and `{Z,C}HEWY` and `{D,S}SYWY` routines for applying it from both sides to a complex Hermitian or real symmetrix matrix (SYM). Again, mainly level-3 BLAS operations are used, and $\mathcal{O}(m^2 b_\mathsf{A})$ additional operations per block column are required, as compared to the non-blocked, level-2 BLAS approach. This implementation is called the SBR-based version in the following.

Figure 31 compares the parallel performance of the bandwidth reduction using the SBR-based transformations with the LAPACK-based variant (the latter data are identical with Figure 30) and a version from [2] made by Thomas Auckenthaler. This implementation relies on another way to distribute matrix and uses different scheme of parallelization instead of the block cyclic distribution of a matrix. Namely, for matrix size N and process number p, the first batch of $\frac{N}{p}$

adjacent columns belongs to the first process, the second batch belongs to the second process and so on. If N is not a multiple of p, then the largest amount of columns is stored by the last process in order to achieve a better workload balance.

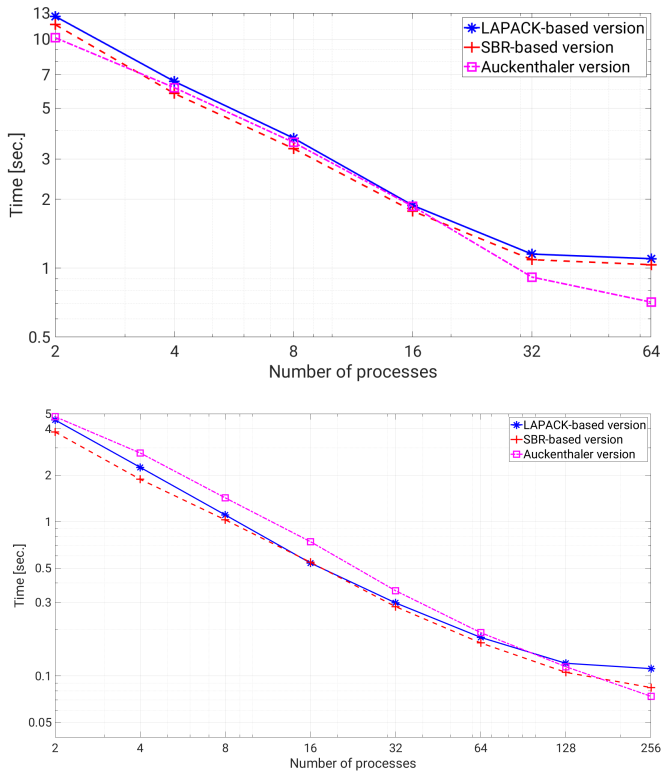

Figure 31: Timings for 1D reduction, $n = 16\text{Ki}$. Upper: $b = 256 \rightarrow 32$. Lower: $b = 64 \rightarrow 32$.

Such an approach has an obvious advantage: it leads to a better optimized communication scheme. Since a process stores adjacent block columns of a matrix, it doesn't need to receive data from left and right neighbours for updates of all of its block columns except of the first and last (boundary) ones. While the amount of block columns per process stays relatively high, the new algorithms partly compensate a necessity to receive data by a clever communication driven choice of a local block column to update, combined with the asynchronous data transfers. But with amount of local block columns per process decrease, the new implementations suffer from a lack of flexibility in a choice of which of the local block columns to update.

However, a data distribution used in the Auckenthalers' implementation leads to a worse workload balance. If every process has n local block columns to update, then the last process will have $n \cdot r$ updates to do with $r = \frac{b_A}{b_B}$, the second-last process will have $2 \cdot n \cdot r$ updates to run, the next process will participate in $3 \cdot n \cdot r$ updates and so on. Finally, the first process will have the

largest volume of calculations to do.

With $r = \frac{b_{\mathsf{A}}}{b_{\mathsf{B}}}$ and $s = \frac{n}{p \cdot b_{\mathsf{A}}}$, the numbers of local updates can be calculated as follows:

- First process, the new algorithm: $\frac{p \cdot r \cdot s \cdot (s+1)}{2}$;
- First process, Auckenthalers algorithm: $r \cdot s \cdot \left(s \cdot (p-1) + \frac{s+1}{2}\right)$
- Last process, the new algorithm: $r \cdot \left(s + \frac{p \cdot s \cdot (s-1)}{2}\right)$
- Last process, Auckenthalers algorithm: $\frac{r \cdot s \cdot (s+1)}{2}$

Comparisons of the ratios of the local updates numbers for the first (most loaded) and the last (less loaded) processes are presented on Figure 32. One should not expect a strict dependency between the workload imbalance and the algorithm performance thanks to the pipelined manner of the both implementations, and of the Auckenthalers realization in particular: the first process has more job to do, but its starts calculations earlier and works on the new sweeps while the last process is idle.

However, according to the Figure 32 one could expect the largest difference in the workload balances between the two implementations for the middle range of the process numbers. And that is exactly the area, where the new implementation outperforms the Auckenthalers one (cf. Figure 31). The effect is stronger for a smaller b_{A}, what implies more local block columns per process (lower part of the figure) and less pronounced for a larger b_{A} (upper part of the figure).

The new version cannot benefit from the communication driven implementation features on the grids of 2 processes, because the left and the right neighbours are represented with the same process, what leads to the mostly predetermined order of the local updates without a sufficient flexibility. Also, the workload imbalance distinction between the new and Auckenthalers' realizations is not dramatic in this case.

With the close to maximal amount of processes, the Auckenthalers' implementation is again better due to the facts, that firstly, an increase of the process numbers leads to smaller amount of the local block columns per process, implying a less flexible regime of our algorithm, and secondly, the workload imbalances approach each other as the process number grows. At the very end the data distribution schemes of the both implementations converge to the same representation: one local block column per process.

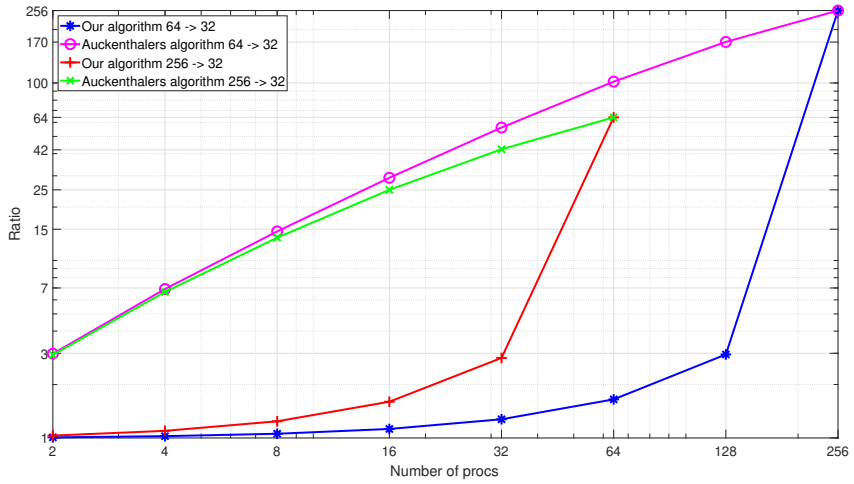

Figure 32: Ratios of numbers of updates for the most and less loaded processes. A smaller ratio indicates better workload balance.

Comparing the SBR-based and the LAPACK-based versions one could notice a slightly larger performance of the SBR-based implementation.

3.4 Exploiting parallelism within each block

In order to utilize more processors, each local update may be fulfilled on many processes in parallel. To achieve this, every block column (only nonzero elements) is distributed on a separate process grid (sub-grid). That means, that every block of size $2b_{\mathsf{A}} \times b_{\mathsf{A}}$ is distributed on a $p_r \times p_c$ process sub-grid (or process group) with torus wrapped mapping of size-b_{B} blocks. Then the local updates are done using the parallel multiplication routines. A process group may store and update several block columns, but they are mapped on the process grid separately one after another. For example, if a process group contains 2 block columns with b_{A} columns each, then initially the first b_{A} columns are distributed on the grid, and after the same is done with the last b_{A} columns. This mapping differs from what we would have, if we distributed the whole block of $2 \cdot b_{\mathsf{A}}$ columns over the process group.

The parallel running of the updates provides us an additional level of parallelism by a price of some code complications. Now not just a one process must decide which of its block columns to update (for which local block column it has received all the necessary data: b_{B} columns from the right neighbour and reflectors Y with τ values from the left), but a group of processes must make a decision cooperatively. The neighbours are not the single processes now, but the process groups.

The updates of block columns are not done locally, but with the parallel routines now. An efficient scheme of the update, allowing to avoid unnecessary data transfers, will be presented in the next sections.

In addition, a shift of band to left-up looks more complicated for a 2D case. For a 1D case it was enough for a process to receive data from the right

neighbour and copy it locally to a proper position in array. With a 2D data distribution, a process has to choose a suitable process from the group to the right (to determine a correct process row of the source). Also, an upward shift of data must be done inside of a process group having more than one process row. A scheme to avoid data transfers along process columns for such shifts will also be presented.

3.4.1 Choosing a block column to update

Different processes play different roles in a group. The left-most processes of the group receive Householder reflectors from the group to the left, and the right-most processes receive the b_B columns from the group to the right. The presented implementation relies on an assumption, that if one process has finished with the receive, then all the other processes of his column have also done or will do it soon. That is why, a receiving of reflectors is checked only on one of the left-most processes (namely, on the top-left process: it will be called left leader of the group in the following). For the receiving from the right, the actual data receives are tested on the top-right process (right leader of the group) only. Under some circumstances (namely, if $\frac{b_A}{b_B}$ is a multiple of $p_c + 1$) the left and right leaders of the group are represented by the same process.

The left leader finds a local block column, for which the corresponding receive of reflectors is finished, and sends the index of the block column to the right leader (this block column is a potential candidate to be updated, if the corresponding data from the right neighbouring group has arrived). Right leader checks, if the corresponding receive from the right neighbour for this block column is fulfilled. If yes, it sends `True` to the left leader, and the index of the block column to be updated is broadcasted over the process group (sub-grid). All the processes, who took part in receives (left- and right- most processes of the group), call the `MPI_Wait` function to ensure a successful receive of elements needed for the update. If the right leader has not yet received b_B columns from the right, it sends `False` to the left leader, and the left leader searches for the next local block column with the finished receiving of reflectors Y from the left neighbouring sub-grid.

If b_A is not a multiple of b_B, then not only the right-most processes of the sub-grids receive data from the right, but the second-right processes do it also. For example, if $b_A = 17$ and $b_B = 4$, then the right-most processes of the group will receive 1 column, and the second-right processes will get 3 columns from the sub-grid to the right. In such cases, when the second-right processes receive more data than the right-most ones, we also need to ask them, if they have finished with their receive (we ask a right sub-leader indeed: top process in the second column from the right in the process group). If the second-right processes of the sub-grid receive less data than the right-most ones, we do not ask them about the receive: the probability is high, that they have already got all the data, since their right neighbours in the sub-grid have completed a receive of the larger amount of data from the same source during approximately the same time. In this case the second-right processes simply call a `MPI_Wait` function.

3.4.2 Communications between and inside the groups

In contrast to 1D case, where it was enough for a process to send the first b_B columns to the left neighbour, in a 2D case a process has to choose a proper destination (process row) in a group to the left. Figure 33 shows two adjacent process groups (the global picture can be found on Figure 28). The whole block of $2 \cdot b_\mathsf{A}$ rows is distributed over 3 process rows and consists of 2 parts with b_A rows each: the upper part, to which the SYM operation is applied, and the lower one, which is updated with the POST, PRE and QR operations. The dotted rectangle shows elements to be updated by the first process group on the next sweep. For simplicity reasons it is assumed, that b_A is a multiple of b_B. That means, that the left-most processes of group 2 send the updated columns to the right-most processes of group 1 only.

Process with row index 0 needs to send its part of data to the starting process having the lower part (this is process row 1 in our case). The row index of this process can be calculated as $P_{StartLow} = (\frac{b_\mathsf{A}}{b_\mathsf{B}})\%p_r$, where % represents a modulo operation. Process of row 1 has to send data to the row 2 of the left neighbour and so on. In general, every process can determine a row index of its destination as $(P_{StartLow} + \mathsf{myRow})\%p_r$.

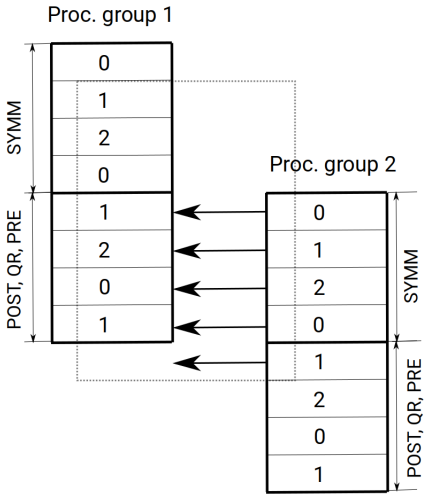

Figure 33: Data transfers between adjacent process groups to realize a band shift. By numbers from 0 to 2 the process rows are represented. The dotted rectangle indicates elements to be updated on the group 1 during the next sweep.

After a block column is updated, the band is shifted left-up. For the 2D case this shift not only includes communication with the neighbouring groups, but data transfers inside of the group also. However, communications for the shift up can be avoided (among process rows in groups).

According to Figure 33, if we had one additional block row of size b_B in the block, then it could be possible to shift a matrix descriptor instead of real data transfers. A block column can be updated then as being distributed over the same 3 process rows, but starting with row 1 instead of 0-th (it can be easily

achieved by changing the matrix descriptor slightly). Similarly, on the following sweep, data can be treated as a matrix distributed over 3 processes with the starting process row 2 (the last row in a grid). However, 2 additional block rows are required in this case. For the next sweep, the block column is already distributed starting with the 0-th process row. Thus, every process just needs to shift its data locally by b_B rows up to achieve desired distribution. In such a way, having $p_r - 1$ additional block rows (1 additional block row per process is allocated), it is possible to completely avoid communications to represent a vertical shift.

3.4.3 Efficient block column update

As it was discussed, the bandwidth reduction algorithm works on two parts of the block column: upper part having b_A rows is updated with the SYM operation, and on the lower part the operations QR, POST and PRE are fulfilled. In order to achieve higher performance, these calculations must be parallelized as much as possible. If enough process rows ($\geq \frac{2b_A}{b_B}$) are available, the separate communicators for the upper and lower parts are built, and the updates are fulfilled independently in parallel to each other (Figure 34, c) on these communicators. In this case two sub-grids are built on a process group, one above another, instead of a single one. However, a communication free method to realize the vertical shifts has a more complicated form in this case, because update of a block column on the next sweep will include another set of process rows than on the current one, namely, processes from 1 to 4 instead of the ones from 0 to 3. The corresponding idea will be presented in details later in Section 3.5.

During the first experiments on supercomputers it was noticed, that a QR algorithm for one block column of size $b_A \times b_B$ (i.e. performing on one process column) runs slowly. The variants of the algorithm with the QR step executed on a single process show significantly better performance. 3 different variants of this idea were implemented:

- OneDoes - OneSends: assemble the whole QR block of size $b_A \times b_B$ on a single process; this process performs the QR operation of the whole block and sends the reflectors by parts to all the needed destinations of the right process group.
- AllDO - AllSend: assemble the whole QR block of size $b_A \times b_B$ on all the processes of the first process column (left-most processes of the group); every such process performs the QR operation of the whole block; every such process sends its part of the reflectors to the right process group.
- OneDoes - AllSend: assemble the whole QR block of size $b_A \times b_B$ on a single process; this process performs the QR operation of the whole block and redistributes the reflectors over all the other processes of the first process column; every process sends its part of the reflectors to the right process group.

A motivation for the first variant is simple: communication volume for the data assembling is reduced: it is cheaper to collect all the data on a single process. It makes it also possible to run the QR operation fully in parallel to the SYM calculation, if at least $\frac{b_A}{b_B} + 1$ process rows are available: it is simply possible to choose a process, which is not involved in the SYM update, and to

run the QR operation on it (Figure 34, c). However, the obvious drawback of the variant is that it is only a single process, who sends all the Householder reflectors to the right group. It has to prepare data (copy operation) for all the destinations and send it, while all the other processes of the column are idle.

That gives a motivation to involve all the processes of the left-most process column in data sending of the reflectors. One obvious way could be to redistribute results of the QR operation among all the processes of the column, and then all such processes send their part of data to the right. This scheme still allows to run QR in parallel to the SYM operation with relatively low process rows number, but now we face the necessity of the additional communications to redistribute data along process column. This step is poorly balanced, since one process (one that performed the QR operation) has to participate in more data transfers than the others.

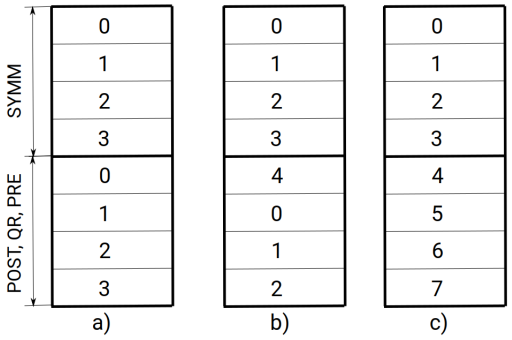

Figure 34: Distribution of a block column over process groups with different process rows numbers. a): update upper and lower blocks consecutively; b): process 4 can make QR in parallel to the SYM update on processes from 0 to 3; c): updates of lower and upper parts run in parallel fully independently.

To avoid this disadvantage the `AllDO - AllSend` scheme was implemented. The whole QR block is assembled on each of the processes of the first column. After performing of the calculation, every process sends the needed part of reflectors to the right. Such an approach leads to a better scalability with the process groups increase, but it requires more communications among process rows in order to collect the whole QR block on every process. That is why one could expect a performance decrease for the process sub-grids having many rows.

Application of the parallel `ScaLAPACK` routines on the sub-grids of process groups would be not very efficient, because some of the operations involve just tall or wide matrices distributed over one process column or row of the group. Instead of calling the parallel matrix multiplication functions on the whole communicators of the groups, it is more efficient to do calculations locally and to take care about the data transfers manually.

According to Section 3.3, the SYM update in the compact WY representation consists of the following steps:

1. $\mathsf{W} = \mathsf{YT}$: a multiplication of a $b_\mathsf{A} \times b_\mathsf{B}$ matrix Y, distributed over the first

process column of a group, by a $b_B \times b_B$ triangular matrix T.

2. $Z = A_{jj}W$: a multiplication of a symmetric $b_A \times b_A$ matrix A_{jj}, distributed over the process group, by a $b_A \times b_B$ matrix W; both Z and W are distributed over the first process column of a group.

3. $S = W^H Z$: a multiplication of a $b_B \times b_A$ matrix W^H by a $b_A \times b_B$ matrix Z with a $b_B \times b_B$ matrix S as a result.

4. $Z = Z - \frac{1}{2}YS$, with $b_B \times b_A$ matrices Z and Y distributed over the first process column, and $b_B \times b_B$ matrix S

5. $A_{jj} := A_{jj} - Y Z^H - Z Y^H$, a Hermitian rank-$2m$ update

The first step is done as follows: the matrix T is broadcasted along the first process column, and then every process of the column multiplies its local part of matrix Y by T. For the second item a `ScaLAPACK` function `P_SYMM` for symmetric matrix multiplication is invoked on the whole process group.

Multiplication $S = W^H Z$ is calculated in the following way: every process of the first process column multiplies locally its part of W with transposition by its local part of Z. Then a collective reduction `MPI_Allreduce` is called in such a way, that the resulting S is available on every process of the first process column. Now the 4-th step can be done purely locally: every process uses its local blocks of Z and Y with the previously calculated S. The final step is done with a `ScaLAPACK` routine `P_SYR2K` invoked on all processes of a process group.

Figure 35 presents results of the 2D reduction using the 4-by-4 (upper part) and 8-by-4 (lower part) process sub-grids (groups). $b_A = 512$, $b_B = 32$, $n =$ 16Ki, tests were run on the Cobra supercomputer system. For example, a run on the 64 processes in a case of 4-by-4 groups implies 4 sub-grids of 16 processes each. As expected, on the grids with a small number of process rows, the `AllDO - AllSend` version delivers the best results for any amount of sub-groups.

For the increased process rows number, the `AllDO - AllSend` implementation is predictably the slowest for a small amount of the process sub-grids: gathering the QR blocks on all the process rows becomes expensive in this case. However, this variant provides the best performance again with the increase of the groups number due to a better scalability.

If the process rows number is increased over the $\frac{b_A}{b_B}$ value (over 16 in our case), then the `OneDoes - OneSends` implementation delivers the best performance, because the QR and SYM operations run in parallel in this case only: it is possible to utilize one process to do the QR operation and send reflectors Y to the right, while the other 16 process rows are working on the SYM update. For the $2 \cdot \frac{b_A}{b_B}$ process rows, the `AllDO - AllSend` variant is the fastest implementation again, because two different grids with $\frac{b_A}{b_B}$ process rows each are used now for updates of the upper and lower parts, thus running these updates in parallel for all the versions. Table 8 presents results for 4 sub-grids of 17-by-4 and 32-by-4 for all the three variants of the algorithm.

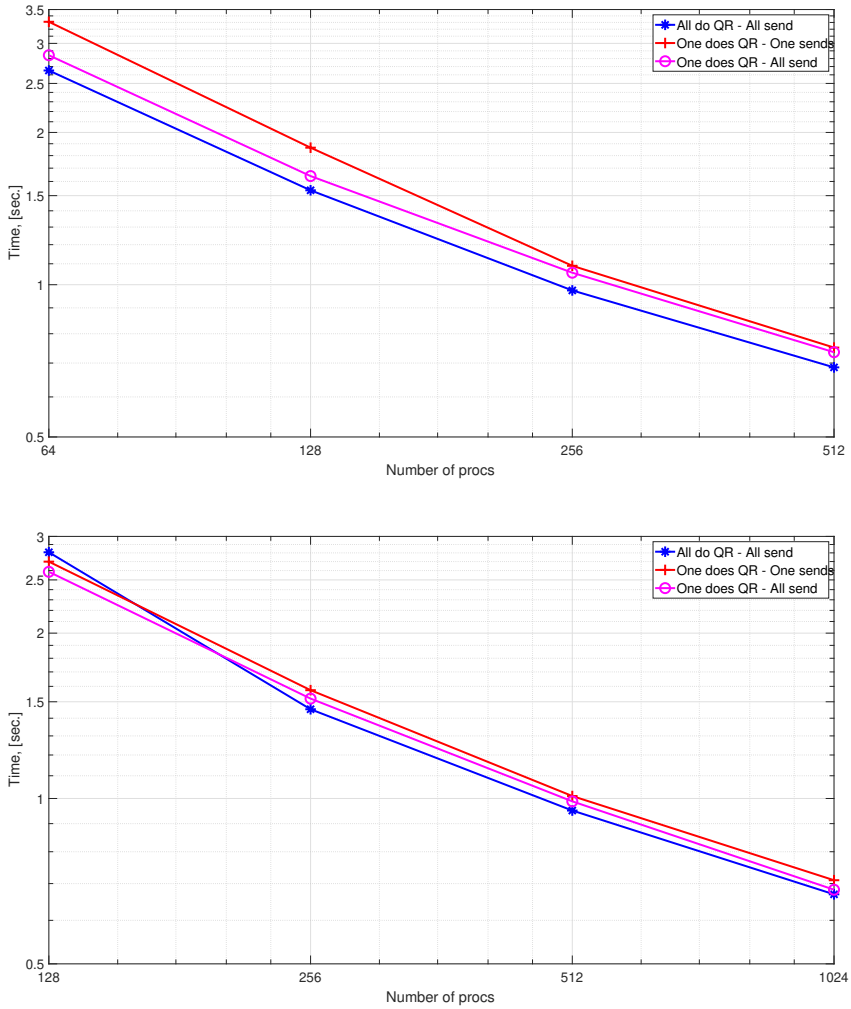

Figure 35: Timings on Cobra: reduction from bandwidth 512 to 32, $n = 16\text{Ki}$. Upper: with the 4 x 4 sub-grids. Lower: with the 8 x 4 sub-grids.

	AllDO - AllSend	OneDoes - AllSend	OneDoes - OneSends
groups 17×4	2.979542 s	3.371378 s	2.634023 s
groups 32×4	2.174447 s	2.228969 s	2.344752 s

Table 8: Timings on Cobra: reduction from bandwidth 512 to 32, $n = 16\text{Ki}$. 4 process groups (sub grids) were used for each run.

Figure 36 presents results for various configurations of the sub-grids. The `AllDO - AllSend` implementation of the algorithm was chosen for these tests, because it provides the best results among all the other variants. It is worth to

mention, that these results may be machine dependent. Since 32 processes per node were used, all the groups configurations except of the 8×8 sub-grids make updates in a shared memory without inter-nodes communications, whereas every 8×8 sub-grid contains 64 cores and is distributed over 2 nodes, thus involving inter-nodes data transfers for each update.

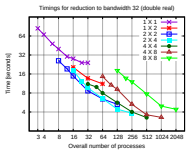

Figure 36: 2D reduction timings on COBRA, $n = 16\text{Ki}$, $b = 512 \rightarrow 32$.

The 2D version of the reduction algorithm can also be compared with the 1D variant showing the best performance for this case (Auckenthalers version) using multithreaded BLAS in order to utilize more processor cores. Matrix size was 16 Ki , $b_{\mathsf{A}} = 512$, $b_{\mathsf{B}} = 32$. For the 1D version, 32 MPI processes with 1, 4 and 8 threads per process were used. Figure 37 shows, that the 2D implementation is a more efficient way to use larger amount of computer resources.

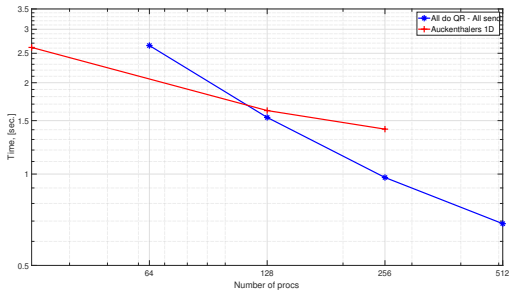

Figure 37: 2D and 1D timings comparison, $n = 16\text{Ki}$, $b = 512 \rightarrow 32$.

3.5 Back transformation of eigenvectors

Once matrix A with bandwidth b_{A} has been reduced to narrower band (b_{B}), $\mathsf{B} = \mathsf{Q}_{\mathsf{AB}}^H \mathsf{A}\, \mathsf{Q}_{\mathsf{AB}}$, and the requested eigenpairs of B have been determined, $\mathsf{B}\, \mathsf{X}_{\mathsf{B}} = \Lambda\, \mathsf{X}_{\mathsf{B}}$, the eigenvectors must be transformed back in order to find the eigenpairs of the initial matrix A: $\mathsf{X}_{\mathsf{A}} = \mathsf{Q}_{\mathsf{AB}}\, \mathsf{X}_{\mathsf{B}}$. As pointed out in Section 3, the reduction is done with a sequence of transformations, $\mathsf{Q}_{\mathsf{AB}} = \mathsf{Q}_1 \cdots \mathsf{Q}_s$ (all unitary/orthogonal transformations $\mathsf{Q}_j^{(k)}$ from all sweeps), and therefore the back transformation becomes $\mathsf{X}_{\mathsf{A}} = \mathsf{Q}_1 \cdots \mathsf{Q}_s \cdot \mathsf{X}_{\mathsf{B}}$, meaning that all these transformations must be applied to X_{B} from the left (no conjugate transposing), and in reverse order.

Note, that if some transformation Q_ℓ acted on b_{A} rows (and columns) $i_1, \ldots, i_2$ of the initial matrix A during the bandwidth reduction procedure, then it affects just the same rows $i_1, \ldots, i_2$ of X for the eigenvectors restore. That implies, that

if two different transformations updated not overlapping row ranges during the bandwidth reduction procedure, then these transformations can be applied independently from each other and run fully in parallel during the eigenvectors restore, because they will affect the row ranges which do not intersect. The row ranges updated by different transformations during a one sweep are disjoint. Thus we can proceed sweep-by-sweep (starting with the last sweep) and apply transformations $\mathsf{Q}_1^{(k)}, \ldots, \mathsf{Q}_{\max_k}^{(k)}$ in parallel and in any order.

This provides the first level of parallelism for the eigenvectors back transformation procedure: every slice of b_A rows of matrix X can be restored in parallel to all the other slices one the same sweep. Additional parallelism can be achieved by doing each transformation of every such slice in a distributed way with many processes.

The transformations from the first reduction sweep affect the largest amount of rows, therefore a description of the algorithm is started from these transformations. However, since the reflectors are applied in a reverse order, these transformations are applied the last for the eigenvectors restore.

The matrix of eigenvectors X is assumed to be distributed in a block cyclic manner with a distribution block size of b_B over a $p_r \times p_c$ process grid, what is a standard `ScaLAPACK` layout. For simplicity of presentation it is also assumed, that X is distributed starting with the last process row of the grid, however this requirement is not necessary for the implementation.

To simplify the presentation, the number of process rows is considered to be a multiple of the ratio of initial to final bandwidth, $p_r = m \cdot \rho$, where $\rho = b_\mathsf{A}/b_\mathsf{B}$. This requirement allows to utilize m process groups with ρ process rows each. Every such process group will apply transformations to a slice having b_A rows of X. Again, the developed routine is implemented to work with any process rows numbers, but such an assumption allows to describe the idea of algorithm in a clearer way.

The algorithm is presented for a case illustrated on Figure 38. The bandwidth reduction was from $b_\mathsf{A} = 128$ to $b_\mathsf{B} = 32$ (what gives $\rho = 4$). With the matrix size of $n = 800$ and $k = 480$ vectors to be back-transformed, X comprises 25×15 size-b_B blocks. A 12×5 process grid is assumed: $m = 3$ process groups of 4×5 each (every such process group is highlighted with its own colour on Figure 38). Every such process group has its own dedicated MPI communicator and represents a separated sub-grid of size $\rho \times p_c$. The first sub-grid comprises processes with the rows indices from 0 to 3, the second group includes processes with the rows indices from 4 to 7 and the last one has processes of rows from 8 to 11.

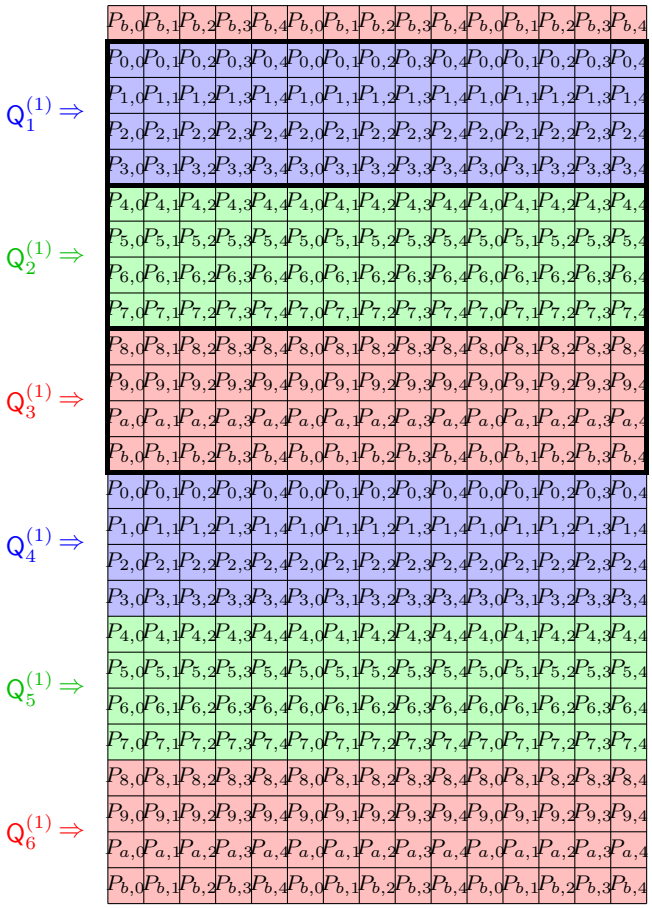

Figure 38: Distribution of a matrix X with 25×15 blocks over a 12×5 process grid for back transformation (row indices are hexadecimal: $a = 10$ and $b = 11$). The background shading corresponds to the sub-grids explained in the main text. Left: Transformations $\mathsf{Q}_j^{(1)}$ from the first reduction sweep.

Recall, that transformations from the first reduction sweep affect all the rows of matrix except of the first b_B ones (the first block row). Thus, the data to be updated is distributed over the 12 process rows starting from the 0-th one. Each transformation updates b_A consecutive rows of X, what corresponds to ρ block rows, and these row ranges do not overlap. Therefore these transformations can be applied simultaneously in parallel, if they are fulfilled by disjoint groups of processes. Since there are m process groups of $\rho \times 5$ each, m transformations ($\mathsf{Q}_1^{(1)}$, $\mathsf{Q}_2^{(1)}$ and $\mathsf{Q}_3^{(1)}$ in our case) can be applied in parallel. Then another m transformations can be applied, and so on, until the sweep is completed.

The second bandwidth reduction sweep (which precedes the first one in the back transformation procedure) affects all the rows of a matrix except of the first 2 block rows ($2 \cdot b_\mathsf{B}$ rows). Now data to be touched is distributed starting with the 1-st process row instead of the 0-th one as it was for the first reduction sweep. In order to update the slices of X on the previously used sub-grids

(having process rows $\{0..3\}, \{1..7\}$ and $\{8..11\}$) we need to "roll" the matrix X through the whole process grid by 1 block row upwards. That would include data transfers and it should be avoided.

This can be achieved by moving the process groups, but not the matrix itself. Indeed, if we had the sub-grids having the following sets of process rows: $\{1..4\}, \{5..8\}$ and $\{9, 10, 11, 0\}$, than it could be possible to keep data in place. The first such process group could update block rows from 2 to 5, the second one - from 6 to 9 and so on. For the next sweep, the first 3 block rows will be skipped and data to be affected is distributed starting with the 2-nd process row. In order to apply the transformations without vertical shift of the matrix rows, the sub-grids with the following sets of process rows are needed: $\{2..5\}, \{6..9\}$ and $\{10, 11, 0, 1\}$. For the next sweep the corresponding sets will be $\{3..6\}, \{7..10\}$ and $\{11, 0, 1, 2\}$. The following sweep can be done with the very first configuration of sub-grids.

Thereby, every process will participate in ρ different sub-grids having different row indices in every sub-grid. Before making an update of a X slice of b_A rows, a process has to determine a proper sub-grid to participate in. All these sub-grids and communicators are established before the whole algorithm starts. A total of ρ sets of m size $\rho \times p_c$ sub-grids are created. Table 3.5 presents all the required configurations for our demonstration case with the block row indices to be transformed by every sub-grid.

Such approach allows to completely avoid data communication while shifting from one sweep to another. Also, no additional memory allocation is needed: data transfers are avoided for free. Exactly the same idea is realized in the implementation of the bandwidth reduction for a case, when the upper and lower parts of a block column are updated on the separate communicators in parallel, i.e. with the process row number $\geq \frac{2b_\mathsf{A}}{b_\mathsf{B}}$ (see Section 3.4.2).

		Sub-grid 0 ("blue")	Sub-grid 1 ("green")	Sub-grid 2 ("red")
Set 0	process rows	$\{0, 1, 2, 3\}$	$\{4, 5, 6, 7\}$	$\{8, 9, 10, 11\}$
	block rows of X	$\{1 \ldots 4, 13 \ldots 16\}$	$\{5 \ldots 8, 17 \ldots 20\}$	$\{0, 9 \ldots 12, 21 \ldots 24\}$
Set 1		$\{1, 2, 3, 4\}$	$\{5, 6, 7, 8\}$	$\{0, 9, 10, 11\}$
		$\{2 \ldots 5, 14 \ldots 17\}$	$\{6 \ldots 9, 18 \ldots 21\}$	$\{0 \ldots 1, 10 \ldots 13, 22 \ldots 23\}$
Set 2		$\{2, 3, 4, 5\}$	$\{6, 7, 8, 9\}$	$\{0, 1, 10, 11\}$
		$\{3 \ldots 6, 15 \ldots 18\}$	$\{7 \ldots 10, 19 \ldots 22\}$	$\{0 \ldots 2, 11 \ldots 14, 23 \ldots 24\}$
Set 3		$\{3, 4, 5, 6\}$	$\{7, 8, 9, 10\}$	$\{0, 1, 2, 11\}$
		$\{4 \ldots 7, 16 \ldots 19\}$	$\{8 \ldots 11, 20 \ldots 23\}$	$\{0 \ldots 3, 12 \ldots 15, 24\}$

Table 9: Assignment of the process rows and the block rows of the matrix X to the $m = 3$ sub-grids in each of the $\rho = 4$ sets. (Matrix and grid size as in Figure 38.)

Note, that matrix X is usually distributed over a process grid starting with the 0-th process row in contrast to our demonstration case, where the matrix distribution starts with the 1-st row. In the presented implementation that simply means, that the 1-st set of sub-grids will be chosen instead of the 0-th one to apply transformations of the first reduction sweep.

Algorithm 7 gives a high-level summary of the back transformation within each process. Note, that at most $2 \cdot \rho$ process rows can be utilized for the bandwidth reduction procedure, because $2 \cdot b_{\mathsf{A}}$ matrix rows are stored and updated. These rows are distributed with a block size of b_{B}. However, the whole matrix X with n rows is affected for the eigenvectors back transformation. Consequently, much more process rows can and should be used. If a process row participated in the bandwidth reduction, then a process of some column in this process row may already have the needed Householder transformations in a form of reflectors Y and triangular matrix T (and it for sure has the needed reflectors, if the process rows amount was not larger than ρ, what means that every process row participated in all the QR operations). On the contrary, the additional process rows, which did not took part in the bandwidth reduction, will have to receive the required transformations to apply them. Since progress is controlled by communication, there is limited potential for asynchrony: some processes may progress to the next sweep (if their whole new sub-grid can do so) while others are still working on the previous sweep. However, the overlapping rows of X must be updated in a certain prescribed order: the transformations from the $i+1$-th reduction sweep must precede the ones from the i-th sweep.

Algorithm 7: Back transformation for one process

```
1  for sweep = last downto 1 do
2  |  adopt my sub-grid for this sweep: choose the proper communicator ;
3  |  for all transformations Y from this sweep do
4  |  |  if I have (parts of) Householder transforms for Y then
5  |  |  |  send them to needed destination ;
6  |  |  end
7  |  |  if I participate in applying Y then
8  |  |  |  if I do not have all data for Y then
9  |  |  |  |  receive them
10 |  |  |  end
11 |  |  |  apply Y (jointly with the whole sub-grid) ;
12 |  |  end
13 |  end
14 end
```

The algorithm can also be extended to the general situation where the above assumptions (b_{A} is a multiple of b_{B}, and p_r is a multiple of $\rho = b_{\mathsf{A}}/b_{\mathsf{B}}$) do not hold.

If p_r is not a multiple of ρ, then a process may participate in different sub-grids during the same sweep (Figure 39). For example, processes of the 0-th row are included both in the blue and red sub-grids to apply $\mathsf{Q}_1^{(1)}$ and $\mathsf{Q}_3^{(1)}$ transforms respectively. That implies, that a suitable sub-grid (communicator) must be adopted to apply every single transformation (instead of once per sweep). Also, some processes may be idle while waiting the other members of its sub-grid. For the case presented on Figure 39, the process rows from 0 to 3 and from 4 to 7 will simultaneously apply transforms $\mathsf{Q}_1^{(1)}$ and $\mathsf{Q}_2^{(1)}$ respectively, whereas the process rows 8 and 9 will not be able to start with $\mathsf{Q}_3^{(1)}$, while process rows 0 and 1 are busy. After the first 2 sub-grids have finished with their updates, the

group of process rows $\{8, 9, 0, 1\}$ starts with application of $\mathsf{Q}_3^{(1)}$ and sub-gird $\{2, 3, 4, 5\}$ applies $\mathsf{Q}_4^{(1)}$. However, process rows 6 and 7 can not start with the $\mathsf{Q}_5^{(1)}$ update, because processes of the 8-th and 9-th rows are busy. In general, it is better to use p_r being a multiple of ρ to achieve maximal efficiency of the algorithm implementation.

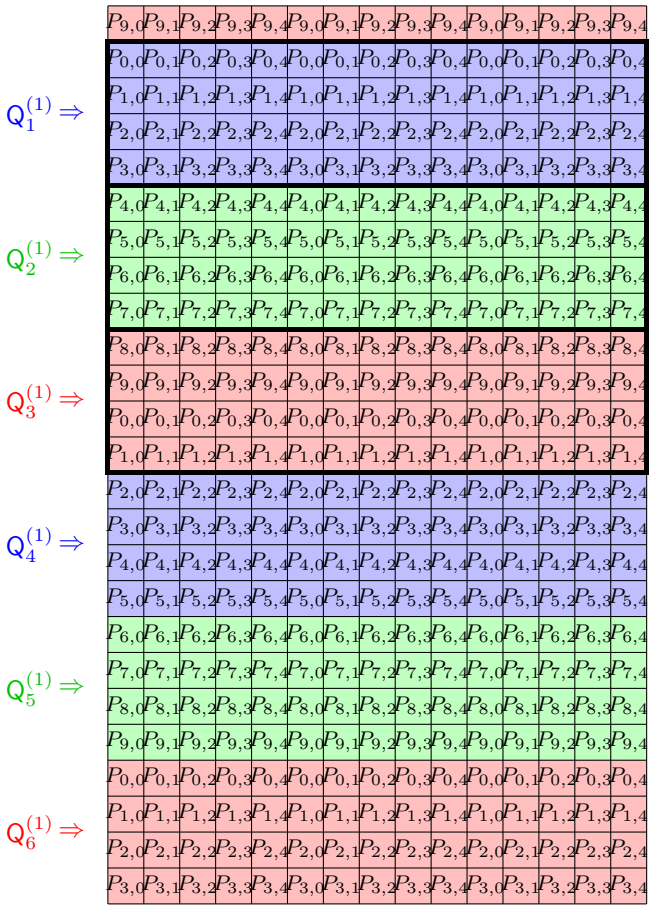

Figure 39: Distribution of a matrix X with 25×15 blocks over a 10×5 process grid for back transformation.

If b_A is not a multiple of b_B, then a process may be involved in the update of two adjacent slices of X. In this case an update of the second slice cannot be started before the update of the first one is finished (top process of the second sub-grid is busy with the update of the first slice in a role of the bottom process of the first sub-grid). Update of the third slice will start only when the update of the second one is finished, and so on: on a high level algorithm performs in a serial way. In order to avoid this, firstly the odd-numbered transformations within each sweep must be applied, and then the even-numbered ones. The updates of the first and of the third block rows for example, can be done in parallel since the corresponding sub-grids have no common process rows. However,

having b_A as a multiple of b_B increases performance of the implementation.

3.5.1 Efficient application of transformations

As it was for the bandwidth reduction, there are again different ways to apply each transformation $\mathsf{X}(i_1 : i_2, :) := Q_\ell \cdot \mathsf{X}(i_1 : i_2, :)$, on a $\rho \times p_c$ sub-grid.

Q_ℓ represents the product of Householder transformations, which already had been determined during the bandwidth reduction. `ScaLAPACK` library contains a routine `PDLARFB` to apply Q_ℓ directly. In a case of eigenvectors back transformation this routine affects a relatively large $b_\mathsf{A} \times k$ block of X in contrast to a small block of size $b_\mathsf{A} \times b_\mathsf{A}$ for the bandwidth reduction procedure. That is why application of this function looks more promising for the eigenvectors restore.

Alternatively, it was also implemented a version with the parallel multiplications of the compact WY representation as it was described in the Section 3.3. According to this procedure, application of the Householder reflectors from the left is represented by the following multiplications:
$\mathsf{X}(i_1 : i_2, :) = \mathsf{X}(i_1 : i_2, :) - \mathsf{W}\mathsf{Y}^H\mathsf{X}(i_1 : i_2, :)$,
with $\mathsf{W} = \mathsf{Y}\mathsf{T}$. Here the updated block of X has a size of $b_\mathsf{A} \times k$, W and Y are $b_\mathsf{A} \times b_\mathsf{B}$ tall matrices, distributed over one process column, and T is a small upper triangular matrix of size $b_\mathsf{B} \times b_\mathsf{B}$.

This update can be realized with the following steps:

1. $\mathsf{W} = \mathsf{Y}\mathsf{T}$
2. $\mathsf{V} = -\mathsf{Y}^H\mathsf{X}(i_1 : i_2, :)$
3. $\mathsf{X}(i_1 : i_2, :) = \mathsf{X}(i_1 : i_2, :) + \mathsf{W}\mathsf{V}$

The first item can be implemented as local multiplications by a triangular matrix T on one process column. T must be broadcasted along this process column before the calculations. For the last two operations the general matrix multiplication routines of the `ScaLAPACK` library can be utilized. This is not a very efficient approach and it can be improved by several ideas. These improvements will be presented one after another to observe the benefits of every idea separately. Finally, the best of the implementations will be compared with the `ScaLAPACK` version using the `PDLARFB` function.

Performance of different versions will be demonstrated for a case of matrix size 16Ki, $b_\mathsf{A} = 512$ and $b_\mathsf{B} = 32$. The corresponding $\rho = \frac{b_\mathsf{A}}{b_\mathsf{B}}$ is 16. 100% of eigenvectors will be restored ($k = n$) on the COBRA supercomputer system. Benchmarks for a version with the general matrix multiplications are presented on Figure 40. The upper plot of the figure demonstrates scalability of the implementation with p_c growth for two different values p_r of 4 and 32. The lower plot presents efficiency of the algorithm as a function of p_r with the constant $p_c = 32$.

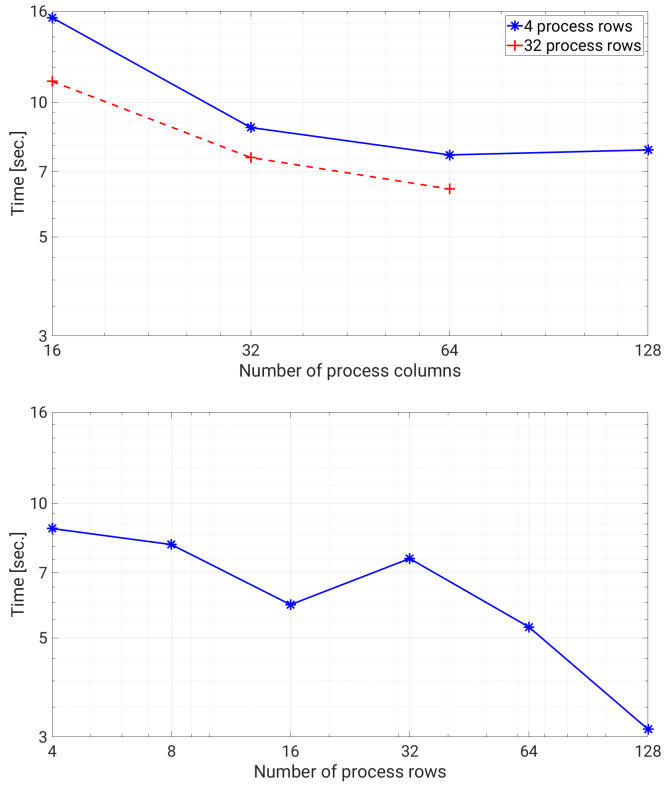

Figure 40: Strong scaling for back transformation of all eigenvectors for the first version of the algorithm. Upper plot: grids of 4 and 32 process rows with different process columns numbers. Lower plot: different process rows numbers with $p_c = 32$.

The first naive implementation has poor scaling as p_c grows. That happens because the `ScaLAPACK` multiplications (items 2 and 3) are applied on wide process grids ($p_c > p_r$). These routines provide lower efficiency for large rectangular grids with a lot of process columns. Also, this implementation scales poorly with the p_r growth. As it was mentioned in Section 3.5, in order to guarantee that every process row contains all the needed Householder transforms in some of the processes of this row (different reflectors may belong to different processes of the row), it is needed that a requirement $p_r \leq \rho$ was true, because all the process rows have participated in every QR operation in this case. However, if a process row needs reflectors resulted in a QR operation, for which this process row was not utilized, then the processes of this row have to receive the required reflectors before the update. That causes a significant drop of efficiency on the lower plot of Figure 40 for $p_r = 32$ in comparison to the grid with $p_r = 16$: since the ρ value is 16, no communications for the Householder transforms are needed for the cases of process rows number less or equal to 16.

At first, the problem of the algorithm scaling with the p_r increase is solved. Since all the needed reflectors are already available before the eigenvectors restore starts, and it is known, which of them will be needed on which processes

and in which order, it is possible to initiate all the sends and receives in the asynchronous regime before the real calculations take place. On the computation phase, if a process needs the specific reflectors in order to make a corresponding calculation, it waits till the appropriate receive is finished and applies the received reflectors. In practice, however, this receive is usually finished by the moment it is needed. While working on first update, a process is receiving data for the next updates. In the following this implementation is called a "send ahead" version of the algorithm.

The results of such an approach are presented on Figure 41. There is predictably no difference between the first and the send ahead versions for 4 process rows, because there are no communications for Householder transforms in cases of $p_r \leq \rho$. However, for the grids with 32 process rows the send ahead implementation has significantly higher performance than the first naive version. The lower picture of the figure shows scalability properties of the both implementations as a function of p_r with constant p_c. The both versions have the same efficiency for all the cases with $p_r \leq \rho$, however for $p_r > \rho$ the new variant scales noticeably better. Also, there is no efficiency drop after crossing the border of $p_r = \rho$.

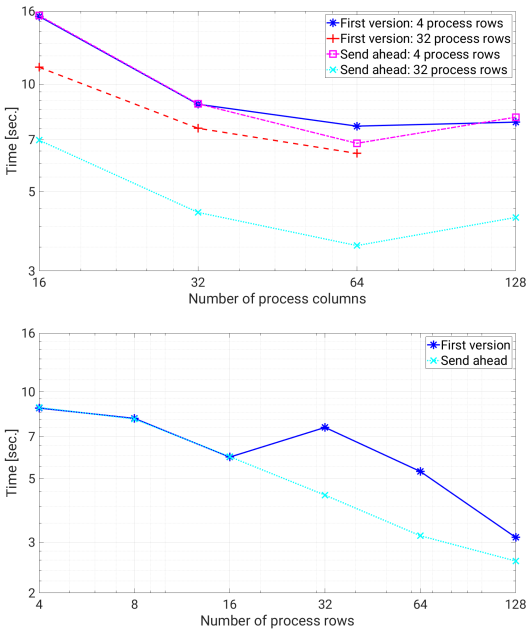

Figure 41: Strong scaling for back transformation of all eigenvectors for the first and send ahead versions of the algorithm. Upper plot: grids of 4 and 32 process rows with different process columns numbers. Lower plot: different process rows numbers with $p_c = 32$.

Though a better scalability with the process rows growth was achieved, a speed up with the process columns increase could be better. An attention should be paid to the last two multiplications of the update: $\mathsf{V} = -\mathsf{Y}^H\mathsf{X}(i_1 : i_2, :)$ and $\mathsf{X}(i_1 : i_2, :) = \mathsf{X}(i_1 : i_2, :) + \mathsf{WV}$, where Y and W have size of $b_\mathsf{A} \times b_\mathsf{B}$ and are

distributed over every process column of a sub-grid (Y was broadcasted over all the process columns and W was calculated on each of them on the first step of the update). Result of the second multiplication $\mathsf{V} = -\mathsf{Y}^H\mathsf{X}(i_1 : i_2, :)$ has shape of $b_\mathsf{B} \times k$ and distributed over the first process row of the sub-grid. Every process column has to compute its block V of size $b_\mathsf{B} \times b_\mathsf{B}$ using Y and its own block column of X. Thus, every process column has all the necessary data to compute its part of V. That is also true for the last operation $\mathsf{X}(i_1 : i_2, :) = \mathsf{X}(i_1 : i_2, :) + \mathsf{WV}$ of the update: every process column can compute its part of X using its own parts of V and W only.

Thus, once the reflectors Y are broadcasted over the process columns, the whole update can be done by each process column separately from the others with no communications among them. That should be much more efficient than applying the parallel multiplication routines on the whole sub-grids having many process columns. For the presented implementation, the separate communicators with grids of size $\rho \times 1$ are built for each of the process columns of a sub-grid, and the multiplication functions are invoked on these one-column grids. Figure 42 compares performance of this version with the send ahead implementation (the new variant is called "updates on columns" implementation).

The upper plot of the figure shows, that the new version scales significantly better than the old one as p_c increases for all the presented values of p_r. Surprisingly, making computations on the process columns improves efficiency of implementation with increase of process rows number also. Processes were ordered by column indices while building the global process grid in the presented experiments. Increase of the process rows number leads to larger distances between processes of different columns in terms of global indexing. Since sub-grids, which are used for computations, include all the p_c process columns, that means that processes of the same sub-grid belong to different nodes of the supercomputer system (not even adjacent ones), if p_r exceeds amount of available cores per node. That significantly slows down parallel routines running on the whole sub-grids because of expensive inter nodes communications. Changing process order from column-wise to row-wise would not solve the problem: distances between process rows (in terms of global indexing) would grow with increase of p_c in this case. However, processes of the same column are adjacent ones in the case of column indexing. Running updates on the process columns separately avoids data transfers among process columns, thus all the calculations can be done on the nodes locally, if every node provides not less than ρ cores (since ρ is a length of the process columns of sub-grids). That was the case for the presented measurements.

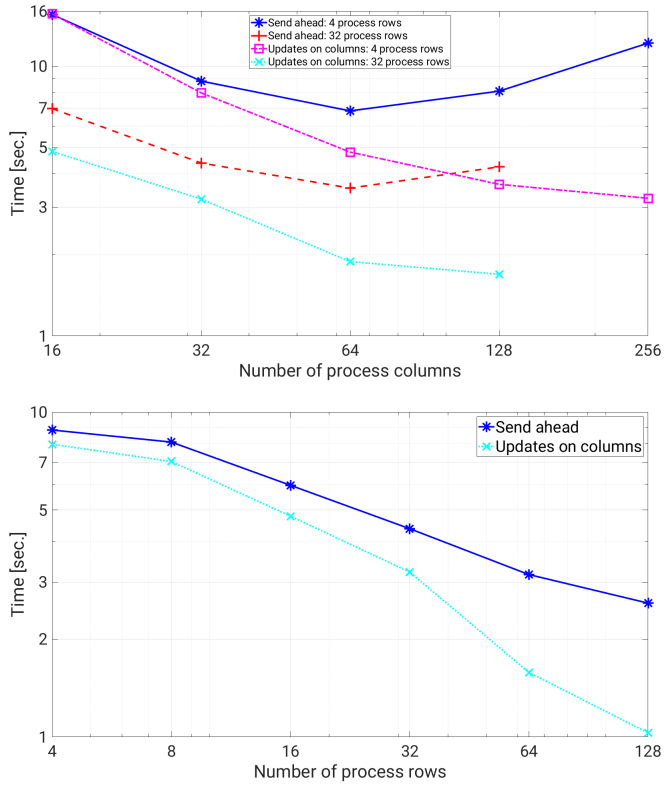

Figure 42: Strong scaling for back transformation of all eigenvectors for the send ahead and on-columns update versions of the algorithm. Upper plot: grids of 4 and 32 process rows with different process columns numbers. Lower plot: different process rows numbers with $p_c = 32$.

The previously mentioned two multiplications $\mathsf{V} = -\mathsf{Y}^H\mathsf{X}(i_1 : i_2, :)$ and $\mathsf{X}(i_1 : i_2, :) = \mathsf{X}(i_1 : i_2, :) + \mathsf{WV}$ can be further improved by optimizing communications on the process columns. During the first multiplication every process calculates its part of V with the following summation (reduction) of the result on the first process of the column. Then this matrix V is used for the second multiplication, what implies data transfers from the first process of the column to the other ones. That leads to redundant communications during these two multiplications.

Performance of the computations can be improved by making communications manually and using `LAPACK` locally to realize calculations on the processes. The scheme of computations on a process column looks as follows:

- Every process calculates its part of V using its local blocks of Y and X.
- The resulting matrix V is summed up on every process of the column with the `MPI_Allreduce` function.
- Every process updates its part of X locally according to the second multiplication.

In such a way, only one all-reduction communication operation of the $b_{\mathsf{B}} \times b_{\mathsf{B}}$ block is required in order to implement the above mentioned multiplications. This version is called "local updates" in the following.

Figure 43 presents efficiency comparison of the new version with the previous one (updates on columns). It can be seen, that the new version provides slightly better performance for all the grid configurations. Efficiency benefit is relatively small for the grid with 4 process row, because the applied idea optimizes communications along process columns (among rows). With small number of process rows this optimization is less significant. However, for the larger values of p_r the improvements are noticeable.

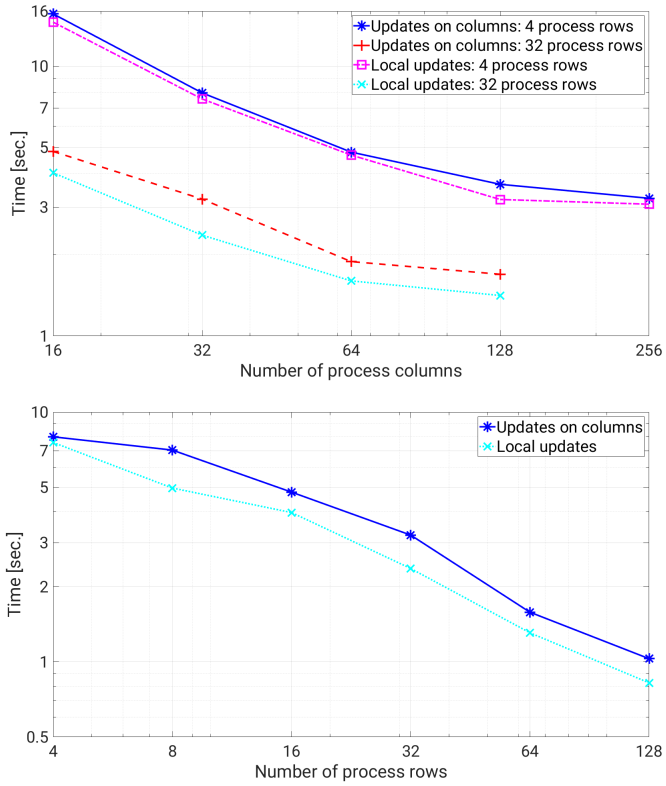

Figure 43: Strong scaling for back transformation of all eigenvectors for the on-columns update and local updates versions of the algorithm. Upper plot: grids of 4 and 32 process rows with different process columns numbers. Lower plot: different process rows numbers with $p_c = 32$.

In Section 3.3 it was mentioned, that it may be beneficial to reduce amount of generations of matrix T by sending it with the Householder transformations to the right neighbours. That doesn't lead to increase of communication volume, because T is copied as an upper triangular matrix to the upper part of the trapezoidal matrix Y. All the previous implementations of the 2D algorithm didn't exploit this idea, but now it is possible to estimate the real improvements of such an approach.

Figure 44 demonstrates comparisons of this implementation with the previous one (the new version is called "send T"). The new variant of the update provides slightly better performance for all process grid configurations.

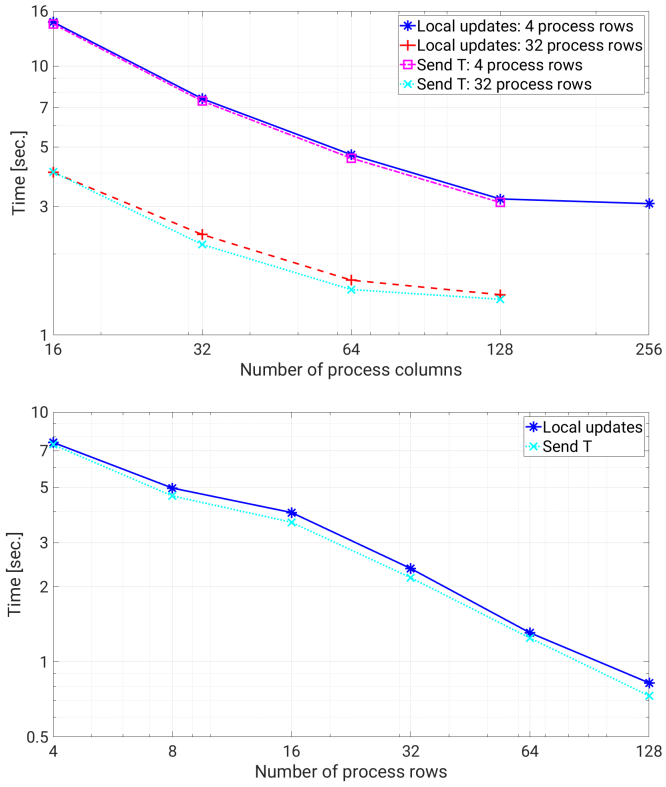

Figure 44: Strong scaling for back transformation of all eigenvectors for the local updates and send T versions of the algorithm. Upper plot: grids of 4 and 32 process rows with different process columns numbers. Lower plot: different process rows numbers with $p_c = 32$.

Eventually, a comparison of the final implementation of the algorithm with the version using the `PDLARFB` routine of `ScaLAPACK` to apply the Householder transformations (the "ScaLAPACK backend" version in the following) is done. For the ScaLAPACK implementation all the possible improvements from the presented version have been applied, including send ahead idea for the Householder reflectors and updates on the process columns. Since the global communication schemes of the both implementations coincide, the performance difference comes from different realizations of the Householder reflectors application on the sub-grids.

Figure 45 presents the corresponding results. The mannual way to apply the Householder transforms is slightly (for small p_r) or noticeably (for larger process rows numbers) faster than the `PDLARFB` function of `ScaLAPACK`. There exists an older routine `PDORM2R` in the `ScaLAPACK` library to apply the Householder reflectors, however it demonstrated significantly lower performance in the tests.

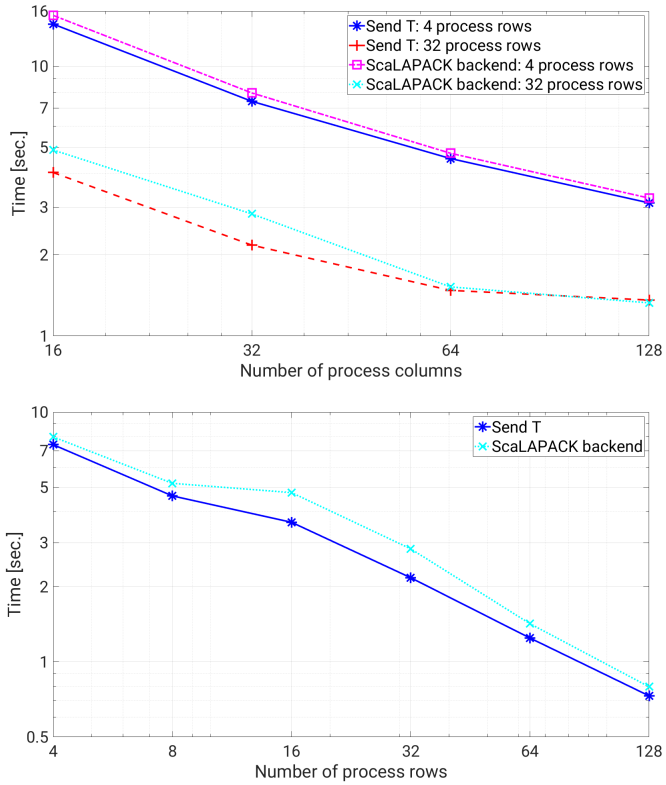

Figure 45: Strong scaling for back transformation of all eigenvectors for the send T and ScaLAPACK update versions of the algorithm. Upper plot: grids of 4 and 32 process rows with different process columns numbers. Lower plot: different process rows numbers with $p_c = 32$.

4 Conclusions

The fast and efficient functions have been implemented to speed up certain steps of eigenvalues solvers. Namely, performance of eigenproblems reductions and of the corresponding eigenvectors back transformations was increased.

The main emphasis was made on parallel performance of the algorithms on modern supercomputers. By applying the most advanced features of the `MPI` library, as well as introducing of tricks and fresh ideas it was possible to achieve higher efficiency of the programs in comparison with the existing solutions.

The presented implementation of the generalized to standard eigenproblem reduction and the corresponding restore of eigenvectors outperform the previous solutions thanks to its following features:

- utilization of triangular structures and symmetry properties of the involved matrices in order to reduce computational complexity and communication volume
- avoidance of the collective communications and of strict synchronization

points

- overlapping of calculations and communications where it is possible
- using the increased memory volume of the modern supercomputers to avoid redundant communications (buffering idea)
- paying much attention to the process grids configurations according to specific features of the tasks to be solved.

The implemented bandwidth reduction and the following back transformation of eigenvectors provide better efficiency as compared to the existing implementations due to:

- wide usage of asynchronous communications and early sends to realize overlapping of communications and calculations
- data driven approach providing maximum of flexibility to the algorithm
- reducing amount of the communication operations by doing them manually where it is necessary and using local `LAPACK` routines instead of the `ScaLAPACK` ones; shifting the process grids instead of sending data where it is possible
- building the most suitable process grids to run certain parallel functions on them (one-column grids)
- implementation of the algorithms to use level-3 BLAS functions.

Acknowledgments

This work was partly supported by the Federal Ministry of Education and Research within the project "ELPA-AEO, Eigenvalue soLvers for Petaflop Applications – Algorithmic Extensions and Optimizations"under Grant 01IH15001.

The author wants to thank the Max Planck Computing and Data Facility (MPCDF), Garching, and the Riken Advanced Institute for Computational Science, Kobe, for providing access to the COBRA, HYDRA and K computer systems, respectively.

List of Figures

List of Tables

Bibliography

[1] E. Anderson et al. *LAPACK Users' Guide*. 3rd. Philadelphia, PA: SIAM, 1999.

[2] Thomas Auckenthaler. "Highly Scalable Eigensolvers for Petaflop Applications". Dissertation. München: Technische Universität München, 2013. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:%2091-diss-20130115-1115722-0-3.

[3] Christian H. Bischof, Bruno Lang, and Xiaobai Sun. "A Framework for Symmetric Band Reduction". In: *ACM Trans. Math. Software* 26.4 (Dec. 2000), pp. 581–601. DOI: 10.1145/365723.365735.

[4] Christian H. Bischof, Bruno Lang, and Xiaobai Sun. "Algorithm 807: The SBR Toolbox—Software for Successive Band Reduction". In: *ACM Trans. Math. Software* 26.4 (Dec. 2000), pp. 602–616. DOI: 10.1145/365723.365736.

[5] Christian Bischof and Charles Van Loan. "The WY Representation for Products of Householder Matrices". In: *SIAM J. Sci. Stat. Comput.* 8.1 (Jan. 1987), s2–s13.

[6] L. S. Blackford et al. *ScaLAPACK Users' Guide*. Philadelphia, PA: SIAM, 1997. DOI: 10.1137/1.9780898719642.

[7] Lang Bruno and Manin Valeriy. "Cannon-type triangular matrix multiplication for the reduction of generalized HPD eigenproblems to standard form". In: *Parallel Computing* (2019). DOI: 10.1016/j.parco.2019.102597.

[8] Lang Bruno and Manin Valeriy. "Efficient parallel reduction of bandwidth for symmetric matrices". In: *Submitted to Parallel Computing* (2022).

[9] Lynn Elliot Cannon. "A Cellular Computer to Implement the Kalman Filter Algorithm". PhD thesis. Bozeman, MT: Montana State University, 1969.

[10] Jaeyoung Choi. "A new parallel matrix multiplication algorithm on distributed-memory concurrent computers". In: *Concurrency Computat.: Pract. Exper.* 10.8 (1998), pp. 655–670. DOI: 10.1002/(SICI)1096-9128(199807)10:8<655::AID-CPE369>3.0.CO;2-O.

[11] Jaeyoung Choi, Jack J. Dongarra, and David W. Walker. "Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers". In: *Parallel Comput.* 21.9 (Sept. 1995), pp. 1387–1405. DOI: 10.1016/0167-8191(95)00016-H.

[12] Jaeyoung Choi, David W. Walker, and Jack J. Dongarra. "Pumma: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers". In: *Concurrency: Pract. Exper.* 6.7 (Oct. 1994), pp. 543–570. DOI: 10.1002/cpe.4330060702.

[13] E. Cuthill and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices". In: *Proc. 24th Nat. Conf. Assoc. Comput. Mach.* Assoc. Comput. Mach., 1969, pp. 157–172.

[14] Eliezer Dekel, David Nassimi, and Sartaj Sahni. “Parallel Matrix and Graph Algorithms”. In: *SIAM J. Comput.* 10.4 (1981), pp. 657–675. DOI: 10.1137/0210049.

[15] Jack J. Dongarra et al. “Algorithm 656—An Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs”. In: *ACM Trans. Math. Software* 14.1 (Mar. 1988), pp. 18–32.

[16] Jack J. Dongarra et al. “An Extended Set of FORTRAN Basic Linear Algebra Subprograms”. In: *ACM Trans. Math. Software* 14.1 (Mar. 1988), pp. 1–17.

[17] V. Fock. “Näherungsmethode zur Lösung des quantenmechanischen Mehrkörperproblems”. In: *Z. Phys.* 1930.1–2 (Jan. 1930), pp. 126–148. DOI: 10.1007/BF01340294.

[18] G. C. Fox, S. W. Otto, and A. J. G. Hey. “Matrix Algorithms on a Hypercube I: Matrix Multiplication”. In: *Parallel Comput.* 4 (1987), pp. 17–31. DOI: 10.1016/0167-8191(87)90060-3.

[19] R. A. van de Geijn and J. Watts. “SUMMA: Scalable Universal Matrix Multiplication Algorithm”. In: *Concurrency: Pract. Exper.* 9.4 (Apr. 1997), pp. 255–274. DOI: 10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2.

[20] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. 4th. Baltimore, MD: The Johns Hopkins University Press, 2013.

[21] Toshiyuki Imamura, Susumu Yamada, and Masahiko Machida. “Development of a High-Performance Eigensolver on a Peta-Scale Next-Generation Supercomputer System”. In: *Progress in Nuclear Science and Technology* 2 (2011), pp. 643–650.

[22] S. Lennart Johnsson. “Minimizing the communication time for matrix multiplication on multiprocessors”. In: *Parallel Comput.* 19.11 (1993), pp. 1235–1257. DOI: 10.1016/0167-8191(93)90029-K.

[23] W. Kohn and L. J. Sham. “Self-Consistent Equations Including Exchange and Correlation Effects”. In: *Phys. Rev.* 140.4A (1965), A1133–A1138. DOI: 10.1103/PhysRev.140.A1133.

[24] Hyuk-Jae Lee, James P. Robertson, and José A. B. Fortes. “Generalized Cannon’s Algorithm for Parallel Matrix Multiplication”. In: *Proc. ICS ’97, Intl. Conf. Supercomputing, July 7–11, 1997, Vienna, Austria*. ACM Press, 1997, pp. 44–51. DOI: 10.1145/263580.263591.

[25] A. Marek et al. “The ELPA Library: Scalable Parallel Eigenvalue Solutions for Electronic Structure Theory and Computational Science”. In: *J. Phys.: Condens. Matter* 26.21 (May 2014), p. 213201. DOI: 10.1088/0953-8984/26/21/213201.

[26] Jack Poulson, Robert A. van de Geijn, and Jeffrey Bennighof. *Parallel Algorithms for Reducing the Generalized Hermitian-Definite Eigenvalue Problem*. Technical Report TR-11-05. FLAME Working Note #56. The University of Texas at Austin, Department of Computer Science, 2011.

[27] Jack Poulson et al. “Elemental: A New Framework for Distributed Memory Dense Matrix Computations”. In: *ACM Trans. Math. Software* 39.2 (Feb. 2013), 13:1–13:24. DOI: 10.1145/2427023.2427030.

[28] Michael Rippl. “Parallel Algorithms for the Solution of Banded Symmetric Generalized Eigenvalue Problems”. PhD thesis. Technische Universität München, 2020.

[29] Michael Rippl, Bruno Lang, and Thomas Huckle. “Parallel Eigenvalue Computation for Banded Generalized Eigenvalue Problems”. In: *Parallel Comput.* 88 (Oct. 2019), p. 102542. DOI: 10.1016/j.parco.2019.07.002.

[30] C. C. J. Roothaan. “Modern Developments in Molecular Orbital Theory”. In: *Rev. Mod. Phys.* 23.2 (1951), pp. 69–89. DOI: 10.1103/RevModPhys.23.69.

[31] Robert Schreiber and Charles Van Loan. “A Storage-Efficient WY Representation for Products of Householder Transformations”. In: *SIAM J. Sci. Stat. Comput.* 10.1 (Jan. 1989), pp. 53–57.

[32] Edgar Solomonik and James Demmel. “Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms”. In: *Euro-Par 2011 Parallel Processing. 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29–September 2, 2011, Proceedings, Part II*. Ed. by Emmanuel Jeannot, Raymond Namyst, and Jean Roman. Vol. 6853. LNCS. Berlin, Heidelberg: Springer, 2011, pp. 90–109. DOI: 10.1007/978-3-642-23397-5_10.