Technische Universität München
Lehrstuhl für Wissenschaftliches Rechnen

Parallel Algorithms for the Solution of Banded Symmetric Generalized Eigenvalue Problems

Michael Rippl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des Akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende(r):		Prof. Dr. rer. nat. Helmut Seidl
Prüfer der Dissertation:	1.	Prof. Dr. rer. nat. Thomas Huckle
	2.	Prof. Dr. rer. nat. Bruno Lang

Die Dissertation wurde am 06.08.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 30.11.2020 angenommen.

Acknowledgements

At this point it is a good idea to pause and recapitulate the last few years in which I was working on the topics of this thesis. First, I want to thank Thomas Huckle for his open door, his patience and his support. This helped me a lot and let me grow. In the same way, I want to thank Bruno Lang for his ideas and his support, especially with the twisted Crawford approach, but also for sharing some tricks in numerical linear algebra with me. Thanks also to Pavel Kůs for the evaluation of the first approach and for encouraging me at the second attempt. Great thanks also to Valeriy Manin for providing his bandreduction code to me.

Developing and writing such a thesis is work. But work is not only work when sharing the pain, the success, the problems and their solutions. Hence, I want to thank Benjamin Rüth for sharing the office, the pain and success of the past years with me. Of course, thanks also to the rest of the SCCS chair for the joyful time. Special thanks to my sister Christine for patiently proofreading this thesis and, finally, I want to thank my family and my friends for their support in this partly stressful times.

Abstract

The solution of eigenvalue problems is one of the most important problem types in numerical linear algebra. Many eigenvalue problems are generalized eigenvalue problems which are transformed to standard eigenvalue problems and solved as such. If the matrices of the generalized eigenvalue problem have banded structure, this procedure leads to a huge overhead when using modern two-step solver for computing the resulting standard eigenvalue problem. The reason is the loss of banded structure in the standard eigenvalue problem matrix.
A similar loss of the banded structure appears when solving banded generalized singular value decompositions by the transformation to standard singular value decompositions.

This thesis describes serial and parallel algorithms for the transformation of the generalized eigenvalue problem to a standard eigenvalue problem and for the transformation of the generalized singular value decomposition to a standard singular value decomposition while maintaining the band. Maintaining the band allows to further exploit the banded structure, e.g. by directly running the second step of a two-step solver, or, if the band is too wide, utilizing a bandreduction step and then employing the second step of a two-step solver to obtain eigenvalues and eigenvectors or singular values and singular vectors. The developed algorithms are based on the Crawford algorithm.
The parallel algorithms are analyzed on a theoretical basis and in performance measurements. They have demonstrated high scalability for medium to large size matrices with thin bands. In a comparison with the ELPA two-step solver the eigenvalue implementation has demonstrated its capabilities by reducing the time to solution significantly. The singular value implementation achieves similar runtimes as the eigenvalue implementation.

Contents

1 Introduction **1**
1.1 High Performance Computing . . . 1
1.2 Generalized eigenvalue problems . . . 2
1.3 Generalized singular value problems . . . 5
1.4 Scope and structure of this work . . . 7

2 Basic Algorithms **9**
2.1 Orthogonal transformations . . . 9
2.1.1 Householder transformations . . . 10
2.2 Non-blocked operations . . . 11
2.2.1 Application of a Householder transformation . . . 11
2.2.2 Orthogonal transformations using Householder transformations (QR, QL, RQ, LQ) . . . 12
2.3 Blocked operations . . . 14
2.3.1 Blocked application of Householder vectors . . . 14
2.3.2 Blocked orthogonal transformations using blocked Householder transformations (QR, QL, RQ, LQ) . . . 17
2.4 Factorization . . . 17

3 Algorithms **21**
3.1 Twisted Crawford algorithm . . . 21
3.1.1 Twisted factorization . . . 24
3.1.2 Eigenvectors . . . 31
3.1.3 Overall algorithm . . . 33
3.2 Crawford SVD algorithm . . . 35
3.2.1 Twisted factorization . . . 38
3.2.2 Singular vectors . . . 45
3.2.3 Overall algorithm . . . 48

4 Parallel Algorithms **51**
4.1 Parallelization strategies in Numerical Linear Algebra . . . 51
4.1.1 Distribution of data and computation . . . 51
4.1.2 Shared memory computation . . . 52
4.1.3 Pipelining . . . 52
4.1.4 Splitting in independent subtasks . . . 53
4.2 Twisted Crawford algorithm . . . 54
4.2.1 Pipelining approach . . . 54
4.2.2 Parallel execution of the upper and a lower matrix half . . . 59
4.2.3 Block and inter-block parallelization . . . 59
4.2.4 Process and data structures . . . 62

4.2.5 Parallel algorithms on the level of pairs of block rows and block columns 64
4.3 Backtransformation of eigenvectors 72
4.3.1 Pipelining approach 74
4.3.2 Backtransformation matrix approach 82
4.3.3 Summary 87
4.4 Crawford SVD algorithm 89
4.4.1 Pipelining approach 89
4.4.2 Parallel execution of the upper and a lower matrix half 93
4.4.3 Block and inter-block parallelization 97
4.4.4 Process and data structures 100
4.4.5 Parallel algorithms on the level of pairs of block rows and block columns 102
4.5 Backtransformation of singular vectors 109
4.5.1 Pipelining approach 110
4.5.2 Backtransformation matrix approach 118

5 Numerical Analysis 119
5.1 Twisted Crawford algorithm 119
5.1.1 Modeling of the speedup of the block and inter-block parallelization . . 119
5.1.2 Modeling of the speedup of the pipelining approach 123
5.1.3 Modeling of the speedup by splitting the matrix in upper and lower half 126
5.2 Twisted SVD algorithm 126
5.2.1 Modeling of the speedup of the block and inter-block parallelization . . 126
5.2.2 Modeling of the efficiency of the pipelining approach 130
5.2.3 Modeling of the speedup by having twisted matrices 132

6 Performance Analysis 133
6.1 Computational resources 133
6.2 Twisted Crawford algorithm 133
6.2.1 Block and inter-block parallelization 134
6.2.2 Pipelining approach 136
6.2.3 Parallel execution of the upper and a lower matrix half 143
6.2.4 Scaling of the overall implementation 146
6.2.5 Comparison to ELPA 149
6.3 Crawford SVD algorithm 153
6.3.1 Block and inter-block parallelization 153
6.3.2 Pipelining approach 158
6.3.3 Parallel execution of the upper and a lower matrix half 162
6.3.4 Scaling of the overall implementation 162
6.3.5 Comparison to twisted Crawford algorithm 164

7 Conclusion 167

Bibliography 169

1 Introduction

The development of numerical linear algebra code is the basis for most if not all computational programs. It takes place at the intersection of the fields high performance computing (HPC), linear algebra and numerical mathematics. Linear algebra is the basis of this kind of codes: It defines the problem statement but also the way to solution. Numerical mathematics has to be considered to achieve reasonable accuracy and to derive theoretical estimates for properties of algorithms. HPC concerns the efficient implementation of the algorithms on modern many-core systems or supercomputers. Efficient numerical linear algebra code is based on all three pillars and is usually developed in a multidisciplinary way.

1.1 High Performance Computing

Supercomputers have helped to advance the frontiers of science and engineering over the past 60 years. The increasing computing power over the years allowed to run more and more complex and detailed simulations of physical phenomena. An estimate for the advances in computing power has been described in 1965 by the famous Moore's law [47]. It predicts an exponential growth of the computing power over time. Figure 1.1 provides the development of performance of the fastest supercomputers over the past 25 years. Despite a small deviation over the past years, it shows that Moore's law has still its validity.

For some time, performance gains have been generated by increasing the clock frequency of processors. This, however, led to a disproportionate high growth in the energy consumption and the development stopped roundabout 15 years ago. Since then the complexity of processors rises more and more to increase the computing power. Vectorization was introduced, multiple fused multiply add units became available per core and the number of cores per node increased to name a few aspects. This requires more advanced programming techniques as in the times where increasing clock speed was the driving force of performance gains and single core performance increased without additional programming effort.
On a coarser level, also the complexity of supercomputers rises more and more. Accelerator cards like GPGPU can be used additionally to the classical CPUs to increase the available computing power. This trend to more complex and more heterogeneous architectures seems to continue over the next years. However, pure CPU supercomputers like SuperMUC-NG [43] are still state of the art and have currently (numbers of November 2019) a share of 50% of the fastest supercomputers.

To exploit the available hardware a thorough understanding of the processors and the overall system is necessary. As a consequence, codes for HPC systems also increase in their complexity. Algorithms tailored to certain problem classes can exploit special properties of these classes. This allows to omit unnecessary computations and will usually result in a shorter

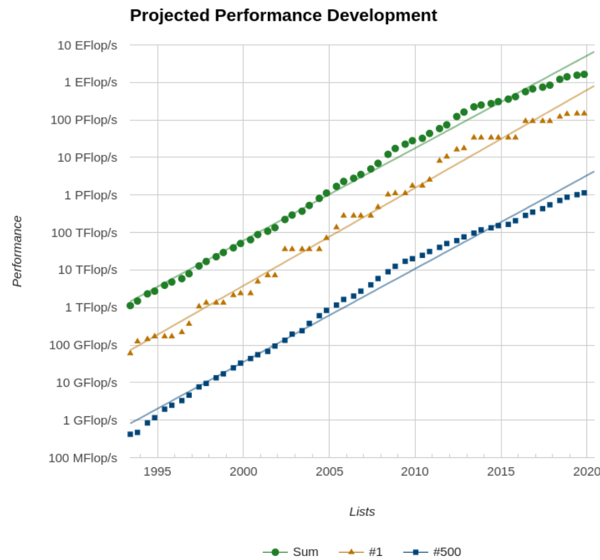


Figure 1.1: Performance development of the top 500 supercomputers over the past 25 years [62]: Sum of the 500 fastest supercomputer (Sum), world's fastest supercomputer (#1) and number 500 in the world (#500).

time to solution. The algorithms presented in this work fit into this framework. Their implementations will focus on distributed memory systems with MPI [60] as communication layer. Many tuning parameters in the algorithms allows to adapt them to the underlying hardware.

1.2 Generalized eigenvalue problems

Generalized eigenvalue problems (GEP) are defined as

$$AX = BX\Lambda \tag{1.1}$$

for given matrices A and B. X is the matrix of eigenvectors and Λ is a diagonal matrix containing the eigenvalues of the GEP on the diagonal. The order of the eigenvalues in Λ is the same as the order of eigenvectors in X.

Generalized eigenvalue problems appear in many fields in science and engineering. In mechanics, eigenfrequencies of mass-spring systems can be determined via computation of the eigenspectrum of the regarding system of mass and stiffness matrix. Another application in this field is for example the computation of the buckling load of a rod. These and many other problems are described by second order differential equations of the form $M\ddot{y}(t)+Ky(t) = b(t)$. The solution for this class of problems can be found via solving a generalized eigenvalue problem.
Another important source of generalized eigenvalue problems is computational chemistry. Schrödinger's equation $\hat{\mathrm{H}}\Phi_i = \mathrm{E}_i\Phi_i$ is the basis for describing and understanding the physics in the scale of atoms and molecules. Density functional theory [36] provides a methodology to compute these interactions by discretization and simplification steps and results in the it-

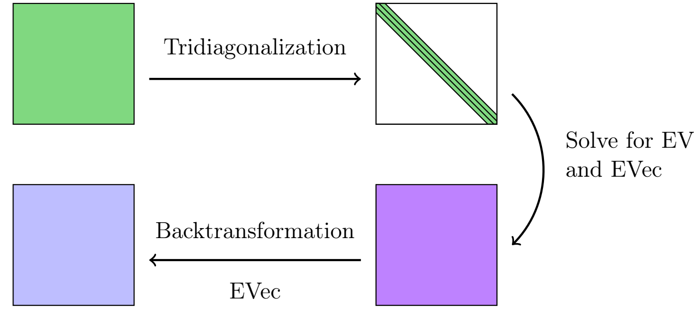


Figure 1.2: Steps of the ELPA one-step solver to compute eigenvalues (EV) and eigenvectors (EVec).

erative solution of a generalized eigenvalue problem $\hat{H}c_i = \epsilon_i S c_i$. The matrices $\hat{H}$ and S are usually banded matrices in this setup.
Sparse generalized eigenvalue problems can be found in the fields of data analysis, model order reduction and machine learning [61].

GEPs are a generalization of standard eigenvalue problems (SEP)

$$CY = Y\Lambda. \tag{1.2}$$

The solution strategy for GEPs depends on the size and the properties of the matrices A and B. In many of the named examples A and B are symmetric matrices and, additionally, B is positive definite. For large sparse matrices, iterative approaches like Jacobi Davidson variants [59], variants of the Arnoldi method [42] or LOBPCG [35] will be the methods of choice. For dense matrices the approach consists of transforming the generalized eigenvalue problem to a standard eigenvalue problem and employing algorithms for the SEP on it [56].

To tackle medium to large size standard eigenvalue problems in parallel, the use of a one-step solver (see Figure 1.2) was for quite a long time the state of the art. The symmetric matrix C is transformed to a tridiagonal matrix using orthogonal transformations. The eigenvalues of the resulting tridiagonal matrix are computed using an iterative approach like Bisection and inverse iteration [66], QR algorithm [27], Divide and conquer [22] or MRRR [24] and the requested number of eigenvectors is transformed back to obtain the eigenvectors of the matrix C. Before the ELPA library [46, 5, 6] was developed, the standard library for eigenvalue computation of dense symmetric matrices was the MKL implementation of ScaLAPACK [15] with it's one-step solver. This solver had several weaknesses which had been overcome in the ELPA one-step solver that outperformed the MKL version [46].

A new development of the ELPA project was the two-step solver [5]. The original idea was published in [11]. Instead of immediately transforming the matrix C to a tridiagonal matrix, it uses an intermediate step (see Figure 1.3). First, C is transformed to a banded matrix and the banded matrix is then transformed in a second step to a tridiagonal matrix. This approach allows to overcome the bottlenecks of the one-step solver. There, many operations are memory bound. In the two-step solver the first step uses mainly BLAS3 operations and is hence compute bound. The second step will still remain memory bound, but by choosing a small

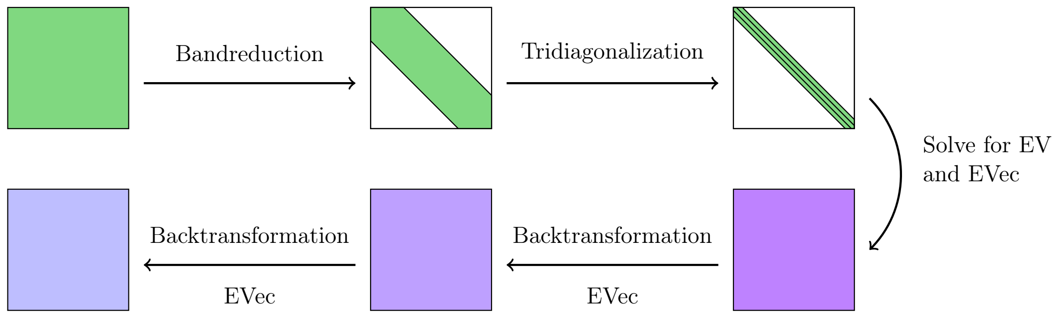


Figure 1.3: Steps of the ELPA two-step solver to compute eigenvalues and eigenvectors.

intermediate bandwidth it is relatively cheap compared to the first step. The performance gain, however, has to be paid at a different place. Instead of one step of backtransformation of the eigenvectors, the two-step solver needs two steps of backtransformation. Especially, when requesting a big fraction of the eigenspectrum, the backtransformation becomes expensive. But when only a small part of the eigenspectrum is needed (as in many applications), then the two-step solver will easily outperform the one-step solver on modern CPUs.
A slightly different approach is followed by EigenExa [34]. There, the matrix is transformed to a pentadiagonal matrix and a Divide and conquer algorithm is run on the pentadiagonal matrix to obtain the eigenvalues.

The transformation of a generalized eigenvalue problem to a standard eigenvalue problem is done by computing the Cholesky factorization of B

$$B = F^T F. \tag{1.3}$$

The inverse of the Cholesky factor is multiplied form left to A to obtain

$$C = F^{-T} A F^{-1}. \tag{1.4}$$

The original eigenvectors can be computed by $X = F^{-1} Y$.

Considering banded matrices as they appear for example in computational chemistry, this procedure has a drawback. Starting point are two banded matrices A and B. The factorization of B is still a banded matrix, but for applying the factorization to A, the inverse F^{-1} and its transpose are needed. The inversion step, however, generally destroys the banded structure and hence the matrix F^{-1} and also C end up to be full matrices. To recap, starting point were the two banded matrices A and B, which have been transformed to a full matrix C and this full matrix is then again transformed to a banded matrix.

To overcome these issues, Crawford [21] proposed an algorithm to maintain the band while applying the Cholesky factorization. This allows to omit the expensive band reduction step and, depending on the bandwidth of the matrices A and B, immediately start with the tridiagonalization (see Figure 1.4). In the case of wider bands in A and B, an additional band reduction step has to be performed.
Successive bandreduction algorithms are described in [13, 14]. A communication avoiding approach is described in [9] and has been implemented for shared memory environments. The

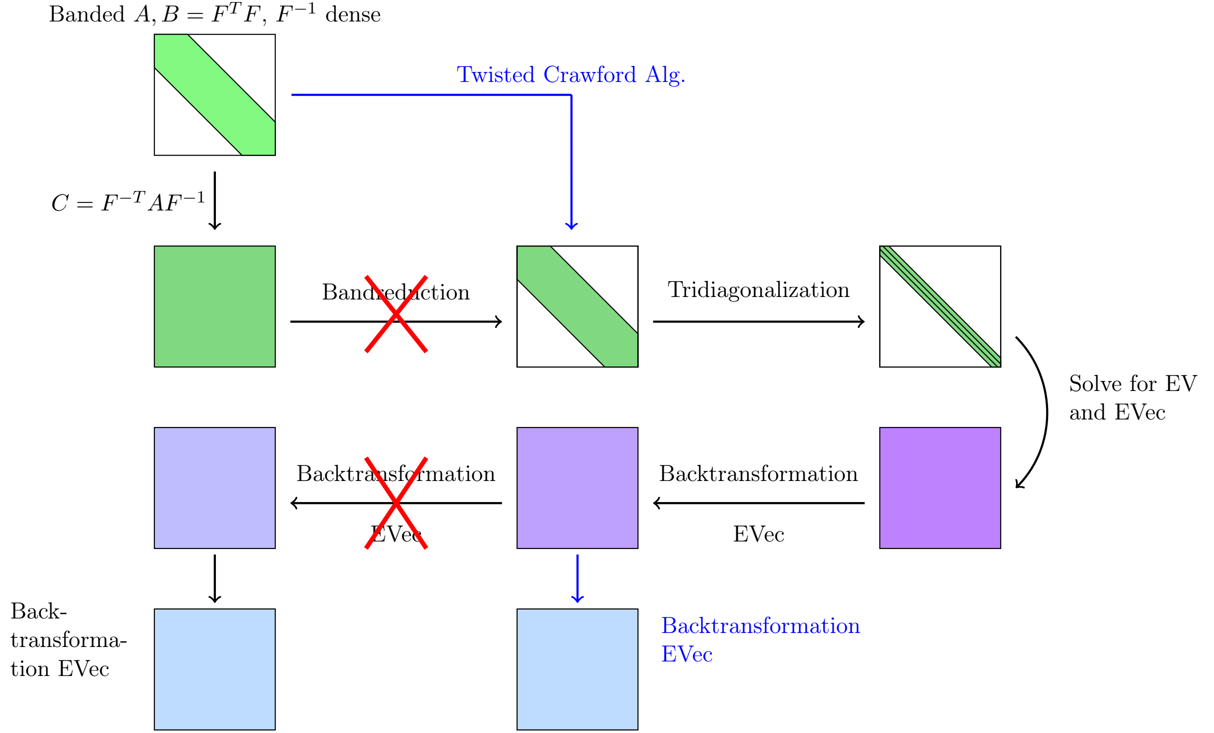


Figure 1.4: Solving a generalized eigenvalue problem with the ELPA two-step solver and the new “twisted Crawford” approach to circumvent losing the banded structure.

idea of these approaches is the generalization of the two-step solver for computing eigenvalues and eigenvectors to a multi-step solver. Two-step implementations are usually optimized for thin intermediate bandwidths and hence not suitable for wider bands. Multi-step approaches, however, can be used to bridge the gap from the wide band to the narrow band.

Lang [40] proposed some improvements for Crawford's method that allow to reduce the computational effort. In this work, the ideas of Crawford and Lang will be presented and a parallel implementation for HPC systems will be described. This is followed by an analysis of the implementation and performance measurements on current supercomputers.

1.3 Generalized singular value problems

Additionally to the decomposition in eigenvector and eigenvalue matrix, matrices can be decomposed by the singular value decomposition (SVD). The SVD of a matrix C is given by

$$C = U \Sigma V^T, \tag{1.5}$$

where U is the matrix of left singular vectors, V is the matrix of right singular vectors and Σ is the diagonal matrix containing the singular vectors of C.

The singular value decomposition is a generalization of the eigenvalue decomposition and is hence closely connected. Having the singular value decomposition of $C = U\Sigma V^T$, an eigenvalue problem can be obtained by

$$C^TC = (U\Sigma V^T)^T(U\Sigma V^T) = (V\Sigma U^T)(U\Sigma V^T) = V\Sigma^2 V^T.$$

Due to the orthogonality of the singular vectors, U drops out and V remains as the eigenvectors of C^TC. If U as eigenvector matrix shall remain, then CC^T has to be computed.

Singular value decompositions are widely used in science and engineering. They appear at computing the pseudoinverse of a matrix [50] to solve ill-conditioned linear systems or to perform principal component analysis. In image processing applications can be found at noise reduction [57] and digital watermarks [20].

Analogous to the eigenvalue problem, also for the SVD generalizations exist. The generalized singular value decomposition (GSVD) [27, 64, 49] is such a generalization and is defined as

$$\begin{aligned} G &= U \cdot {}^G\Sigma \cdot X^T \\ F &= V \cdot {}^F\Sigma \cdot X^T. \end{aligned} \tag{1.6}$$

The link to the SVD can be established for F being square and invertible by

$$\begin{aligned} W &= G \cdot F^{-1} \\ \mathrm{SVD}(W) &= \mathrm{SVD}(G \cdot F^{-1}) = U\Sigma V^T. \end{aligned} \tag{1.7}$$

The singular values of the generalized singular value problem are given by

$$\sigma_k = {}^W\sigma_k = \frac{{}^G\sigma_k}{{}^F\sigma_k}. \tag{1.8}$$

Here, ${}^G\sigma_k$ is the k-th singular value of G, ${}^F\sigma_k$ it the k-th singular value of F and ${}^W\sigma_k$ is the k-th singular vector of W, respectively.

Applications of generalized singular value problems can be found in various fields. In the following a few examples for use cases are given.
One main application is the solution of generalized eigenvalue problems. Equations of the form $G^TGx = \lambda F^TFx$, as they appear in signal processing, can be solved directly by using algorithms for generalized eigenvalue problems. To omit unnecessary roundoff errors, this problem can also be tackled by GSVD [7]. Another source for such ill-conditioned GEPs are finite element problems [54, 53]. They lead to a generalized eigenvalue problem with symmetric positive definite (spd) A and B. The problem arises from the spectral norm of A that grows rapidly with the matrix dimension. Since A and B are spd, they can be factorized by Cholesky decomposition and the problem can be solved as GSVD. For ill-conditioned matrices this approach provides higher accuracy [23].
In statistics, Tikhonov-Phillips regularization can be used for the regularization of ill-posed problems. GSVD is a way to solve these problems [7].

Iterative methods like Lanczos or subspace iteration based methods are usually used to compute the SVD of large sparse matrices [10]. For dense matrices, analogous to the standard eigenvalue problem, direct methods are used to obtain the singular values and vectors. Again, one-step or two-step solver are the methods of choice. The one-step procedure is similar to the solution of the eigenvalue problem. Instead of a tridiagonal matrix, the matrix is transformed to a bidiagonal matrix. The bidiagonal problem is solved iteratively and the singular vectors undergo a backtransformation procedure. The two-step approach was proposed in [29], recent parallel implementations are presented in [44, 25]. For the two-step approach, a banded triangular matrix is computed as substep. As in the SEP, the advantage of the two-step approach is given by the extensive use of BLAS3 operations and the resulting higher computational performance.

For the solution of generalized SVD problems, LAPACK [3] uses the approach described in [8]. It first reduces the matrices to upper triangular form and then applies an iterative procedure to obtain the singular values. The latter is based on the idea of computing the SVD of $W = G \cdot F^{-1}$ without explicitly computing W. For the GSVD, no parallel implementation is available in the MKL.
Under the constraint of initially having symmetric banded matrices, the ideas of Crawford and Lang can be ported to the generalized SVD problem [55]. Starting point is having banded triangular matrices G and F which are transformed to the banded matrix W while maintaining the banded structure. From there, the second step of a two-step solver can be used to obtain the singular values of the GSVD and by backtransformation steps, the left and right singular vectors of the original GSVD can be found. If necessary, a bandreduction step can be used to further reduce the bandwidth before using the two-step solver.

In this work, a serial algorithm and a parallel algorithm for HPC systems will be developed that transform the banded GSVD to a banded SVD while maintaining the band. The described algorithms work on the lower triangle of the band but can easily be ported to the upper triangle, if needed. The implementation of the parallel algorithm is analyzed on a theoretical basis and the findings are verified against runs on supercomputers.

1.4 Scope and structure of this work

This thesis consists of two essentially independent story lines: The algorithms for the generalized eigenvalue problem and the algorithms for the generalized singular value problem. The steps and the argumentation are widely aligned and therefore, most of the chapters consist of a section on the GEP and a section on the GSVD. This allows to read both parts mainly independently of each other and to outline the specifics in both variants. Cross references are used to depict similarities and differences of the two.
Chapter 2 summarizes some basic linear algebra properties used in this thesis and presents basic algorithms that will be used later on in a more elaborate and parallel way. Chapter 3 recapitulates the ideas of Crawford and Lang for the generalized eigenvalue problem and transfers these ideas to the generalized singular value problem. Parallel versions for HPC systems are presented in Chapter 4 for the GEP as well as the GSVD algorithms. In Chapter 5 a theoretical analysis of both algorithms is given and performance estimates are derived for

the single parallelization layer. These theoretical assumptions are evaluated by runs on HPC systems in Chapter 6. Chapter 7 summarizes the findings of this work and gives an outlook on open topics.

2 Basic Algorithms

In the following a few linear algebra concepts are recapitulated that will be used throughout the thesis. This is followed by a presentation of basic algorithms of which parallel versions will be used in the later text. Many of these algorithms have been presented in [39] and [4].

2.1 Orthogonal transformations

Definition 2.1.1 (Orthogonal matrix) *A quadratic matrix $Q \in \mathbb{R}^{n,n}$ is called orthogonal if the product with its transposed, Q^T, results in the identity matrix:*

$$Q \cdot Q^T = Q^T \cdot Q = I$$

Therefore, also $Q^{-1} = Q^T$ holds.

Definition 2.1.2 (Similarity transformation) *Two matrices $A, B \in \mathbb{R}^{n,n}$ are called similar if an invertible matrix $Q \in \mathbb{R}^{n,n}$ exists such that*

$$B = Q^T \cdot A \cdot Q.$$

Theorem 2.1.1 (Invariance of eigenvalues under orthogonal similarity transformations) *Let $B = Q^T \cdot A \cdot Q$ with $A, B \in \mathbb{R}^{n,n}$ and $Q \in \mathbb{R}^{n,n}$ be an orthogonal matrix.*
Then

$$(B - \lambda I)x = 0$$

is the eigenvalue equation for the matrix B.
Replacing B by $Q^T \cdot A \cdot Q$ in the equation leads to

$$(Q^T \cdot A \cdot Q - \lambda I)x = 0.$$

Multiplying from the left with Q leads to

$$(QQ^T \cdot A \cdot Q - \lambda Q)x = (A \cdot Q - \lambda Q)x = 0.$$

Introducing $Q^T Q = I$ allows to further simplify the equation:

$$(A \cdot Q - \lambda Q)Q^T Qx = (A \cdot QQ^T - \lambda QQ^T)Qx = (A - \lambda I)Qx = (A - \lambda I)y = 0$$

Therefore, matrix A and B have the same eigenvalues λ, but different eigenvectors. B has the eigenvectors x and matrix A has the eigenvectors $y = Qx$.

2.1.1 Householder transformations

A Householder transformation [32] is a linear transformation that uses a hyperplane through the origin as reflection plane. It dates back to 1958 and was introduced by Householder.

Definition 2.1.3 (Householder transformation) *Using* $\tau = \frac{2}{v^T v}$, *the Householder transformation matrix* H *for a Householder vector* v *is described by*

$$H = I - \tau v v^T.$$

Due to the symmetry of the outer product vv^T it is clear that H is symmetric. The orthogonality of Householder matrices can easily be shown by computing HH^T.

Besides Givens rotations [26], Householder transformations are a commonly used way to zero selected entries of a given vector x. The selection of the Householder vector v on basis of x determines the result of the transformation

$$\hat{x} = Hx. \tag{2.1}$$

It turns out that for a vector x with m entries all entries but entry j are zeroed when setting up the Householder vector as follows:

$$\begin{aligned} \hat{x}(i) &= 0 & \forall i :\ & v(i) = x(i) \\ \hat{x}(j) &= -\operatorname{sign}(x_j)\,\|x\| & :\ & v(j) = x(j) + \operatorname{sign}(x_j)\,\|x\| \end{aligned}$$

The choice for using $\operatorname{sign}(x_j)\,\|x\|$ is motivated by the danger of a loss of significance.

To ignore entries in the vector x at the transformation, for example if the first k entries should not be modified, then setting these entries in the Householder vector to zero prevents the update of them:

$$\hat{x}(k) = \quad x(k) \quad \forall k :\ v(k) = 0$$

A householder vector can be scaled arbitrarily without changing the zero pattern it produces. Common choices listed in the literature [39] are

- $v(j) = x(j) + \operatorname{sign}(x_j)\,\|x\|$
- $v(j) = 1$
- $\|v\| = 1.$

As already mentioned, the first version offers good numerical properties since all but one entry can be taken from x. Version two allows to omit storing the 1 and therefore save storage. In the case of using the Householder transformattion during a QR decomposition (or similar transformation), the entries of the Householder vector can be filled in the zeroed entries of the matrix. Version three allows to simplify τ to a fixed value of 2 and therefore also saves storage. In this work, $v(j) = 1$ is used.

Algorithm 1 Householder vector generation, the first entry of the given vector remains.

function $[v,\ \tau,\ x] \leftarrow \text{genHHvecUpper}(x)$
  $m \leftarrow \text{size}(x)$
  $\beta \leftarrow \text{sign}(x(1))\sqrt{x(1:m)^T x(1:m)}$
  $\tau \leftarrow \frac{x(1)+\beta}{\beta}$
  $v \leftarrow (1,\ \frac{x(2:m)}{x(1)+\beta})^T$
  $x \leftarrow (-\beta,\ 0(2:m))^T$
end function

Algorithm 2 Householder vector generation, the last entry of the given vector remains.

function $[v,\ \tau,\ x] \leftarrow \text{genHHvecUpper}(x)$
  $m \leftarrow \text{size}(x)$
  $\beta \leftarrow \text{sign}(x(m))\sqrt{x(1:m)^T x(1:m)}$
  $\tau \leftarrow \frac{x(m)+\beta}{\beta}$
  $v \leftarrow (\frac{x(1:m-1)}{x(m)+\beta},\ 1)^T$
  $x \leftarrow (0(1:m-1),\ -\beta))^T$
end function

Algorithms 1 and 2 compute the elementary reflector τ and the Householder vector v for a given vector x. In Algorithm 1 the generated Householder vector v will, when applied to x, zero all but the first entry of x whereas Algorithm 2 zeros all but the last entry of x. The former will be used for the orthogonal transformations QR and RQ, the latter for the transformations QL and LQ.
The computation of one Householder vector costs $3m + O(1)$ Flops in both cases.

2.2 Non-blocked operations

2.2.1 Application of a Householder transformation

Householder transformations can be applied from left or right to a $m \times n$ matrix A. This requires the multiplication of the Householder transformation matrix H from left $A \leftarrow H \cdot A$ or right $A \leftarrow A \cdot H^T$. In the former case, H is a $m \times m$ matrix, in the latter a $n \times n$ matrix. Also the Householder vector has to have size m in the left-sided case or n in the right-sided. Alternatively, Householder vectors can be applied to a matrix without explicitly setting up the transformation matrix. Algorithms 3 and 4 present the left and right-sided application of a Householder transformation which only needs a temporary array of size n (left-sided application) or m (right-sided application). The left application as well as the right-sided application of a Householder vector cost $4mn$ Flops.

Algorithm 3 Application of Householder vector from left.

function $A \leftarrow$ applyHHleft($A,\ v,\ \tau$)
  $z^T \leftarrow \tau v^T A$
  $A \leftarrow A - vz^T$
end function

Algorithm 4 Application of Householder vector from right.

function $A \leftarrow$ applyHHright($A,\ v,\ \tau$)
  $z \leftarrow \tau Av$
  $A \leftarrow A - zv^T$
end function

2.2.2 Orthogonal transformations using Householder transformations (QR, QL, RQ, LQ)

Using Householder transformations and their left-sided application, the orthogonal transformations QR, QL can be formulated. Algorithms 5 and 6 give the respective unblocked algorithms. The Householder vectors are stored in the matrix Y and the reflectors are stored in the vector τ, both in the order of their creation. The update of the trailing matrix performed by applyHHLeft only touches the remaining matrix and does not update the already created zeros.

Algorithm 5 QR factorization.

function $[A,\ Y,\ \tau] \leftarrow$ QR(A)
  $Y \leftarrow 0$
  $[m, n] \leftarrow$ size(A)
  for $j \leftarrow 1, 2, \ldots, n$ **do**
    $[Y(j:m,j),\ \tau(j),\ A(j:m,j)] \leftarrow$ genHHvecUpper($A(j:m,j)$)
    $A(j:m, j+1:n) \leftarrow$ applyHHleft($A(j:m,j+1:n),\ Y(j:m,j),\ \tau(j)$)
  end for
end function

Similarly to the left-sided transformations their right-sided pendants can be formulated. Algorithms 7 and 8 list the respective algorithms using applyHHright for the application of the transformation from right.
It has to be mentioned that LQ and RQ transformations can be mapped to the left-sided transformations QR and QL. The LQ transformation of a matrix A can be computed using a QR factorization by $[\hat{Q}, \hat{R}] = \text{QR}(A^T)$ with finally obtaining $L = \hat{R}^T$ and $Q = \hat{Q}^T$. In the same way the RQ transformation can be performed by $[\hat{Q}, \hat{L}] = \text{QL}(A^T)$. In this case, $R = \hat{L}^T$ and $Q = \hat{Q}^T$.

The left and the right-sided transformations differ regarding their progress direction. left-sided transformations process stepwise the columns of the matrix whereas the right-sided transformations process the matrix row by row. Depending on the underlying memory layout (column major, row major) the left or the right-sided transformations are beneficial due to

Algorithm 6 QL factorization.

function $[A,\ Y,\ \tau] \leftarrow \text{QL}(A)$
  $Y \leftarrow 0$
  $[m, n] \leftarrow \text{size}(A)$
  for $j_1 \leftarrow n, n-1, \ldots, 1$ **do**
    $y_1 \leftarrow n - j_1 + 1$
    $[Y(1:j_1, y_1),\ \tau(y_1),\ A(1:j_1, j_1)] \leftarrow \text{genHHvecLower}(A(1:j_1, j_1))$
    $A(1:j_1, 1:j_1-1) \leftarrow \text{applyHHleft}(A(1:j_1, 1:j_1-1),\ Y(1:j_1, y_1),\ \tau(y_1))$
  end for
end function

Algorithm 7 RQ factorization.

function $[A,\ Y,\ \tau] \leftarrow \text{RQ}(A)$
  $Y \leftarrow 0$
  $[m, n] \leftarrow \text{size}(A)$
  for $j_1 \leftarrow m, m-1, \ldots, 1$ **do**
    $y_1 \leftarrow m - j_1 + 1$
    $[Y(1:j_1, y_1),\ \tau(y_1),\ A(j_1, 1:j_1)] \leftarrow \text{genHHvecLower}(A(j_1, 1:j_1))$
    $A(1:j_1-1, 1:j_1) \leftarrow \text{applyHHright}(A(1:j_1-1, 1:j_1),\ Y(1:j_1, y_1),\ \tau(y_1))$
  end for
end function

Algorithm 8 LQ factorization.

function $[A,\ Y,\ \tau] \leftarrow \text{LQ}(A)$
  $Y \leftarrow 0$
  $[m, n] \leftarrow \text{size}(A)$
  for $j \leftarrow 1, 2, \ldots, m$ **do**
    $[Y(j:n, j),\ \tau(j),\ A(j, j:n)] \leftarrow \text{genHHvecUpper}(A(j, j:n))$
    $A(j+1:m, j:n) \leftarrow \text{applyHHright}(A(j+1:m, j:n),\ Y(j:n, j),\ \tau(j))$
  end for
end function

the contiguous memory access.

The computational costs of the QR and QL transformations (in the direct way) are in the order of $2mn^2 - \frac{2}{3}n^3 + O(mn)$ Flops when computing the decomposition of a $m \times n$ matrix with $m \geq n$. In the same way, the computational cost of a LQ and RQ transformation can be computed and it results in $2nm^2 - \frac{2}{3}m^3 + O(mn)$ Flops for a $m \times n$ matrix.

2.3 Blocked operations

All the aforementioned algorithms employ vector-vector or matrix-vector operations. On modern processors however, level 1 BLAS operations (vector-vector operations) and level 2 BLAS operations (matrix-vector operations) are memory bound. Level 3 BLAS operations (matrix-matrix operations) however, are compute bound. Memory bound operations saturate the available memory bandwidth and no further speedup on the node level is possible. Compute bound operations saturate the compute resources and by adding more computational resources the computation time can further be reduced. For memory bound operations however, no further speedup can be gained easily.
Blocking of operations is a way to replace inefficient BLAS 1 and BLAS 2 operations by BLAS 3 operations. Several of the non-blocked operations are applied together to a matrix. This often leads to having locally where the computation happens still BLAS 1 and BLAS 2 operations, but during the application to a comparably large trailing matrix, BLAS 3 operations are used. This strategy increases the number of Flops but improves performance in general.

2.3.1 Blocked application of Householder vectors

The blocked application of Householder vectors performs not the application of one transformation after another but applies several Householder vectors at the same time. Two variants are commonly used: The WY representation [12] and the compact WY (CWY) representation [58] of the Householder transformations.

Both store the Householder vectors in a matrix which is denoted as Y:

$$Y = [v_1, v_2, \dots v_n] \tag{2.2}$$

The Householder vectors v_i have to be filled up by zeros at the bottom or top to match the number of rows in Y. This comprises to not modifying the respective row or column by a Householder vector.

$$v_i = [0, \dots, 0, 1, *, \dots, *]^T \quad \text{or} \quad v_i = [*, \dots, *, 1, 0, \dots, 0]^T \tag{2.3}$$

WY and CWY representation differ slightly in the computations but are very similar. In the following, only the used compact WY representation is described. The compact WY representation has been chosen since the requirements to store the matrices are smaller.

When applying b Householder transformations $H_i = \tau_i v_i v_i^T$ in a blocked way, then the regarding transformation matrix Q can be written as

$$Q = \prod_i^b H_i. \tag{2.4}$$

The obtained Q is represented in CWY by

$$Q = I - YTY^T, \tag{2.5}$$

where Y is the matrix containing the Householder vectors and T is an upper triangular matrix of size $b \times b$.

The iterative construction of the matrix T is described in Algorithm 9. Generating T causes computational costs of $mb^2 + \frac{2}{3}b^3 + O(mb)$. During every iteration two matrix-vector products have to be computed.

Algorithm 9 Computation of T.

```
function T ← genT(Y, τ)
    T ← 0
    T(1,1) ← τ(1)
    for j ← 2,...,b do
        T(1 : j−1, j) ← τ(j)T(1 : j−1, 1 : j−1)Y(:, 1 : j−1)^T Y(:, j)
        T(j, j) ← τ(j)
    end for
end function
```

When having all Householder vectors already available, this procedure can be further optimized towards using more matrix-matrix products [51]. This procedure is described in Algorithm 10. Instead of computing the matrix-vector product of parts of the matrix Y with the next Householder vector, the inner product of Y with itself is computed. This product includes the necessary information and additionally, the symmetry of the product can be exploited. By this, only one matrix-vector product remains in the algorithm and the computational cost stay the same.

Applying the b Householder vectors to a $m \times n$ matrix A from the left as it occurs in QR and QL transformations can be done by only using memory efficient matrix multiplications as described in Algorithm 11. In the same way the application from right can be formulated for a $n \times m$ matrix A (Algorithm 12). The left and right-sided application cause computational cost of $4mnb + mb^2$ Flops for the regarding matrices.

Table 2.1 shows a comparison of the Flop count of blocked and non-blocked variants of applying a series of Householder vectors. It can be seen that the non-blocked variant causes the least Flops. The two blocked variants have more or less the same number of Flops when considering the one-time application of the transformation from left or right. When applying

Algorithm 10 Blocked computation of T.

function $T \leftarrow \text{blockedGenT}(Y,\ \tau)$
  $T \leftarrow 0$
  $S \leftarrow Y^T Y$
  $T(1,1) \leftarrow \tau(1)$
  for $j \leftarrow 2, \ldots, b$ **do**
    $T(1:j-1, j) \leftarrow \tau(j) T(1:j-1, 1:j-1) S(1:j-1, j)$
    $T(j,j) \leftarrow \tau(j)$
  end for
end function

Algorithm 11 Blocked application of Householder vectors from left.

function $A \leftarrow \text{applyCWYleft}(A,\ Y,\ T)$
  $X \leftarrow YT$
  $Z^T \leftarrow X^T A$
  $A \leftarrow AYZ^T$
end function

Algorithm 12 Blocked application of Householder vectors from right.

function $A \leftarrow \text{applyCWYright}(A,\ Y,\ T)$
  $X \leftarrow YT$
  $Z \leftarrow AX$
  $A \leftarrow A - ZY^T$
end function

	Generation of W or T	left/right application
WY	$2mb^2 + O(mb)$	$4mnb$
CWY	$mb^2 + \frac{2}{3}b^3 + O(mb)$	$4mnb + mb^2$
Non-blocked	–	$4mnb$

Table 2.1: Comparison of the Flops necessary to apply b Householder transformations to a $m \times n$ matrix in their blocked and non-blocked variants.

the transformation at a later time (e.g. as in the backtransformation step of an eigensolver), then the WY variant becomes more favorable.

However, inspecting only the Flop count is misleading since memory consumption (WY vs. CWY) considerations have as well to be taken into account as the fact that the blocked variants will mostly outperform the non-blocked version due to the better memory transfer patterns.

2.3.2 Blocked orthogonal transformations using blocked Householder transformations (QR, QL, RQ, LQ)

Blocked versions of QR, QL, RQ and LQ can be formulated using the compact WY formulation. A detailed description will be provided for the QR method, but can be transferred to the other three methods.

The matrix to run the QR on is subdivided in column tiles of size b. The algorithm steps tile-wise through the matrix starting from the left. On the columns of the current tile, a non-blocked QR decomposition is computed. The obtained Householder vectors and τ can be used to compute the matrix T. Having T and Y available, the Householder transformation can be applied in a blocked way to the rest of the columns using applyCWYleft. After this, the step is finished and the next tile can be processed where the next tile-local QR is computed. In this second unblocked QR, the first b rows can be ignored. The algorithm proceeds until all tiles have been processed. Algorithm 13 gives the respective pseudo code for the blocked QR. The computational cost of the blocked transformations sum up to $2mn^2 - \frac{5}{3}n^3 + mnb - n^2b - \frac{1}{6}nb^2 + O(mn)$ Flops.

Algorithm 13 Blocked QR factorization.

function $A \leftarrow \text{blockedQR}(A,\ b)$
  $Y \leftarrow 0$
  $[m, n] \leftarrow \text{size}(A)$
  for $j_1 \leftarrow 1,\ b+1,\ 2b+1,\ \ldots,\ n$ **do**
    $i_1 \leftarrow j_1; \quad j_2 \leftarrow j_1 + b - 1$
    $y_1 \leftarrow j_1; \quad y_2 \leftarrow j_2$
    $[A(i_1 : m, j_1 : j_2),\ Y(i_1 : m, y_1 : y_2),\ \tau(y_1 : y_2)] \leftarrow \text{QR}(A(i_1 : m,\ j_1 : j_2))$
    $T \leftarrow \text{blockedGenT}(Y(i_1 : m, y_1 : y_2),\ \tau(y_1 : y_2))$
    $A(i_1 : m, j_2 + 1 : n) \leftarrow \text{applyCWYleft}(A(i_1 : m, j_2 + 1 : n),\ Y(i_1 : m, y_1 : y_2),\ T)$
  end for
end function

2.4 Factorization

Dense generalized eigenvalue problems $Ax = \lambda Bx$ are usually solved by factorizing the symmetric positive matrix B with a Cholesky factorization and applying the factorization to A.

The factorization can be described by

$$B = LL^T, \tag{2.6}$$

where L is a lower triangular matrix.
Having a banded matrix B, the computations can be reduced and result in a basic algorithm as given in Algorithm 14.

Algorithm 14 Cholesky algorithm.

function $L \leftarrow \text{chol}(L,\ b_B)$
  $n \leftarrow \text{size}(L)$
  for $i \leftarrow 1, 2, \ldots, n$ **do**
    $L(i,i) \leftarrow \sqrt{L(i,i)}$
    $k \leftarrow \min(i + b_B,\ n)$
    $L(i+1:k, i) \leftarrow L(i+1:k, i)/L(i,i)$
    $L(i+1:k, i+1:k) \leftarrow L(i+1:k, i+1:k) - L(i+1:k, i)L(i+1:k, i)^T$
    $L(i, i+1:k) \leftarrow 0$
  end for
end function

A slightly different variant of the Cholesky factorization is defined by

$$B = L^T L. \tag{2.7}$$

This reverse Cholesky variant is computed from the end of the matrix as shown in Algorithm 15.

Algorithm 15 Reverse Cholesky algorithm.

function $L \leftarrow \text{reverseChol}(L,\ b_B)$
  $n \leftarrow \text{size}(L)$
  for $i \leftarrow n, n-1, \ldots, 1$ **do**
    $L(i,i) \leftarrow \sqrt{L(i,i)}$
    $k \leftarrow \max(i - b_B,\ 1)$
    $L(i, k:i-1) \leftarrow L(i, k:i-1)/L(i,i)$
    $L(k:i-1, k:i-1) \leftarrow L(k:i-1, k:i-1) - L(i, k:i-1)^T L(i, k:i-1)$
    $L(k:i-1, i) \leftarrow 0$
  end for
end function

If the upper triangular matrix U is requested as Cholesky factor and not the lower triangular matrix L, then Algorithm 16 can be used to obtain the factorization

$$B = U^T U. \tag{2.8}$$

Algorithm 16 Reverse Cholesky algorithm for upper triangular Cholesky factor.

function $U \leftarrow \text{reverseCholUpper}(U,\ b_B)$
  $n \leftarrow \text{size}(U)$
  for $i \leftarrow 1, 2, \ldots, n$ **do**
    $U(i,i) \leftarrow \sqrt{U(i,i)}$
    $k \leftarrow \min(i + b_B,\ n)$
    $U(i+1:k, i) \leftarrow U(i+1:k, i)/U(i,i)$
    $U(i+1:k, i+1:k) \leftarrow U(i+1:k, i+1:k) - U(i+1:k, i)^T U(i+1:k, i)$
    $U(i, i+1:k) \leftarrow 0$
  end for
end function

Any of the Cholesky variants causes computational cost of $nb_B^2 - \frac{2}{3}b_B^3 + O(nb_B)$ Flops for a $n \times n$ matrix B with bandwidth b_B.

A twisted factorization [65] is a modified version of the Cholesky factorization. The matrix is separated in an upper and a lower matrix half at a so called twist index p. In the lower matrix half, the matrix is lower triangular, in the upper matrix half it is upper triangular. The entries in the lower part are identical to the reverse Cholesky factorization (Algorithm 15). In the upper matrix half, Algorithm 16 can be used to compute the values. A blocked algorithm for the twisted factorization of a banded matrix is given in Algorithm 17.

Algorithm 17 Twisted factorization.

function $S \leftarrow \text{twistedFact}(S,\ n,\ b_B,\ n_b,\ p)$
  $P \leftarrow p/n_b$
  for $i \leftarrow N, N-1, \ldots, P+1$ **do**
    $i_1 \leftarrow (i-1) \cdot n_b + 1;\quad i_2 \leftarrow \min(i_1 + n_b - 1,\ n);\quad n_{b0} \leftarrow i_2 - i_1 + 1$
    $S(i_1 : i_2, i_1 : i_2) \leftarrow \text{reverseChol}(S(i_1 : i_2, i_1 : i_2),\ b_B)$
    if $(i_1 > 1)$ **then**
      $i_0 \leftarrow \max(i_1 - b_B,\ 1)$
      $S(i_1 : i_2, i_0 : i_1 - 1) \leftarrow S(i_1 : i_2, i_1 : i_2)^{-T} S(i_1 : i_2, i_0 : i_1 - 1)$
      $S(i_0 : i_1 - 1, i_0 : i_1 - 1) \leftarrow S(i_0 : i_1 - 1, i_0 : i_1 - 1)$
        $- S(i_1 : i_2, i_0 : i_1 - 1)^T S(i_1 : i_2, i_0 : i_1 - 1)$
      $S(i_0 : i_1 - 1, i_1 : i_2) \leftarrow 0$
    end if
  end for
  for $i \leftarrow 1, 2, \ldots, P$ **do**
    $i_1 \leftarrow (i-1) \cdot n_b + 1;\quad i_2 \leftarrow \min(i_1 + n_b - 1,\ p);\quad n_{b0} \leftarrow i_2 - i_1 + 1$
    $S(i_1 : i_2, i_1 : i_2) \leftarrow \text{reverseCholUpper}(S(i_1 : i_2, i_1 : i_2),\ b_B)$
    if $(i_2 < p)$ **then**
      $i_0 \leftarrow \max(i_2 + b_B,\ p)$
      $S(i_1 : i_2, i_2 + 1 : i_0) \leftarrow S(i_1 : i_2, i_1 : i_2)^{-T} S(i_1 : i_2, i_2 + 1 : i_0)$
      $S(i_2 + 1 : i_0, i_2 + 1 : i_0) \leftarrow S(i_2 + 1 : i_0, i_2 + 1 : i_0)$
        $- S(i_1 : i_2, i_2 + 1 : i_0)^T S(i_1 : i_2, i_2 + 1 : i_0)$
      $S(i_2 + 1 : i_0, i_1 : i_2) \leftarrow 0$
    end if
  end for
end function

3 Algorithms

3.1 Twisted Crawford algorithm

This section focuses on the transformation of generalized eigenvalue problems with symmetric band matrices A and B, B being additionally positive definite, to standard eigenvalue problems. The bandwidth of matrix A is denoted as b_A, the bandwidth of matrix B is denoted as b_B.
The standard way of solving $AX = BX\Lambda$ for the eigenvectors X and the eigenvalues Λ is to factorize $B = F^T F$ with a Cholesky factorization and transfer it to a standard eigenvalue problem:

$$\begin{aligned} AX &= BX\Lambda = F^T FX\Lambda \\ \Leftrightarrow F^{-T} AF^{-1} FX &= FX\Lambda \\ \Leftrightarrow F^{-T} AF^{-1} Y &= Y\Lambda \\ \Leftrightarrow CY &= Y\Lambda \end{aligned} \tag{3.1}$$

The standard eigenvalue problem can then be solved with the procedures described in [4].
A drawback of this approach is that the banded structure of A and B cannot be exploited. Like B, the matrix F is still banded. However, the inverse of it is applied to A, and F^{-1} and F^{-T} are in general full matrices. Therefore C will be a full matrix, too.

To maintain the band, Crawford [21] proposed an algorithm to apply F stepwise to A while removing the occurring fill-in outside the band immediately after every step. Lang [40] refined this procedure by reducing the restrictions for the bandwidth to $b_A \geq b_B$, offering more freedom for parameter tuning and by introducing features for saving computational work. In the following, Crawford's original algorithm will be described and the extensions of Lang will be introduced.

As mentioned before, the factorization F is applied not as a whole but stepwise to A. For this, F itself is factorized as a product of $N = \lceil \frac{n}{n_b} \rceil$ partial matrices F_k:

$$F = F_1 \cdot F_2 \cdots F_N \tag{3.2}$$

Each of this F_k is an identity matrix besides the rows $(k-1)n_b + 1 : kn_b$ (see Figure 3.1). Within these rows it has the same content as the matrix F. The inverse of F can also be expressed as a product of factors:

$$F^{-1} = F_N^{-1} \cdot F_{N-1}^{-1} \cdots F_1^{-1} \tag{3.3}$$

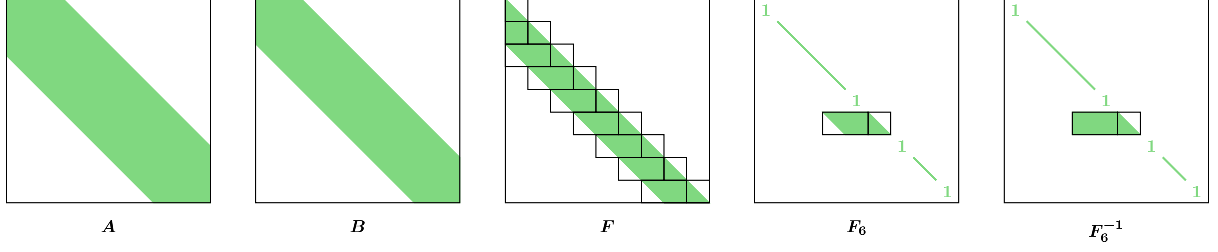


Figure 3.1: Matrices A (first picture) and $B = F^T F$ (second picture) with different bandwidth, the block structure of the Cholesky factorization F (third picture), one of its partial factors following Equation (3.2), F_6 (forth picture) and the inverse of it, F_6^{-1} (fifth picture).

Rewriting $F^{-T} A F^{-1} Y = Y\Lambda$ of Equation (3.1) using Equation (3.3) leads to

$$F_1^{-T} \cdot F_2^{-T} \cdots F_N^{-T} \cdot A \cdot F_N^{-1} \cdot F_{N-1}^{-1} \cdots F_1^{-1} Y = Y\Lambda. \tag{3.4}$$

The inverse of a F_k can be found by inverting the diagonal block and by applying the inverted diagonal block to the block left of the diagonal. All other parts of the matrix stay unchanged. This means, that the block structure of F_k^{-1} is the same as for F_k as shown in Figure 3.1, picture four and five.

In a more formal way, with $F_{k,k}$ describing the non-identity diagonal block in F_k and $F_{k,k-1}$ describing the block left of it, the following expressions can be obtained for the inverse blocks:

$$D_k = F_{k,k}^{-1} \tag{3.5}$$

$$E_k = -D_k \cdot F_{k,k-1} \tag{3.6}$$

D_k will be used as abbreviation for the diagonal block of the inverse F_k^{-1}, E_k as abbreviation for the block left of the diagonal block.

In the following, the application of F_k^{-T} from left and F_k^{-1} from right to a symmetric banded matrix M is analyzed. This will be referred to as "inversion step" k. M takes over the role of the matrix A that has already undergone an arbitrary number of inversion steps which have been succeeded by restoring the band of the matrix as described in the following.
The right-sided application multiplies the columns $(k-1)n_b + 1 : kn_b$ of M with E_k and adds the result to the columns $(k-1)n_b - b_B + 1 : (k-1)n_b$ of M. Afterwards, the columns $(k-1)n_b + 1 : kn_b$ of M are updated by themselves multiplied with D_k. This generates triangular fill-in outside the band (the so called "bulge") if $n_b + b_B - 1 > b_A$. Otherwise the updated columns and rows are fully located inside the band. The generated fill-in is located in the rows $(k-1)n_b + b_A - b_B + 2 : kn_b + b_A$ and the columns $(k-1)n_b - b_B + 1 : kn_b - 1$, hence in M's lower triangle.
The left-sided application multiplies the rows $(k-1)n_b + 1 : kn_b$ with E_k^T and adds the result to the rows $(k-1)n_b - b_B + 1 : (k-1)n_b$. The rows $(k-1)n_b + 1 : kn_b$ of M are then multiplied by D_k^T. The triangular fill-in (if $n_b + b_B - 1 > b_A$) generated by the left-sided application is located in the rows $(k-1)n_b - b_B + 1 : kn_b - 1$ and the columns $(k-1)n_b + b_A - b_B + 2 : kn_b + b_A$, hence in the upper triangle of M.

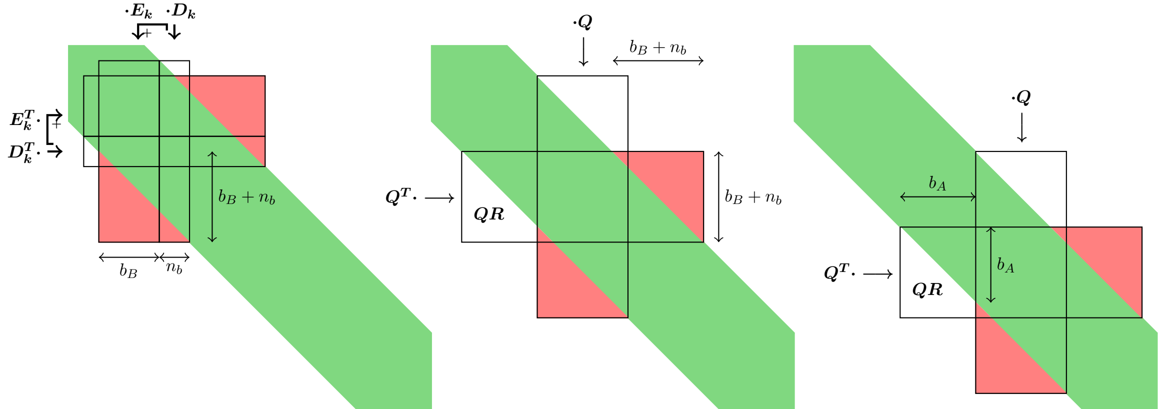


Figure 3.2: (left)The application of inversion step k updates the entries within the rectangles (see main text for details). Marked in red is the bulge of newly created non-zeros outside the band. (middle) First chasing step using QR decomposition and its symmetric application. The newly created bulge has again a size of $b_B + n_b$. (right) Second chasing step for pushing the bulge by b_A columns to the right and bottom.

Applying now the next inversion step F_{k-1}^{-1} would lead to a growth of the fill-in outside the band by n_b rows and columns. Consequently, when applying more and more inversion steps, this procedure will lead to a full matrix M. Hence, the fill-in has to be removed immediately after it is generated.

To do so, a two-sided orthogonal transformation is applied to M. Due to symmetry, the orthogonal factor Q can be computed by a QR decomposition of the lower triangular bulge or by a LQ decomposition of the upper triangular bulge. To zero the fill-in, the QR has to be run on the rows $(k-1)n_b+b_A-b_B+1 : kn_b+b_A$ and the columns $(k-1)n_b-b_B+1 : kn_b$, the LQ on the rows $(k-1)n_b-b_B+1 : kn_b$ and the columns $(k-1)n_b+b_A-b_B+1 : kn_b+b_A$. In this work the QR decomposition is used since it fits better to the memory layout of Fortran. The two sided transformation removes the two bulges generated by the application of the inversion step but introduces two new bulges. The new bulges appear b_A rows and columns further down the matrix. Again, a QR decomposition of the lower triangular bulge is computed (QR on the rows $(k-1)n_b+2b_A-b_B+1 : kn_b+2b_A$ and columns $(k-1)n_b+b_A-b_B+1 : kn_b+b_A$) and the obtained Q is applied two-sided to the matrix. By that the fill-in is shifted ("chased") another b_A rows and columns towards the end of the matrix. This procedure is repeated until the fill-in drops out of the matrix and the band is fully restored. Then the next inversion step can be applied. Figure 3.2 illustrates the first steps of the procedure and shows the sizes of the different parts.

This procedure can be formalized to

$$\widehat{M} = Q_k^{(\nu_k)T} \cdots Q_k^{(2)T} \cdot Q_k^{(1)T} \cdot F_k^{-T} \cdot M \cdot F_k^{-1} \cdot Q_k^{(1)} \cdot Q_k^{(2)} \cdot \cdots \cdot Q_k^{(\nu_k)}. \tag{3.7}$$

M is a symmetric banded matrix to which inversion step F_k^{-1} is applied to. The $Q_k^{(i)}$ are the

orthogonal factors that are applied symmetrically to evict the fill-in from the matrix. The index k corresponds to the number of the inversion step, i corresponds to the order of the orthogonal transformations. $i = 1$ is the first orthogonal transformation taking place after applying F_k^{-1}, $i = 2$ is the next transformation and $i = \nu_k$ is the last orthogonal transformation that evicts the fill-in from the end of the matrix. To point out the difference to M, the resulting matrix is called $\widehat{M}$.
The amount of necessary transformations can be explicitly computed by

$$\nu_k = \lfloor \frac{n - (k-1)n_b + b_B - 1}{b_A} \rfloor. \tag{3.8}$$

ν_k can be obtained when considering the top most row of the fill-in generated by inversion step k in the lower triangle, $(k-1)n_b + b_A - b_B + 2$. Since the fill-in moves every stage by b_A rows to the bottom, the top most row of the bulge after i bulge chasing stages is $(k-1)n_b + (i+1)b_A - b_B + 2$. The formula for ν_k can be obtained when taking into account that the bulge drops out of the matrix if the top most row of it is greater than the size of the matrix n.

The $Q_k^{(i)}$ obtained by QR are orthogonal matrices and hence $Q_k^{(i)T} = Q_k^{(i)-1}$. Consequently, with Z being a symmetric matrix, $\widehat{Z} = Q_k^{(i)T} \cdot Z \cdot Q_k^{(i)} = Q_k^{(i)-1} \cdot Z \cdot Q_k^{(i)}$ is a similarity transformation and therefore the resulting matrix $\widehat{Z}$ has the same eigenvalues as Z.
Applying this to Equation (3.7) allows to conclude that $\widehat{M}$ has the same eigenvalues as $F_k^{-T} \cdot M \cdot F_k^{-1}$.

Using

$$\widetilde{F}_k^{-1} = F_k^{-1} \cdot Q_k^{(1)} \cdot Q_k^{(2)} \cdots Q_k^{(\nu_k)}, \tag{3.9}$$

the sequence of applying the inversion steps while removing the occurring fill-in can be written as

$$\begin{aligned}\widetilde{C} &= \widetilde{F}_1^{-T} \cdot \widetilde{F}_2^{-T} \cdots \widetilde{F}_N^{-T} \cdot A \cdot \widetilde{F}_N^{-1} \cdot \widetilde{F}_{N-1}^{-1} \cdots \widetilde{F}_1^{-1} \\ &= \widetilde{F}^{-T} \cdot A \cdot \widetilde{F}^{-1}.\end{aligned} \tag{3.10}$$

The resulting standard eigenproblem

$$\widetilde{C}\widetilde{Y} = \widetilde{Y}\Lambda \tag{3.11}$$

is similar to the eigenproblem $CY = Y\Lambda$ derived in Equation (3.1) from the generalized eigenproblem, the eigenvectors of the resulting problem are given by

$$\widetilde{Y} = \widetilde{F}X. \tag{3.12}$$

3.1.1 Twisted factorization

So far, a Cholesky factorization has been used to factorize matrix B. When applying it stepwise, the fill-in is removed by two-sided QR transformations towards the lower end of the matrix. This means, that when the fill-in is generated at the very top of the matrix, a long chain of QR steps has to be used to chase the bulge out of the matrix. To shorten the number

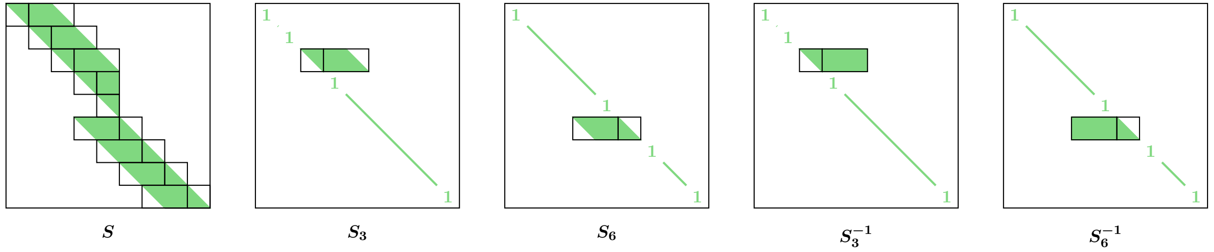


Figure 3.3: Block structure of a twisted factorization S (first picture), one of its partial factors in the upper matrix half S_3 (second picture) and one of the partial factors in the lower matrix half S_6 (third picture). The inverse of the factors S_3^{-1} and S_6^{-1} are shown in pictures four and five.

of chasing steps, Lang proposed to use a so called twisted factorization [65] (also known as split factorization, e.g. in LAPACK).
Figure 3.3 gives an illustration of a twisted Factorization S. Already shown here is the block structure when the twist index p is chosen as a multiple of n_b.

Similar to Equation (3.2), the matrix can be factorized. Using P as abbreviation for the block having the twist position p at its end,

$$S = S_P \cdot S_{P-1} \cdots S_1 \cdot S_{P+1} \cdot S_{P+2} \cdots S_N \tag{3.13}$$

gives the factorization of S in partial factors S_k. The factors have the same structure as when using a Cholesky factorization. Only in the upper matrix half the block structure is mirrored: the diagonal block is an upper triangular sub-matrix and the off-diagonal block is located right of the diagonal block. The numerical ordering is changed for a twisted factorization to an order where first the upper matrix half factors appear in reverse order and then the lower matrix half factors appear in normal ordering.

The inverse of S is given by

$$S^{-1} = S_N^{-1} \cdot S_{N-1}^{-1} \cdots S_{P+1}^{-1} \cdot S_1^{-1} \cdot S_2^{-1} \cdots S_P^{-1}. \tag{3.14}$$

Before, the application of an inversion step to the matrix was described for the lower matrix half case. Similarly, it can be formulated for the upper matrix half. Analogously to before, $E_k = -D_k \cdot S_{k,k+1}$ denotes the inverse of the off-diagonal block in S_k, $D_k = S_{k,k}^{-1}$ the inverse of the diagonal block.

Applying S_k^{-1} with $k \leq P$ from right to a banded symmetric matrix M starts with multiplying the columns $(k-1)n_b + 1 : kn_b$ of M by E_k and adding the result to the columns $kn_b + 1 : kn_b + b_B$. Afterwards, M's columns $(k-1)n_b + 1 : kn_b$ are multiplied by D_k. This introduces fill-in outside the band in the columns $(k-1)n_b + 2 : kn_b + b_B$ in the rows $(k-1)n_b - b_A + 1 : kn_b - b_A + b_B - 1$.
The application of S_k^{-T} from left to M works similarly by multiplying the rows $(k-1)n_b + 1 : kn_b$ with E_k^T and adding the outcome to the rows $kn_b + 1 : kn_b + b_B$, followed by a multiplication of the rows $(k-1)n_b + 1 : kn_b$ with D_k^T. The generated fill-in is located in the columns

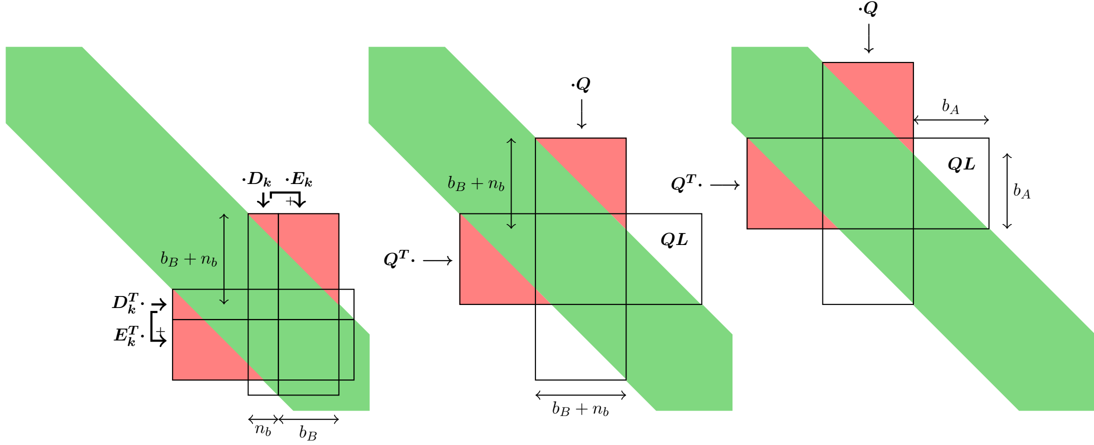


Figure 3.4: Applying an inversion step in the upper matrix half: (left)The application of the inversion step k updates the entries within the rectangles (see main text for details). Marked in red is the bulge of newly created non-zeros outside the band. (middle) First chasing step using QL decomposition and its symmetric application. The newly created bulge has again a size of $b_B + n_b$. (right) Second chasing step for pushing the bulge by b_A columns to the left and top.

$(k-1)n_b - b_A + 1 : kn_b - b_A + b_B - 1$ of the rows $(k-1)n_b + 2 : kn_b + b_B$.
This fill-in can be chased towards the top left end of the matrix using QL or RQ decomposition, depending on whether working in the upper or the lower triangle of the symmetric matrix. Exemplarily, a QL is performed on the columns $(k-1)n_b + 1 : kn_b + b_B$ of the rows $(k-1)n_b - b_A + 1 : kn_b - b_A + b_B$. The obtained orthogonal Q is applied from left and right to the matrix and by that, the bulge is shifted by b_A rows and columns towards the top end. In Figure 3.4 the application of S_k^{-1} and two of the following bulge chasing steps using QL decomposition are illustrated.

Not considered in the above description of applying the inversion steps is the special structure of S_k^{-1} in the upper matrix half: There, the band ends at column p. Hence, the updated columns when applying an inversion step never exceed p. This leads to the key point of the twisted factorization, the decoupling of the upper and the lower matrix half. Applying lower matrix half inversion steps does not cause fill-in in the upper matrix half and vice versa. Hence, fill-in generated by the lower matrix half can be chased towards the lower matrix end and fill-in generated by the upper matrix half can be chased towards the upper matrix end.

The number of bulge chasing steps in the upper matrix half can be computed analogously to the steps in the lower matrix half. Here, the lowest row of the bulge in the upper triangle can be used to obtain

$$\nu_k = \lfloor \frac{kn_b + b_B - 1}{b_A} \rfloor. \tag{3.15}$$

With

$$\widetilde{S}_k^{-1} = S_k^{-1} \cdot Q_k^{(1)} \cdot Q_k^{(2)} \cdots Q_k^{(\nu_k)}, \tag{3.16}$$

the sequence of applying the factors and removing the occurring fill-in can be rewritten as

$$\begin{aligned}\widetilde{C} &= \widetilde{S}_P^{-T} \cdots \widetilde{S}_1^{-T} \cdot \widetilde{S}_{P+1}^{-T} \cdots \widetilde{S}_N^{-T} \cdot A \cdot \widetilde{S}_N^{-1} \cdots \widetilde{S}_{P+1}^{-1} \cdot \widetilde{S}_1^{-1} \cdots \widetilde{S}_P^{-1} \\ &= \widetilde{S}^{-T} \cdot A \cdot \widetilde{S}^{-1}\end{aligned} \tag{3.17}$$

and the resulting standard eigenproblem $\widetilde{C}\widetilde{Y} = \widetilde{Y}\Lambda$ is again similar to the eigenproblem $CY = Y\Lambda$.

Following Equation (3.17), for the computation of $\widetilde{C}$, first Algorithm 18 and then Algorithm 19 have to be run. For simplicity, the possible shrinking of the last bulge before the fill-in drops out of the matrix is not described in the algorithms.

Algorithm 18 Crawford algorithm (serial) for the lower matrix half. B is factorized as $B = S^T S$ (with Cholesky or with twisted factorization). When not using a twisted factorization for B, then $P = 0$.

for $k \leftarrow N, N-1, \ldots, P+1$ **do**
  ▷ "inversion step" k (two-sided application of S_k^{-1})
  $i_1 \leftarrow (k-1)n_b + 1$; $i_2 \leftarrow i_1 + n_b - 1$
  $j_1 \leftarrow i_1 - b_B$; $j_2 \leftarrow i_1 - 1$
  $D_k \leftarrow S(i_1 : i_2, i_1 : i_2)^{-1}$
  if $(j_1 \le j_2)$ **then**
    $E_k \leftarrow -D_k \cdot S(i_1 : i_2, j_1 : j_2)$
    $A(i_1 - b_A : i_2 + b_A, j_1 : j_2) \leftarrow A(i_1 - b_A : i_2 + b_A, j_1 : j_2) + A(i_1 - b_A : i_2 + b_A, i_1 : i_2) \cdot E_k$
  end if
  $A(i_1 - b_A : i_2 + b_A, i_1 : i_2) \leftarrow A(i_1 - b_A : i_2 + b_A, i_1 : i_2) \cdot D_k$
  if $(j_1 \le j_2)$ **then**
    $A(j_1 : j_2, i_1 - b_A : i_2 + b_A) \leftarrow A(j_1 : j_2, i_1 - b_A : i_2 + b_A) + E_k^T \cdot A(i_1 : i_2, i_1 - b_A : i_2 + b_A)$
  end if
  $A(i_1 : i_2, i_1 - b_A : i_2 + b_A) \leftarrow D_k^T \cdot A(i_1 : i_2, i_1 - b_A : i_2 + b_A)$

  ▷ "bulge chasing"
  $i \leftarrow 1$
  $i_1^b \leftarrow i_1 + b_A - b_B$; $i_2^b \leftarrow i_2 + b_A$
  $j_1^b \leftarrow j_1$; $j_2^b \leftarrow i_2$
  while $j_1^b < n - b_A$ **do**
    $[Q_k^{(i)},\ R_k^{(i)}] \leftarrow$ QR decomposition of $A(i_1^b : i_2^b, j_1^b : j_2^b)$
    $A(i_1^b : i_2^b, j_1^b : j_2^b + b_A) \leftarrow Q_k^{(i)T} \cdot A(i_1^b : i_2^b, j_1^b : j_2^b + b_A)$
    $A(j_1^b : j_2^b + b_A, i_1^b : i_2^b) \leftarrow A(j_1^b : j_2^b + b_A, i_1^b : i_2^b) \cdot Q_k^{(i)}$
    $i_1^b \leftarrow i_1^b + b_A$; $i_2^b \leftarrow i_2^b + b_A$
    $j_1^b \leftarrow j_1^b + b_A$; $j_2^b \leftarrow j_2 + b_A$
    $i \leftarrow i + 1$
  end while
end for

Analyzing the bulge chasing steps necessary for maintaining the band, Equations (3.8)

Algorithm 19 Crawford algorithm (serial) for the upper matrix half. B is factorized as $B = S^T S$ (with Cholesky or with twisted factorization). When not using a twisted factorization for B, then S has its band fully in the upper triangle and $P = N$.

for $k \leftarrow 1, 2, \dots, P$ **do**
    ▷ "inversion step" k (two-sided application of S_k^{-1})
    $i_1 \leftarrow (k-1)n_b + 1; \quad i_2 \leftarrow \min(i_1 + n_b - 1, p)$
    $j_1 \leftarrow i_2 + 1; \quad j_2 \leftarrow \min(i_2 + b_B, p)$
    $D_k \leftarrow S(i_1 : i_2, i_1 : i_2)^{-1}$
    if $(j_1 \le j_2)$ **then**
        $E_k \leftarrow -D_k \cdot S(i_1 : i_2, j_1 : j_2)$
        $A(i_1 - b_A : i_2 + b_A, j_1 : j_2) \leftarrow A(i_1 - b_A : i_2 + b_A, j_1 : j_2) + A(i_1 - b_A : i_2 + b_A, i_1 : i_2) \cdot E_k$
    end if
    $A(i_1 - b_A : i_2 + b_A, i_1 : i_2) \leftarrow A(i_1 - b_A : i_2 + b_A, i_1 : i_2) \cdot D_k$
    if $(j_1 \le j_2)$ **then**
        $A(j_1 : j_2, i_1 - b_A : i_2 + b_A) \leftarrow A(j_1 : j_2, i_1 - b_A : i_2 + b_A) + E_k^T \cdot A(i_1 : i_2, i_1 - b_A : i_2 + b_A)$
    end if
    $A(i_1 : i_2, i_1 - b_A : i_2 + b_A) \leftarrow D_k^T \cdot A(i_1 : i_2, i_1 - b_A : i_2 + b_A)$

    ▷ "bulge chasing"
    $i \leftarrow 1$
    $i_1^b \leftarrow i_1 - b_A; \quad i_2^b \leftarrow i_2 - b_A + b_B$
    $j_1^b \leftarrow i_1; \quad j_2^b \leftarrow j_2$
    while $j_2^b > b_A$ **do**
        $[Q_k^{(i)},\ L_k^{(i)}] \leftarrow$ QL decomposition of $A(i_1^b : i_2^b, j_1^b : j_2^b)$
        $A(i_1^b : i_2^b, j_1^b : j_2^b + b_A) \leftarrow Q_k^{(i)T} \cdot A(i_1^b : i_2^b, j_1^b : j_2^b + b_A)$
        $A(j_1^b : j_2^b + b_A, i_1^b : i_2^b) \leftarrow A(j_1^b : j_2^b + b_A, i_1^b : i_2^b) \cdot Q_k^{(i)}$
        $i_1^b \leftarrow i_1^b - b_A; \quad i_2^b \leftarrow i_2^b - b_A$
        $j_1^b \leftarrow j_1^b - b_A; \quad j_2^b \leftarrow j_2 - b_A$
        $i \leftarrow i + 1$
    end while
end for

Factorization variant for B	Chasing steps
Cholesky factorization	$0.50N^2 + 0.5N$
Twisted factorization	$0.25N^2 + 0.5N$

Table 3.1: Number of chasing steps necessary over all inversion steps. The following simplifications are used for this computation: $n_b = b_A = b_B$, $n = Nn_b$ and $P = \frac{N}{2}$.

and (3.15) can be analyzed. To facilitate the comparison, the following simplifications are applied: $n_b = b_A = b_B$, $n = Nn_b$ and $p = \frac{N}{2}$. Table 3.1 gives the number of chasing steps over the complete algorithm. It can be seen that using a twisted factorization saves roughly 1/2 of the chasing steps.

Matrix flipping

The savings due to the use of a twisted factorization include one drawback: Extra code has to be written for the upper matrix half. This more or less doubles the number of lines of code. A possibility to overcome this issue is to flip the matrix in the upper matrix half.
"Flipping" a matrix means

$$M^{\mathrm{f}}(1:l, 1:l) = M(l:-1:1, l:-1:1).$$

The application of the inversion steps as well as the bulge chasing consists of multiplying one matrix by another. Hence, the influence of the flipping to matrix multiplication has to be inspected.
For a matrix $M = X \cdot Y$ the entries are given by

$$M = X \cdot Y = \begin{pmatrix} x_{1,1} & \cdots & x_{1,l} \\ \vdots & \ddots & \vdots \\ x_{l,1} & \cdots & x_{l,l} \end{pmatrix} \begin{pmatrix} y_{1,1} & \cdots & y_{1,l} \\ \vdots & \ddots & \vdots \\ y_{l,1} & \cdots & y_{l,l} \end{pmatrix} = \begin{pmatrix} \sum_k x_{1,k}y_{k,1} & \cdots & \sum_k x_{1,k}y_{k,l} \\ \vdots & \ddots & \vdots \\ \sum_k x_{l,k}y_{k,1} & \cdots & \sum_k x_{l,k}y_{k,l} \end{pmatrix}.$$

Considering now the flipped matrix

$$M^{\mathrm{f}} = \begin{pmatrix} \sum_k x_{l,k}y_{k,l} & \cdots & \sum_k x_{l,k}y_{k,1} \\ \vdots & \ddots & \vdots \\ \sum_k x_{1,k}y_{k,l} & \cdots & \sum_k x_{1,k}y_{k,1} \end{pmatrix} = \begin{pmatrix} x_{l,l} & \cdots & x_{l,1} \\ \vdots & \ddots & \vdots \\ x_{1,l} & \cdots & x_{1,1} \end{pmatrix} \begin{pmatrix} y_{l,l} & \cdots & y_{l,1} \\ \vdots & \ddots & \vdots \\ y_{1,l} & \cdots & y_{1,1} \end{pmatrix} = X^{\mathrm{f}} \cdot Y^{\mathrm{f}},$$

then it can be seen that flipping a product of matrices does not affect the order of the product. Only the matrix factors have to be flipped itself.
Consequently, applying an inversion step S_k, $k \leq P$, to a symmetric matrix M is equivalent to applying the flipped inversion step S_k^{f} to the flipped matrix M^{f} and flipping the result back afterwards. Figure 3.5 illustrates the impact of flipping to the application of the inversion

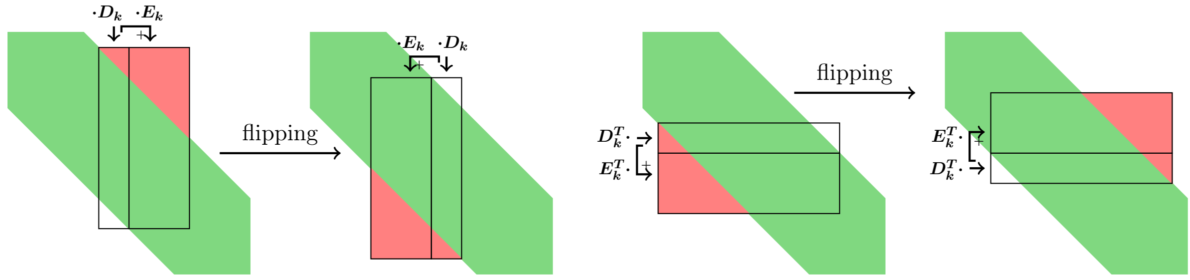


Figure 3.5: Impact of flipping to the left and right-sided application of an inversion step: The bulges change their appearance in the upper or lower triangle and D_k and E_k change their order. The resulting updated areas and the newly created bulges are the same as in the lower matrix half's application of an inversion step.

step. It can be seen that after flipping the pattern is the same as when applying an inversion step in the lower matrix half.
The equivalence holds for the generation and application of Q in the same way. Hence, by flipping the upper matrix half, the lower matrix half algorithm can be used (including small modifications) for the upper matrix half.

Algorithm 20 Flipping of a twisted banded matrix S with twist position p and bandwidth b_B.

function $M \leftarrow$ flipBandedMatrix(S, p, b_B)
  $n \leftarrow$ size(S)
  for $i \leftarrow 1, p$ **do**
    for $j \leftarrow i, \min(i + b_B, p)$ **do**
      $M(n - i + 1, n - j + 1) \leftarrow S(i, j)$
    end for
  end for
  for $i \leftarrow p + 1, n$ **do**
    for $j \leftarrow i, \max(1, i - b_B)$ **do**
      $M(n - i + 1, n - j + 1) \leftarrow S(i, j)$
    end for
  end for
end function

For dense matrices the flipping can become rather expensive, but for thin banded matrices the additional effort is acceptable. Algorithm 20 gives a simple algorithm exploiting the twisted and the banded structure while flipping. It also can be applied for A when using $p = 0$.
Necessary is not only a flipping of the matrix A but also of the matrix B, hence its factorization. For the computation of the twisted factorization the idea of flipping can also be used. The computations in the upper matrix half are just flipped versions of the lower matrix half. Small modifications have also to be considered here.

3.1.2 Eigenvectors

In the following, the formulas for the backtransformation of the eigenvectors are derived. The argumentation is rolled out for using a Cholesky factorization for B and additionally, for using a twisted factorization. After the theoretical considerations, a serial algorithm is presented for the backtransformation.

Equation (3.12) describes the translation of the eigenvectors under applying the Crawford algorithm:

$$\widetilde{Y} = \widetilde{F}X$$

Hence, the original eigenvectors are given by

$$X = \widetilde{F}^{-1}\widetilde{Y} \tag{3.18}$$

or

$$X = \widetilde{S}^{-1}\widetilde{Y} \tag{3.19}$$

when using a twisted factorization.

In Equation (3.10) the abbreviation

$$\widetilde{F}^{-1} = \widetilde{F}_N^{-1} \cdot \widetilde{F}_{N-1}^{-1} \cdots \widetilde{F}_1^{-1}$$

has been introduced. Further unwrapping gives

$$\widetilde{F}^{-1} = F_N^{-1} \cdot Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdot F_{N-1}^{-1} \cdot Q_{N-1}^{(1)} \cdots Q_{N-1}^{(\nu_{N-1})} \cdots F_1^{-1} \cdot Q_1^{(1)} \cdots Q_1^{(\nu_1)}. \tag{3.20}$$

Analogously when using a twisted factorization:

$$\begin{aligned}\widetilde{S}^{-1} = {} & S_N^{-1} \cdot Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdots S_{P+1}^{-1} \cdot Q_{P+1}^{(1)} \cdots Q_{P+1}^{(\nu_{P+1})} \\ & \cdot S_1^{-1} \cdot Q_1^{(1)} \cdots Q_1^{(\nu_1)} \cdots S_P^{-1} \cdot Q_P^{(1)} \cdots Q_P^{(\nu_P)}\end{aligned} \tag{3.21}$$

In the following, the single factors are inspected for the twisted case. The untwisted case is already covered by the lower matrix half of the twisted case. The goal is to prove the interchangeability of the inversion steps and the orthogonal matrices.

When applying $\widetilde{S}^{-1}$ stepwise from left to a matrix, then S_k^{-1}, $k > P$, updates the rows $(k-1)n_b - b_B + 1 : kn_b$. The $Q_{k+1}^{(i)}$ however update the rows $kn_b + ib_A - b_B + 1 : (k+1)n_b + ib_A$. Hence, under the constraint $b_B \leq b_A$, both can be exchanged.
Inspecting an arbitrary inversion step S_{k-l}^{-1}, $k - l > P$ and $l \geq 1$, then it updates the rows $(k-l-1)n_b - b_B + 1 : (k-l)n_b$. This updated rows are again further up in the matrix compared to $Q_{k+1}^{(i)}$. Therefore, the inversion steps of the lower matrix half can be interchanged with all orthogonal factors of the lower matrix half and the order in Equation (3.20) can be changed to

$$\widetilde{F}^{-1} = F_N^{-1} \cdot F_{N-1}^{-1} \cdots F_1^{-1} \cdot Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdot Q_{N-1}^{(1)} \cdots Q_{N-1}^{(\nu_{N-1})} \cdots Q_1^{(1)} \cdots Q_1^{(\nu_1)}. \tag{3.22}$$

Equation (3.21) can be reformulated to

$$\begin{aligned}\widetilde{S}^{-1} = S_N^{-1} \cdots S_{P+1}^{-1} \cdot Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdots Q_{P+1}^{(1)} \cdots Q_{P+1}^{(\nu_{P+1})} \\ \cdot S_1^{-1} \cdot Q_1^{(1)} \cdots Q_1^{(\nu_1)} \cdots S_P^{-1} \cdot Q_P^{(1)} \cdots Q_P^{(\nu_P)}\end{aligned}$$

in the same way.

The same can be conducted for the upper matrix half when using a twisted factorization. Inspecting an arbitrary inversion step S_{k+l}^{-1}, $k \leq k+l \leq P$, it can be seen that it updates the rows $(k+l-1)n_b + 1 : (k+l)n_b + b_B$. The upper matrix half's $Q_k^{(i)}$, however, update the rows $(k-1)n_b - ib_A + 1 : kn_b - ib_A + b_B$. Assuming again $b_A \geq b_B$, the first row updated by applying S_{k+l}^{-1} is further down the matrix than the last row updated by $Q_k^{(i)}$. Hence, both can be exchanged.
Finally, to interchange the upper matrix half's inversion steps S_{k-l}^{-1}, $k > P$, $k - l \leq P$, with the lower matrix half's $Q_k^{(i)}$, the updated rows of both have to be analyzed. The $Q_k^{(i)}$ of the lower matrix half updating the left most columns and the top most rows, $Q_{P+1}^{(1)}$, updates the rows $Pn_b + b_A - b_B + 1 : (P+1)n_b + b_A$. The last inversion step P updates the rows $(P-1)n_b + 1 : Pn_b$. The latter is due to the form of the twisted factorization, where block row P does not contain an off-diagonal block. Comparing the last row updated by the upper matrix half's inversion steps Pn_b and the first row updated by Q of the lower matrix half, it can be seen that there will not be an overlap in the operations. Hence, also the inversion steps of the upper matrix half can be interchanged with the Q from the lower matrix half.

Therefore, Equation (3.21) can be reformulated to

$$\begin{aligned}\widetilde{S}^{-1} &= S_N^{-1} \cdots S_{P+1}^{-1} \cdot S_1^{-1} \cdot \cdots \cdot S_P^{-1} \\ &\cdot Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdots Q_{P+1}^{(1)} \cdots Q_{P+1}^{(\nu_{P+1})} \cdot Q_1^{(1)} \cdots Q_1^{(\nu_1)} \cdot \cdots \cdot Q_P^{(1)} \cdots Q_P^{(\nu_P)}. \end{aligned} \tag{3.23}$$

With the abbreviation $\widetilde{Q}$ for the product of all $Q_k^{(i)}$

$$\begin{aligned}\widetilde{Q} &= Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdot Q_{N-1}^{(1)} \cdots Q_{N-1}^{(\nu_{N-1})} \cdots \\ &\cdot Q_1^{(1)} \cdots Q_1^{(\nu_1)} \qquad \text{if not using twisted factorization}\end{aligned} \tag{3.24}$$

$$\begin{aligned}\widetilde{Q} &= Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdots Q_{P+1}^{(1)} \cdots Q_{P+1}^{(\nu_{P+1})} \\ &\cdot Q_1^{(1)} \cdots Q_1^{(\nu_1)} \cdot \cdots \cdot Q_P^{(1)} \cdots Q_P^{(\nu_P)} \qquad \text{if using twisted factorization}\end{aligned} \tag{3.25}$$

Equation (3.18) can be reformulated to

$$X = F^{-1} \cdot \widetilde{Q} \cdot \widetilde{Y} \tag{3.26}$$

and Equation (3.19), when using a twisted factorization for B, to

$$X = S^{-1} \cdot \widetilde{Q} \cdot \widetilde{Y}. \tag{3.27}$$

This gives freedom in the order of computing the eigenvectors: applying all inversion steps and orthogonal transformations in the order they have been applied to the matrix A as de-

scribed by Equations (3.18) and (3.19) or first applying all orthogonal factors and afterwards applying the inversion steps as in Equations (3.26) and (3.27).
Algorithm 21 presents the latter version. It is, however, restricted to cases with $b_A \geq b_B$. The former variant would consist of adding the application of S_k^{-1} to the two loops applying the $Q_k^{(i)}$.

Another variant is setting up an identity matrix and applying the inversion steps and the $Q_k^{(i)}$ to it when applying them to A. This has the advantage that the $Q_k^{(i)}$ don't have to be stored. Instead the large backtransformation matrix has to be stored until it is multiplied to the eigenvectors $\widetilde{Y}$.

Algorithm 21 Backtransformation of the eigenvectors of the generalized eigenvalue problem $AX = BX\Lambda$. B is factorized as $B = S^T S$ (with Cholesky or with twisted factorization). When not using a twisted factorization for B, then S has its band fully in the upper triangle ($P = N$), or S has its band fully in the lower triangle ($P = 0$).

for $k \leftarrow P, P-1, \ldots, 1$ **do**
  for $i \leftarrow \nu_k, \nu_k - 1, \ldots, 1$ **do**
    $i_1^b \leftarrow (k-1)n_b - ib_A + 1; \quad i_2^b \leftarrow kn_b - ib_A + b_B$
    $X(i_1^b : i_2^b, :) \leftarrow Q_k^{(i)} \cdot X(i_1^b : i_2^b, :)$
  end for
end for
for $k \leftarrow P+1, P+2, \ldots, N$ **do**
  for $i \leftarrow \nu_k, \nu_k - 1, \ldots, 1$ **do**
    $i_1^b \leftarrow (k-1)n_b + ib_A - b_B + 1; \quad i_2^b \leftarrow kn_b + ib_A$
    $X(i_1^b : i_2^b, :) \leftarrow Q_k^{(i)} \cdot X(i_1^b : i_2^b, :)$
  end for
end for

for $k \leftarrow P, P-1, \ldots, 1$ **do**
  $i_1 \leftarrow (k-1)n_b + 1; \quad i_2 \leftarrow kn_b$
  $j_1 \leftarrow kn_b + 1; \quad j_2 \leftarrow kn_b + b_B$
  $X(j_1 : j_2, :) \leftarrow X(j_1 : j_2, :) + E_k \cdot X(i_1 : i_2, :)$
  $X(i_1 : i_2, :) \leftarrow D_k \cdot X(i_1 : i_2, :)$
end for
for $k \leftarrow P+1, P+2, \ldots, N$ **do**
  $i_1 \leftarrow (k-1)n_b + 1; \quad i_2 \leftarrow kn_b$
  $j_1 \leftarrow (k-1)n_b - b_B + 1; \quad j_2 \leftarrow (k-1)n_b$
  $X(j_1 : j_2, :) \leftarrow X(j_1 : j_2, :) + E_k \cdot X(i_1 : i_2, :)$
  $X(i_1 : i_2, :) \leftarrow D_k \cdot X(i_1 : i_2, :)$
end for

3.1.3 Overall algorithm

Algorithm 22 provides the overall procedure of computing eigenvalues and eigenvectors. First, a factorization has to be obtained. Alternatively to the twisted factorization in Algorithm 22,

a Cholesky factorization can be used. When using a Cholesky factorization, then the call to the twisted Crawford upper matrix half algorithm becomes obsolete. From the resulting matrix $\widetilde{C}$ the eigenvalues and eigenvectors have to be obtained by an external algorithm. Two-step solver like ELPA or multi-step solver for banded matrices are the algorithm of choice for this task because they can start with the banded matrix and further reduce the band to tridiagonal form. The obtained tridiagonal matrix is solved for the eigenvalues and eigenvectors and a first backtransformation step is computed. This eigenvectors undergo a second backtransformation step originated by the twisted Crawford algorithm. If no eigenvectors are required, then these steps can be omitted.

Algorithm 22 Overall algorithm when computing eigenvalues and eigenvectors.

$S \leftarrow$ Twisted factorization (B)

$[A,\ Q_l] \leftarrow$ Crawford algorithm lower matrix half $(A,\ S)$ ▷ Algorithm 18
$[A,\ Q_u] \leftarrow$ Crawford algorithm upper matrix half $(A,\ S)$ ▷ Algorithm 19

$[X,\ \Lambda] \leftarrow$ Eigenvalue solver for banded matrix A ▷ e.g. ELPA 2nd stage of two-step solver
$X \leftarrow$ Backtransformation $(X,\ Q_l,\ Q_u)$ ▷ Algorithm 21

3.2 Crawford SVD algorithm

The ideas of Crawford and Lang can also be used for the transfer of generalized singular value problems to standard singular value problems.

In the introduction, the defintion of the generalized singular value problem

$$G = U \cdot {}^{G}\Sigma \cdot X^T$$
$$F = V \cdot {}^{F}\Sigma \cdot X^T$$

is given in Equation (1.6) and the transfer to the standard singular value problem

$$W = G \cdot F^{-1}$$
$$\mathrm{SVD}(W) = \mathrm{SVD}(G \cdot F^{-1}) = U\Sigma V^T$$

is presented in Equation (1.7). The singular values of the GSVD, as identified in Equation (1.8), are computable by the quotient $\sigma_k = \frac{{}^{G}\sigma_k}{{}^{F}\sigma_k}$.

The matrices G and F are in the following restricted to be square and banded lower triangular matrices. $G \in \mathbb{R}^{n \times n}$ has a bandwidth of b_A and $F \in \mathbb{R}^{n \times n}$ has a bandwidth of b_B. Such matrices appear when using Cholesky factorization or twisted factorization to factorize symmetric positive definite banded matrices $A = G^T G$ and $B = F^T F$. Consequently, all other matrices so far, W, U, V, X, Σ, ${}^{G}\Sigma$, ${}^{F}\Sigma$, A and B are also square matrices of size $n \times n$.

As before for the generalized eigenvalue problem, F is banded and lower triangular, but F^{-1} will in general be a full matrix. Consequently, W will be a full matrix and the original existing banded structure of G and F cannot be exploited, e.g. when using two-step solvers [29] as they are provided in PLASMA [16] or MAGMA [63].
To maintain the band when transforming a generalized singular value problem to a standard singular value problem, the ideas of Lang [40] and Crawford [21] can be followed and an algorithm derived [55]. The serial version of this algorithm is described in the remaining part of the chapter.

Again, the application of F is not done at once but stepwise by factorizing

$$F = F_1 \cdot F_2 \cdots F_N$$

as in Equation (3.2). All of the $N = \lceil \frac{n}{n_b} \rceil$ factors F_k have the shape of an identity matrix besides the rows $(k-1)n_b + 1 : kn_b$, where it has the same content as F.
The inverse of F is already defined in Equation (3.3) and is repeated here for completeness:

$$F^{-1} = F_N^{-1} \cdot F_{N-1}^{-1} \cdots F_1^{-1}$$

To obtain the singular values of the generalized SVD, the SVD of $G \cdot F^{-1}$ can be computed.

The inverse of F can be replaced by the stepwise application of its factors F_k^{-1}:

$$\begin{aligned} W &= G \cdot F^{-1} \\ &= G \cdot F_N^{-1} \cdot F_{N-1}^{-1} \cdots F_1^{-1} \end{aligned} \tag{3.28}$$

The F_k^{-1} have the blockwise non-identity matrix shape of F_k as shown in Figure 3.1. Applying a F_k^{-1} to G will again be denoted as "inversion step".

As stated before in Equation (3.6), the following abbreviations are used to describe the non identity part of F_k^{-1}:

$$\begin{aligned} D_k &= F_{k,k}^{-1} \\ E_k &= -D_k \cdot F_{k,k-1} \end{aligned}$$

D_k is the diagonal block of the inverse and E_k is the block next to the diagonal block of the inverse.

The goal is now, as in the transformation of the generalized eigenvalue problem to the standard eigenvalue problem, to apply one F_k^{-1} and immediately start to remove the occurring fill-in by orthogonal transformations. When the fill-in has been evicted from the matrix, then the next inversion step F_{k-1}^{-1} can be applied. Without removing the fill-in, the non-zeros outside the band would grow further and further by every step until the matrix is filled up completely.

A detailed description of the series of applying F_k^{-1} from right to a lower triangular banded matrix M and the subsequent orthogonal transformations will be given in the following lines. M takes over the role of G after the application of an arbitrary number of inversion steps which have been followed by a restoration of the band as described in the following.
Applying F_k^{-1} to the banded matrix M means updating the columns $(k-1)n_b - b_B + 1 : kn_b$ in the matrix. More specifically, the columns $(k-1)n_b + 1 : kn_b$ of M are multiplied by E_k and the result is added to the columns $(k-1)n_b - b_B + 1 : (k-1)n_b$ of M. Afterwards, the columns $(k-1)n_b + 1 : kn_b$ of M are multiplied with D_k. Both operations are limited to the rows $(k-1)n_b + 1 : kn_b + b_A$. The generated bulge is located in M's lower triangle in the columns $(k-1)n_b - b_B + 2 : kn_b$ and the rows $(k-1)n_b + b_A - b_B + 1 : kn_b + b_A - 1$. Figure 3.6 gives an illustration of the updated entries in M and the occurring fill-in.

Contrary to the eigenvalue problem, only one bulge in the lower triangle of the matrix appears. To chase it towards the lower end of the matrix, a QR decomposition has to be used. A LQ decomposition, which was also an option at the eigenvalue problem for chasing the bulge to the lower end, cannot be used here since it operates on the upper matrix half's bulge.
The QR decomposition is run on the columns $(k-1)n_b - b_B + 1 : kn_b$ in the rows $(k-1)n_b + b_A - b_B + 1 : kn_b + b_A$ of M. The obtained Q^T is applied from left to these rows. By this, new fill-in is generated in the upper triangle of the matrix in the columns $(k-1)n_b + b_A - b_B + 1 : kn_b + b_A - 1$ of the QR rows. The bulge moved by $b_A + 1$ columns to the right and by one row to the top. A LQ decomposition can be used now to chase the newly generated bulge towards the lower end. The LQ is run on the columns $(k-1)n_b + b_A - b_B + 1 : kn_b + b_A$ of the rows

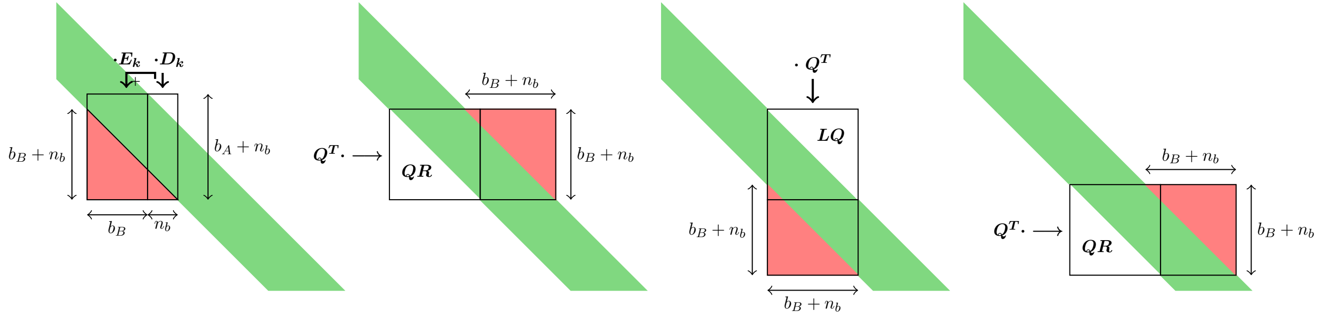


Figure 3.6: Lower matrix half: (First picture) The application of the inversion step k updates the entries within the rectangles (see main text for details). Marked in red is the bulge of newly created non-zeros outside the band. (Second picture) First chasing step for pushing the bulge by b_A columns to the right using QR decomposition. (Third picture) First LQ step for pushing the bulge by b_A rows to the bottom. (Forth picture) Second QR step for pushing the bulge by b_A columns to the right. Updates of matrix entries only happen within the rectangles.

$(k-1)n_b + b_A - b_B + 1 : kn_b + b_A$. The obtained Q^T is now applied from right to the matrix M and updates the LQ columns, removes the old bulge and introduces a new bulge in the rows $(k-1)n_b + 2b_A - b_B + 2 : kn_b + 2b_A$. The upper triangle is clean again and the new bulge is located in the lower triangle, $b_A + 1$ rows further down, but one column to the left compared to the old one. After one QR and one LQ, the bulge moved by b_A rows and by b_A columns towards the lower end of the matrix.
The procedure of computing and applying QR and LQ can be repeated until the fill-in is evicted from the matrix. Then the band is fully restored and the next inversion step F_{k-1}^{-1} can be applied to M. Figure 3.6 illustrates this bulge chasing chain in its pictures 2 to 4.

The orthogonal matrices obtained by QR and LQ will again be denoted by $Q_k^{(i)}$. The subscript k indicates the inversion step it follows after. The superscript (i) denotes the order of the orthogonal transformations starting with $Q_k^{(1)}$ for the first QR following the application of F_k^{-1}, then $Q_k^{(2)}$ for the first LQ transformation until $Q_k^{(\nu_k)}$ for the last LQ transformation that evicts the fill-in from the matrix.
ν_k can be computed when taking into account the upper most row where the fill-in appears, $(k-1)n_b + b_A - b_B + 2$. Every QR/LQ sequence moves the bulge by b_A rows to the bottom, where it finally drops out when the upper most bulge row of the newly generated bulge is beyond n. Hence,

$$\nu_k = 2\lfloor \frac{n - (k-1)n_b + b_B - 1}{b_A} \rfloor. \tag{3.29}$$

In the following, the series of $Q_k^{(i)}$ applied from left (generated by QR decompositions) will be abbreviated as

$$^{\circ}Q_k^{(\cdot)T} = Q_k^{(\nu_{k-1})T} \cdots Q_k^{(3)T} \cdot Q_k^{(1)T}, \tag{3.30}$$

the $Q_k^{(i)}$ applied from right (generated by LQ decompositions) are abridged as

$$^{\mathrm{e}}Q_k^{(\cdot)T} = Q_k^{(2)T} \cdot Q_k^{(4)T} \cdots Q_k^{(\nu_k)T}. \tag{3.31}$$

The given description can be summarized in a formal way: Applying an inversion step F_k^{-1} to the banded lower triangular matrix M creates fill-in outside the band. This fill-in can be removed by a series of orthogonal transformations:

$$\widehat{M} = {}^{\mathrm{o}}Q_k^{(\cdot)T} \cdot M \cdot F_k^{-1} \cdot {}^{\mathrm{e}}Q_k^{(\cdot)T} \tag{3.32}$$

$\widehat{M}$ is again a banded lower triangular matrix without fill-in and the next inversion step F_{k-1}^{-1} can be applied.

The overall transformation of applying F^{-1} to G while restoring the band consists of continuously repeating Equation (3.32) and by that applying all factors of F^{-1} while restoring the band immediately.
It results in the matrix $\widetilde{W}$ given by

$$\widetilde{W} = {}^{\mathrm{o}}Q_1^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_{N-1}^{(\cdot)T} \cdot {}^{\mathrm{o}}Q_N^{(\cdot)T} \cdot G \cdot F_N^{-1} \cdot {}^{\mathrm{e}}Q_N^{(\cdot)T} \cdot F_{N-1}^{-1} \cdot {}^{\mathrm{e}}Q_{N-1}^{(\cdot)T} \cdots F_1^{-1} \cdot {}^{\mathrm{e}}Q_1^{(\cdot)T}. \tag{3.33}$$

An algorithm for this procedure is given in Algorithm 23. As for the eigenvalue algorithms, the corrections to the bulge indices when a bulge reaches the end of a matrix have not been considered to simplify the code.

3.2.1 Twisted factorization

Similar to the eigenvalue procedure, one problem of using this algorithm is that fill-in generated close to the top end of the matrix is chased towards the lower end of the matrix. Again, the use of a twisted factorization provides the opportunity to overcome this issue. An illustration of a twisted factorization and its factors is given in Figure 3.3.

When assuming that G and F are factorizations of matrices $A = G^T \cdot G$ and $B = F^T \cdot F$, then four variants are possible:

- Factorizing A and B with a Cholesky factorization
  $\rightarrow G$ and F are banded lower triangular matrices
- Using a twisted factorization for A and factorize B with a Cholesky factorization
  $\rightarrow G$ is a twisted matrix, F is a banded lower triangular matrix
- Factorizing A with a Cholesky factorization and using a twisted factorization for B
  $\rightarrow G$ is a banded lower triangular matrix , F is a twisted matrix
- Using a twisted factorization for A and B
  $\rightarrow G$ and F are a twisted matrices

Algorithm 23 Crawford SVD algorithm (serial) for the lower matrix half. S is a twisted matrix with twist position $p = Pn_b$, G is a banded lower triangular matrix. When not using a twisted matrix but a banded lower triangular matrix S, then $p = P = 0$.

for $k \leftarrow N, N-1, \ldots, P+1$ **do**
  ▷ "inversion step" k(right-sided application of S_k^{-1})
  $i_1 \leftarrow (k-1)n_b + 1; \quad i_2 \leftarrow i_1 + n_b - 1$
  $j_1 \leftarrow i_1 - b_B; \quad j_2 \leftarrow i_1 - 1$
  $D_k \leftarrow S(i_1 : i_2, i_1 : i_2)^{-1}$
  if $(j_1 \leq j_2)$ **then**
    $E_k \leftarrow -D_k \cdot S(i_1 : i_2, j_1 : j_2)$
    $G(i_1 : i_2 + b_A, j_1 : j_2) \leftarrow G(i_1 : i_2 + b_A, j_1 : j_2) + G(i_1 : i_2 + b_A, i_1 : i_2) \cdot E_k$
  end if
  $G(i_1 : i_2 + b_A, i_1 : i_2) \leftarrow G(i_1 : i_2 + b_A, i_1 : i_2) \cdot D_k$

  ▷ "bulge chasing"
  $i \leftarrow 1$
  $i_1^b \leftarrow i_1 + b_A - b_B; \quad i_2^b \leftarrow i_2 + b_A$
  $j_1^b \leftarrow j_1; \quad j_2^b \leftarrow i_2$
  while $j_1^b < n - b_A$ **do**
    $[Q_k^{(i)},\ R_k^{(i)}] \leftarrow$ QR decomposition of $G(i_1^b : i_2^b, j_1^b : j_2^b)$
    $G(i_1^b : i_2^b, j_1^b : j_2^b + b_A) \leftarrow Q_k^{(i)T} \cdot G(i_1^b : i_2^b, j_1^b : j_2^b + b_A)$
    $j_1^b \leftarrow j_1^b + b_A; \quad j_2^b \leftarrow \min(j_2 + b_A, n)$

    $[L_k^{(i+1)},\ Q_k^{(i+1)}] \leftarrow$ LQ decomposition of $G(i_1^b : i_2^b, j_1^b : j_2^b)$
    $G(i_1^b : i_2^b + b_A, j_1^b : j_2^b) \leftarrow G(i_1^b : i_2^b + b_A, j_1^b : j_2^b) \cdot Q_k^{(i+1)T}$
    $i_1^b \leftarrow i_1^b + b_A; \quad i_2^b \leftarrow \min(i_2^b + b_A, n)$
    $i \leftarrow i + 2$
  end while
end for

In the following, the four cases will be discussed and advantages and disadvantages will be presented.

G and F are banded lower triangular matrices

The first variant, having G and F as banded lower triangular matrices, is discussed above and the procedure is given in Algorithm 23. It has the drawback of chasing bulges from the top end of the matrix towards the lower end.

G is a twisted matrix, F is a banded lower triangular matrix

This variant does not decouple the upper and lower matrix half. Removing fill-in in the upper matrix half and chasing the bulge towards the top end is not possible without introducing additional fill-in in the lower matrix half. This approach is hence not practical.

G is a banded lower triangular matrix, F is a twisted matrix

To clarify the different structure of twisted matrices, S will be used in the following instead of F. This is analogous to the eigenvalue algorithms where this naming has been introduced.

Equation (3.13) gives the factorization of the twisted matrix S

$$S = S_P \cdot S_{P-1} \cdots S_1 \cdot S_{P+1} \cdot S_{P+2} \cdots S_N$$

and Equation (3.14) the inverse of it:

$$S^{-1} = S_N^{-1} \cdot S_{N-1}^{-1} \cdots S_{P+1}^{-1} \cdot S_1^{-1} \cdot S_2^{-1} \cdots S_P^{-1}$$

The application of the S_k^{-1}, $k > P$, (lower matrix half) is described above. It does not cause fill-in in the upper matrix half. In the following, the application of S_k^{-1}, $k \leq P$, to G is described.

Analogously to the eigenvalue algorithm, $E_k = -D_k \cdot S_{k,k+1}$ denotes the inverse of the off-diagonal block in S_k and $D_k = S_{k,k}^{-1}$ denotes the inverse of the diagonal block.
Applying S_k^{-1} multiplies the columns $(k-1)n_b+1 : kn_b$ of G with E_k and adds the result to the columns $kn_b + 1 : kn_b + b_B$. Afterwards, the columns $(k-1)n_b + 1 : kn_b$ are updated by their multiplication with D_k. These operations update the rows $(k-1)n_b + 1 : kn_b + b_A$. A bulge is introduced in the rows $(k-1)n_b + 1 : kn_b + b_B - 1$ in the columns $(k-1)n_b + 2 : kn_b + b_B$. The bulge chasing is started with a QL decomposition of the rows $(k-1)n_b + 1 : kn_b + b_B$ and columns $(k-1)n_b + 1 : kn_b + b_B$ and a left-sided application of the obtained Q^T. This removes the bulge from the upper triangle of the matrix but introduces a new bulge in the lower triangle. The bulge moved by $b_A + 1$ columns to the left and one row to the bottom. The QL is followed by a RQ decomposition in the rows $(k-1)n_b + 1 : kn_b + b_B$ and columns $(k-1)n_b - b_A + 1 : kn_b - b_A + b_B$ and a right-sided application of the determined Q^T. This, again, removes the bulge and introduces a new bulge in the upper triangle in the rows

$(k-1)n_b - b_A + 1 : kn_b - b_A + b_B - 1$ and columns $(k-1)n_b - b_A + 2 : kn_b - b_A + b_B$. The bulge has now moved by $b_A + 1$ rows towards the top but one column to the right. Hence, the bulge moves every QL/RQ sequence by b_A rows and columns towards the top end of the matrix. This procedure is repeated until the bulge drops out of the matrix.
Figure 3.7 illustrates the procedure of applying the inversion step and removing the fill-in.

Due to the non-symmetric matrix G the bulge chasing pattern changes slightly. As for the lower matrix half, first, after applying S_k^{-1}, a left-sided transformation (here QL) is applied. It is followed by a right-sided transformation (RQ). In the lower matrix half, the last transformation that evicts the fill-in from the matrix was a right-sided transformation with Q^T generated by a LQ decomposition. In the upper matrix half, clearing the bulge from the matrix has to be done by a left-sided transformation, which is obtained by a QL decomposition.
This also leads to a slightly modified ν_k. For the computation, the most right column of the bulge has to be considered: $kn_b + b_B$. Like in the lower matrix half algorithm, the bulge moves by b_A columns every QL/RQ sequence. Hence, after stage i the bulge has its right most columns at $kn_b - \lfloor \frac{i}{2} + b_B \rfloor b_A$. The bulge drops out the matrix if the right most column is less or equal to 1. Therefore,

$$\nu_k = 1 + 2\lfloor \frac{kn_b + b_B - 1}{b_A} \rfloor. \tag{3.34}$$

Since ν_k is now appearing at the left-sided transformations, the abbreviations ${}^{\mathrm{o}}Q_k^{(\cdot)T}$ and ${}^{\mathrm{e}}Q_k^{(\cdot)T}$ have to be extended for the upper matrix half:

$$\begin{aligned}
{}^{\mathrm{o}}Q_k^{(\cdot)T} &= Q_k^{(\nu_k)T} \cdots Q_k^{(3)T} \cdot Q_k^{(1)T} && \text{for } k \le P \\
{}^{\mathrm{e}}Q_k^{(\cdot)T} &= Q_k^{(2)T} \cdot Q_k^{(4)T} \cdots Q_k^{(\nu_k - 1)T} && \text{for } k \le P \\
\\
{}^{\mathrm{o}}Q_k^{(\cdot)T} &= Q_k^{(\nu_k - 1)T} \cdots Q_k^{(3)T} \cdot Q_k^{(1)T} && \text{for } k > P \\
{}^{\mathrm{e}}Q_k^{(\cdot)T} &= Q_k^{(2)T} \cdot Q_k^{(4)T} \cdots Q_k^{(\nu_k)T} && \text{for } k > P
\end{aligned} \tag{3.35}$$

Equation (3.33) changes when having a twisted matrix S instead of a banded lower triangular matrix F to

$$\begin{aligned}
\widetilde{W} = {} & {}^{\mathrm{o}}Q_P^{(\cdot)T} \cdot {}^{\mathrm{o}}Q_{P-1}^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_1^{(\cdot)T} \cdot {}^{\mathrm{o}}Q_{P+1}^{(\cdot)T} \cdot {}^{\mathrm{o}}Q_{P+2}^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_N^{(\cdot)T} \\
& \cdot G \cdot S_N^{-1} \cdot {}^{\mathrm{e}}Q_N^{(\cdot)T} \cdot S_{N-1}^{-1} \cdot {}^{\mathrm{e}}Q_{N-1}^{(\cdot)T} \cdots S_{P+1}^{-1} \cdot {}^{\mathrm{e}}Q_{P+1}^{(\cdot)T} \\
& \cdot S_1^{-1} \cdot {}^{\mathrm{e}}Q_1^{(\cdot)T} \cdot S_2^{-1} \cdot {}^{\mathrm{e}}Q_2^{(\cdot)T} \cdots S_P^{-1} \cdot {}^{\mathrm{e}}Q_P^{(\cdot)T}.
\end{aligned} \tag{3.36}$$

Algorithm 24 gives a detailed description of the procedure for the upper matrix half. The bulge index computation is, as before, simplified.

The overall procedure consists of first running Algorithm 23 applying the lower matrix half of S and afterwards running Algorithm 24 applying the upper matrix half of S.
Using a twisted factorization for B allows to decouple the upper from the lower matrix half and reduces the bulge chasing distances significantly (see Table 3.2 for details).

Algorithm 24 Crawford SVD algorithm (serial) for the upper matrix half. S is a twisted matrix with twist position $p = Pn_b$, G is a banded lower triangular matrix. When not using a twisted matrix but a banded upper triangular matrix S, then $p = n$ and $P = N$.

for $k \leftarrow 1, 2, \ldots, P$ **do**
  ▷ "inversion step" k (right-sided application of S_k^{-1})
  $i_1 \leftarrow (k-1)n_b + 1; \quad i_2 \leftarrow min(i_1 + n_b - 1, p)$
  $j_1 \leftarrow i_2 + 1; \quad j_2 \leftarrow \min(i_2 + b_B, p)$
  $D_k \leftarrow S(i_1 : i_2, i_1 : i_2)^{-1}$
  if $(j_1 \leq j_2)$ **then**
    $E_k \leftarrow -D_k \cdot S(i_1 : i_2, j_1 : j_2)$
    $G(i_1 : i_2 + b_A, j_1 : j_2) \leftarrow G(i_1 : i_2 + b_A, j_1 : j_2) + G(i_1 : i_2 + b_A, i_1 : i_2) \cdot E_k$
  end if
  $G(i_1 : i_2 + b_A, i_1 : i_2) \leftarrow G(i_1 : i_2 + b_A, i_1 : i_2) \cdot D_k$

  ▷ "bulge chasing"
  $i \leftarrow 1$
  $i_1^b \leftarrow i_1; \quad i_2^b \leftarrow i_2 + b_B$
  $j_1^b \leftarrow i_1; \quad j_2^b \leftarrow j_2$
  while $j_2^b > 1$ **do**
    $[Q_k^{(i)},\ L_k^{(i)}] \leftarrow$ QL decomposition of $G(i_1^b : i_2^b, j_1^b : j_2^b)$
    $G(i_1^b : i_2^b, j_1^b - b_A : j_2^b) \leftarrow Q_k^{(i)T} \cdot G(i_1^b : i_2^b, j_1^b - b_A : j_2^b)$
    $j_1^b \leftarrow j_1^b - b_A; \quad j_2^b \leftarrow j_2^b - b_A$

    $[R_k^{(i+1)},\ Q_k^{(i+1)}] \leftarrow$ RQ decomposition of $G(i_1^b : i_2^b, j_1^b : j_2^b)$
    $G(i_1^b - b_A : i_2^b, j_1^b : j_2^b) \leftarrow G(i_1^b - b_A : i_2^b, j_1^b : j_2^b) \cdot Q_k^{(i+1)T}$
    $i_1^b \leftarrow i_1^b - b_A; \quad i_2^b \leftarrow i_2^b - b_A$
    $i \leftarrow i + 2$
  end while
end for

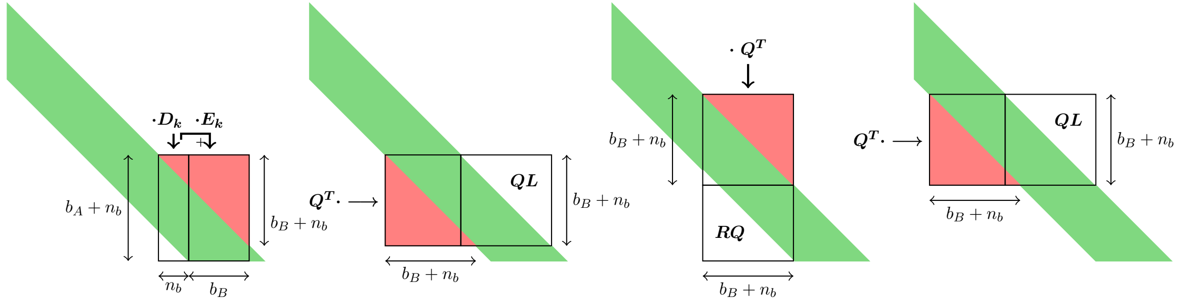


Figure 3.7: Upper matrix half when having a banded lower triangular matrix G and a twisted matrix S: (First picture) Application of inversion step k updates the entries within the rectangles (see main text for details). Marked in red is the bulge of newly created non-zeros outside the band.(Second picture) First chasing step for pushing the bulge by $b_A + 1$ columns to the left (and one row to the bottom) using QL decomposition. (Third picture) First RQ step for pushing the bulge by $b_A + 1$ rows to the top (and one column to the right). (Forth picture) Second QL step for pushing the bulge again by $b_A + 1$ columns to the left (and one row to the bottom). Updates of matrix entries only happen within the rectangles.

G and F are twisted matrices

A drawback of the previous approach, when having a twisted matrix S and a banded lower triangular G, is that the fill-in in the upper matrix half needs one more orthogonal transformation. Additionally, the upper and lower matrix half need their own code since the idea of reusing the code by flipping the upper matrix half cannot be applied here: Flipping the band of G in the upper matrix half does not give the same band pattern as in the lower matrix half. Hence, the chasing pattern cannot match.

A possibility to reuse the code of the lower matrix half in the upper matrix half appears if both matrices G and F are twisted matrices. This leads to having both band patterns and both bulge chasing patterns the same. It also preserves the decoupling of the upper and lower matrix half at bulge chasing. Also in this case, S will be used to indicate the twisted structure of F and hence will be used instead. For G the structure changes too, but since it has the passive part during the inversion steps, the notation remains as it is.

The application of the D_k and E_k is the same as described in the case of only having S as twisted matrix. Due to the upper triangular band in this case, the bulge is located in different rows: rows $(k-1)n_b - b_A + 1 : kn_b - b_A + b_B - 1$ of the columns $(k-1)n_b + 2 : kn_b + b_B$. The chasing will start with a QL, then a RQ step and will end, contrary to the case of having a twisted matrix only at S, with a RQ transformation. This is due to the location of the band and goes along with the transformations in the lower matrix half. Every QL transformation moves the bulge by $b_A + 1$ columns to the left and one row to the bottom, every RQ by $b_A + 1$ rows to the top and one column to the right. Again, every QL/RQ sequence shifts the bulge by b_A rows to the top left end of the matrix. Figure 3.8 gives an illustration of the procedure.

Since the transformations are similar to the lower matrix half, Equations (3.30) and (3.31)

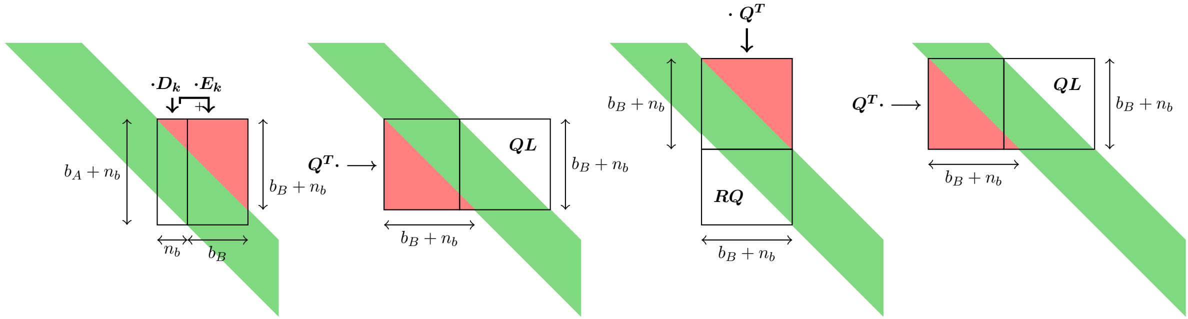


Figure 3.8: Upper matrix half when having two twisted matrices: (First picture) Application of inversion step k updates the entries within the rectangles (see main text for details). Marked in red is the bulge of newly created non-zeros outside the band. The band is now fully located in the triangle and hence the rows where the fill-in appears are different to the case when using a twisted factorization only for B. (Second picture) First chasing step for pushing the bulge by $b_A + 1$ columns to the left and one row to the bottom using QL decomposition. (Third picture) First RQ step for pushing the bulge by $b_A + 1$ rows to the top and one column to the right. (Forth picture) Second QL step for pushing the bulge by $b_A + 1$ columns to the left and one column to the bottom. Updates of matrix entries only happen within the rectangles.

with the abbreviations ${}^{\mathrm{o}}Q_k^{(\cdot)T}$ and ${}^{\mathrm{e}}Q_k^{(\cdot)T}$ also hold in this case for the upper matrix half. The number of orthogonal transformations can be computed by considering the lowest row of the initial bulge, $kn_b - b_A + b_B - 1$. The last row of the first RQ will consequently be $kn_b - b_A + b_B$. The last RQ row is considered here since it will be the operation that evicts the fill-in from the matrix. Every QL/RQ sequence the bulge moves by b_A rows towards the top end of the matrix. The drop out condition in this case is that the last RQ row is greater than 1. This leads to

$$\nu_k = 2\lfloor \frac{kn_b + b_B - 1}{b_A} \rfloor. \tag{3.37}$$

This algorithmic variant for the upper matrix half is a flipped version of the lower matrix half algorithm. Hence, it can be omitted to write code for it when flipping the underlying matrices and running the lower matrix half code on it (with small modifications). More details on flipping matrices are given in Section 3.1.1.

A drawback of this approach is that the resulting matrix $\widetilde{W}$ is a twisted matrix itself. Implementations of algorithms for the computation of the SVD of upper or lower triangular matrices exist, also for banded matrices (e.g. in two-step solvers). Implementations for twisted matrices, however, are not known to the author. Twisted matrices are partly used to compute the SVD of bidiagonal matrices [30], but not banded twisted matrices of wider band. Hence, when having twisted matrices G and F, computational routines for the further processing have to be developed.

Matrix setup	Chasing steps
S and G lower triangular matrices	$N^2 + 1.0N$
S twisted, G lower triangular matrix	$0.5N^2 + 1.5N$
S and G twisted matrices	$0.5N^2 + 1.0N$

Table 3.2: Number of chasing steps necessary over all inversion steps. The following simplifications are used for this computation: $n_b = b_A = b_B$, $n = Nn_b$ and $p = \frac{N}{2}$.

Summary

Equations (3.29), (3.34) and (3.37) give by ν_k the number of chasing steps necessary after inversion step S_k^{-1}. This formulas can be used to obtain the total number of bulge chasing steps for a matrix. To get a comparison, some simplifications are applied. When assuming $n_b = b_A = b_B$, $n = Nn_b$ and $p = \frac{N}{2}$, then the numbers in Table 3.2 give the amount of necessary bulge chasing steps for the different variants.

In the scenario described in the beginning of Section 3.2.1, where G and F are factorizations of matrices $A = G^T \cdot G$ and $B = F^T \cdot F$, it can be chosen which variant is used. For simplicity, S is here also used for the non-twisted matrix F, hence $B = S^T \cdot S$.
Variant two with a twisted matrix G and a banded lower triangular matrix S has not been considered since no formula has been developed for it.
Obviously, when having a twisted matrix for S, much less chasing steps are necessary. Hence, these approaches are preferable over the approach without twisted matrix S. The question of using a twisted factorization for A should not be decided based on the small advantage in the number of chasing steps compared to the Cholesky factorized A. Here, the focus lies on the further processing of the resulting matrix. If a parallel algorithm for computing the singular values for a twisted matrix is available, then using two twisted factorizations is the means of choice. This can already be motivated by reducing the required code by roughly 50%.

3.2.2 Singular vectors

The singular vectors of the original problem $W = G \cdot S^{-1} = U\Sigma V^T$ are U and V. Considering the orthogonal transformations, the problem transfers to $\widetilde{W} = \widetilde{U} \cdot \Sigma \cdot \widetilde{V}^T$. Obviously, the singular vectors are not the same and for obtaining them, a backtransformation step has to be applied.

Starting with Equation (3.36)

$$\begin{aligned}\widetilde{W} = {}^{\mathrm{o}}Q_P^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_1^{(\cdot)T} \cdot {}^{\mathrm{o}}Q_{P+1}^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_N^{(\cdot)T} \cdot G \cdot S_N^{-1} \\ \cdot {}^{\mathrm{e}}Q_N^{(\cdot)T} \cdots S_{P+1}^{-1} \cdot {}^{\mathrm{e}}Q_{P+1}^{(\cdot)T} \cdot S_1^{-1} \cdot {}^{\mathrm{e}}Q_1^{(\cdot)T} \cdots S_P^{-1} \cdot {}^{\mathrm{e}}Q_P^{(\cdot)T},\end{aligned}$$

the interchangeability of S_l^{-1} and ${}^{\mathrm{e}}Q_k^{(\cdot)T}$ can be inspected.
This consists of three steps: showing that all S_l^{-1} of the upper matrix half are interchangeable with ${}^{\mathrm{e}}Q_k^{(\cdot)T}$ for $l > k$ and showing that all S_l^{-1} of the lower matrix half are interchangeable

with ${}^{\mathrm{e}}Q_k^{(\cdot)T}$ for $l < k$. Finally, the interchangeability of S_l^{-1} of the upper matrix half with orthogonal factors of the lower matrix half ${}^{\mathrm{e}}Q_k^{(\cdot)T}$ has to be shown.
In the lower matrix half, S_l^{-1} updates the columns $(l-1)n_b - b_B + 1 : ln_b$. The first element of ${}^{\mathrm{e}}Q_k^{(\cdot)T}$, $Q_k^{(2)}$, updates the columns $(k-1)n_b + b_A - b_B + 1 : kn_b + b_A$. Since $l < k$ and $b_A \geq b_B$ it is ensured that the right most column updated by S_l^{-1} is left of the leftmost column updated by any of the ${}^{\mathrm{e}}Q_k^{(\cdot)T}$.
In the upper matrix half, the case with two twisted matrices and the case with only a twisted matrix in S do not differ. In both cases, the application of S_l^{-1} update the columns $(l-1)n_b + 1 : ln_b + b_B$. The first right-sided orthogonal transformation updates the columns $(k-1)n_b - b_A + 1 : kn_b - b_A + b_B$ and the following update columns further to the left. The leftmost columns of the inversion step are right of the rightmost columns of the ${}^{\mathrm{e}}Q_k^{(\cdot)T}$ for $l > k$ and $b_A \geq b_B$.
Hence, in both matrix halves, the inversion steps can be interchanged with the orthogonal transformation.
The most left right-sided transformation of the lower matrix half, $Q_{P+1}^{(2)}$, updates the columns $Pn_b + b_A - b_B + 1 : (P+1)n_b + b_A$. S_P^{-1}, the most right update of the upper matrix half, updates the columns $(P-1)n_b + 1 : Pn_b$ (the right limit is due to the use of a twisted matrix in S). Also here no intersection is given and lower matrix half's orthogonal transformations can be interchanged with the inversion steps of the upper matrix half.

Concluding, the inversion steps can be interchanged with the orthogonal transformations:

$$\begin{aligned}
\widetilde{W} =& {}^{\mathrm{o}}Q_P^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_1^{(\cdot)T} \cdot {}^{\mathrm{o}}Q_{P+1}^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_N^{(\cdot)T} \cdot G \cdot S_N^{-1} \\
& \cdot {}^{\mathrm{e}}Q_N^{(\cdot)T} \cdots S_{P+1}^{-1} \cdot {}^{\mathrm{e}}Q_{P+1}^{(\cdot)T} \cdot S_1^{-1} \cdot {}^{\mathrm{e}}Q_1^{(\cdot)T} \cdots S_P^{-1} \cdot {}^{\mathrm{e}}Q_P^{(\cdot)T} \\
=& {}^{\mathrm{o}}Q_P^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_1^{(\cdot)T} \cdot {}^{\mathrm{o}}Q_{P+1}^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_N^{(\cdot)T} \\
& \cdot G \cdot S_N^{-1} \cdots S_{P+1}^{-1} \cdot S_1^{-1} \cdots S_P^{-1} \\
& \cdot {}^{\mathrm{e}}Q_N^{(\cdot)T} \cdots {}^{\mathrm{e}}Q_{P+1}^{(\cdot)T} \cdot {}^{\mathrm{e}}Q_1^{(\cdot)T} \cdots {}^{\mathrm{e}}Q_P^{(\cdot)T} \\
=& {}^{\mathrm{o}}Q_P^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_1^{(\cdot)T} \cdot {}^{\mathrm{o}}Q_{P+1}^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_N^{(\cdot)T} \\
& \cdot G \cdot S^{-1} \\
& \cdot {}^{\mathrm{e}}Q_N^{(\cdot)T} \cdots {}^{\mathrm{e}}Q_{P+1}^{(\cdot)T} \cdot {}^{\mathrm{e}}Q_1^{(\cdot)T} \cdots {}^{\mathrm{e}}Q_P^{(\cdot)T}
\end{aligned}$$

With $W = G \cdot S^{-1} = U\Sigma V^T$ the link to the original problem can be established:

$$\begin{aligned}
\widetilde{W} =& {}^{\mathrm{o}}Q_P^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_1^{(\cdot)T} \cdot {}^{\mathrm{o}}Q_{P+1}^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_N^{(\cdot)T} \\
& \cdot G \cdot S^{-1} \\
& \cdot {}^{\mathrm{e}}Q_N^{(\cdot)T} \cdots {}^{\mathrm{e}}Q_{P+1}^{(\cdot)T} \cdot {}^{\mathrm{e}}Q_1^{(\cdot)T} \cdots {}^{\mathrm{e}}Q_P^{(\cdot)T} \\
=& {}^{\mathrm{o}}Q_P^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_1^{(\cdot)T} \cdot {}^{\mathrm{o}}Q_{P+1}^{(\cdot)T} \cdots {}^{\mathrm{o}}Q_N^{(\cdot)T} \\
& \cdot U \cdot \Sigma \cdot V^T \\
& \cdot {}^{\mathrm{e}}Q_N^{(\cdot)T} \cdots {}^{\mathrm{e}}Q_{P+1}^{(\cdot)T} \cdot {}^{\mathrm{e}}Q_1^{(\cdot)T} \cdots {}^{\mathrm{e}}Q_P^{(\cdot)T} \\
=& \widetilde{U} \cdot \Sigma \cdot \widetilde{V}^T
\end{aligned} \tag{3.38}$$

Thus, the left and right singular vectors U and V of W can be obtained by

$$U = {}^{\mathrm{o}}Q_N^{(\cdot)} \cdots {}^{\mathrm{o}}Q_{P+1}^{(\cdot)} \cdot {}^{\mathrm{o}}Q_1^{(\cdot)} \cdots {}^{\mathrm{o}}Q_P^{(\cdot)} \cdot \widetilde{U} \tag{3.39}$$

$$V = {}^{\mathrm{e}}Q_N^{(\cdot)T} \cdots {}^{\mathrm{e}}Q_{P+1}^{(\cdot)T} \cdot {}^{\mathrm{e}}Q_1^{(\cdot)T} \cdots {}^{\mathrm{e}}Q_P^{(\cdot)T} \cdot \widetilde{V} \tag{3.40}$$

or

$$V^T = \widetilde{V}^T \cdot {}^{\mathrm{e}}Q_P^{(\cdot)} \cdots {}^{\mathrm{e}}Q_1^{(\cdot)} \cdot {}^{\mathrm{e}}Q_{P+1}^{(\cdot)} \cdots {}^{\mathrm{e}}Q_N^{(\cdot)}. \tag{3.41}$$

Algorithm 25 gives the algorithm of the backtransformation for the singular vectors of the generalized singular value problem for the case of S being a twisted matrix, G being a banded lower triangular matrix as described in Section 3.2.1.

Algorithm 25 Backtransformation of the singular vectors of the generalized singular problem with a banded lower triangular matrix G and a twisted matrix S with twist position $p = Pn_b$. U is the matrix of left singular vectors, V is the matrix of right singular vectors.

```
for k ← P, P − 1, ..., 1 do
    for i ← ν_k, ν_{k−2}, ..., 3, 1 do
        i_1^b ← (k − 1)n_b − (i−1)/2 b_A + 1;    i_2^b ← kn_b − (i−1)/2 b_A + b_B
        U(i_1^b : i_2^b, :) ← Q_k^(i) · U(i_1^b : i_2^b, :)
    end for
end for
for k ← P + 1, P + 2, ..., N do
    for i ← ν_{k−1}, ν_{k−3}, ..., 3, 1 do
        i_1^b ← (k − 1)n_b + (i+1)/2 b_A − b_B + 1;    i_2^b ← kn_b + (i+1)/2 b_A
        U(i_1^b : i_2^b, :) ← Q_k^(i) · U(i_1^b : i_2^b, :)
    end for
end for

for k ← P, P − 1, ..., 1 do
    for i ← ν_{k−1}, ν_{k−3}, ..., 4, 2 do
        i_1^b ← (k − 1)n_b − i/2 b_A + 1;    i_2^b ← kn_b − i/2 b_A + b_B
        V(i_1^b : i_2^b, :) ← Q_k^(i)T · V(i_1^b : i_2^b, :)
    end for
end for
for k ← P + 1, P + 2, ..., N do
    for i ← ν_k, ν_{k−2}, ..., 4, 2 do
        i_1^b ← (k − 1)n_b + i/2 b_A − b_B + 1;    i_2^b ← kn_b + i/2 b_A
        V(i_1^b : i_2^b, :) ← Q_k^(i)T · V(i_1^b : i_2^b, :)
    end for
end for
```

When having two twisted matrices, the backtransformation step changes: It has to reflect the flipping of the matrix for the upper matrix half. This can be achieved by flipping the singular vectors before applying the upper matrix half's orthogonal transformations.
The flipping of the singular vectors can be achieved by a simplified flipping procedure as given

in Algorithm 26. This simplification is possible since the singular vectors are independent of each other. Clearly, the singular vectors have to be flipped back before applying the lower matrix half's $Q_k^{(i)}$.

Algorithm 26 Flipping of a singular vector matrix X of size $n \times w$: every single vector has to be turned upside down. Due to the independence of the single vectors, no left/right mirroring has to be done.

```
function M ← flipSingularVectors(X)
    [n, w] ← size(X)
    for j ← 1, w do
        for i ← 1, n do
            M(n − i + 1, j) ← X(i, j)
        end for
    end for
end function
```

3.2.3 Overall algorithm

Algorithm 28 provides the overall procedure of computing singular vectors and singular values of a given matrix pair G, S. G is a lower triangular banded matrix, S is a twisted matrix that is banded too. If these matrices are obtained by factorizations, Algorithm 17 can be used for their computation (with $p = 0$ for the non-twisted factorization).
First, the lower matrix half algorithm, then the upper matrix half algorithm have to be run. This is followed by the computation of the singular values and, if needed, the singular vectors. For this step, a solver exploiting the banded structure of the resulting matrix should be employed. Finally, if needed, the step of backtransformation obtains the singular vectors of the original problem.

For the case of having additionally G as a twisted matrix, Algorithm 29 provides the procedure to use. Instead of running a special algorithm for the upper matrix half, the matrix is flipped and the lower matrix half algorithm is used on the upper matrix half data. After flipping back the resulting matrix, a singular value solver that exploits the banded twisted structure is employed to compute the singular values and, if needed, the singular vectors. The latter have to undergo a backtransformation step which also includes a flipping of the singular vectors.

Algorithm 27 Backtransformation of the singular vectors of the generalized singular problem with twisted matrices G and S with twist position $p = Pn_b$. U is the matrix of left singular vectors, V is the matrix of right singular vectors.

```
U ← Flip matrix (U)                                              ▷ Algorithm 26
V ← Flip matrix (V)                                              ▷ Algorithm 26
for k ← P, P − 1, ..., 1 do
    for i ← ν_{k−1}, ν_{k−3}, ..., 3, 1 do
        ▷ "Work on flipped matrix:"
        i_1^b ← n − k n_b + (i+1)/2 b_A − b_B + 1;   i_2^b ← n − (k − 1) n_b + (i+1)/2 b_A
        U(i_1^b : i_2^b, :) ← Q_k^{(i)} · U(i_1^b : i_2^b, :)
    end for
    for i ← ν_k, ν_{k−2}, ..., 4, 2 do
        ▷ "Work on flipped matrix:"
        i_1^b ← n − k n_b + i/2 b_A − b_B + 1;   i_2^b ← n − (k − 1) n_b + i/2 b_A
        V(i_1^b : i_2^b, :) ← Q_k^{(i)T} · V(i_1^b : i_2^b, :)
    end for
end for
U ← Flip matrix (U)                                              ▷ Algorithm 26
V ← Flip matrix (V)                                              ▷ Algorithm 26

for k ← P + 1, P + 2, ..., N do
    for i ← ν_{k−1}, ν_{k−3}, ..., 3, 1 do
        i_1^b ← (k − 1) n_b + (i+1)/2 b_A − b_B + 1;   i_2^b ← k n_b + (i+1)/2 b_A
        U(i_1^b : i_2^b, :) ← Q_k^{(i)} · U(i_1^b : i_2^b, :)
    end for
    for i ← ν_k, ν_{k−2}, ..., 4, 2 do
        i_1^b ← (k − 1) n_b + i/2 b_A − b_B + 1;   i_2^b ← k n_b + i/2 b_A
        V(i_1^b : i_2^b, :) ← Q_k^{(i)T} · V(i_1^b : i_2^b, :)
    end for
end for
```

Algorithm 28 Overall algorithm for computing singular values and singular vectors if G is lower triangular banded and S is a twisted matrix with twist position $p = Pn_b$.

```
[G, Q_l] ← Crawford SVD algorithm lower matrix half (G, S)      ▷ Algorithm 23
[G, Q_u] ← Crawford SVD algorithm upper matrix half (G, S)      ▷ Algorithm 24

[U, Λ, V] ← Singular value problem solver for banded matrices (G)
[U, V] ← Backtransformation (U, V, Q_l, Q_u)                    ▷ Algorithm 25
```

Algorithm 29 Overall algorithm for computing singular values and singular vectors if G and S are twisted matrices with twist position $p = Pn_b$.

$[G,\ Q_l] \leftarrow$ Crawford SVD algorithm lower matrix half $(G,\ S)$ ▷ Algorithm 23
$G \leftarrow$ Flip matrix (G) ▷ Algorithm 20
$S \leftarrow$ Flip matrix (S) ▷ Algorithm 20
$[G,\ Q_u] \leftarrow$ Crawford SVD algorithm upper matrix half $(G,\ S)$ ▷ Algorithm 23
$G \leftarrow$ Flip matrix (G) ▷ Algorithm 20

$[U,\ \Lambda,\ V] \leftarrow$ Singular value problem solver for twisted banded matrix (G)
$[U,\ V] \leftarrow$ Backtransformation $(U,\ V,\ Q_u,\ Q_l)$ ▷ Algorithm 27

4 Parallel Algorithms

The main focus of this work is the development of parallel algorithms for the generalized eigenvalue problem and the generalized singular value problem. In the previous chapter, the algorithmic foundations have been presented. In this chapter the parallel variants of the presented serial algorithms are explained in detail.

4.1 Parallelization strategies in Numerical Linear Algebra

At the beginning, several well known parallelization strategies in Numerical Linear Algebra are presented. Many of the existing parallel algorithms in Linear Algebra use one or more of these strategies. The parallel implementations for the eigenvalue and the singular value problem employ all of them as their different layers of parallelization.

4.1.1 Distribution of data and computation

Linear algebra algorithms usually work with matrices and vectors. Both can contain huge amounts of data that undergoes similar or same operations. For example, scaling a vector will multiply each element by the same factor. A matrix vector product is more complex and performs the same operation on different data entries, similar for a matrix matrix product.
In the case of huge matrices and vectors it pays off to split the computation and the data between different processes. Every process performs a computation on its local data and one or more communication steps might become necessary to obtain the correct result.

One well established way of splitting the data is the data distribution used in ScaLAPACK. ScaLAPACK [15] is a standard library for distributed numerical computations and can be seen as extension of the LAPACK [3] library, the de-facto standard for non-distributed linear algebra computations.
ScaLAPACKuses locally LAPACK functions and employs as abstraction layer for the communication between processes BLACS. The available processes are arranged in a 2D grid, e.g. for 6 processes, the available options are 6×1 processes, 3×2 processes, 2×3 processes or 1×6 processes. Most beneficial are usually quadratic setups or setups close to quadratic setups.
The data is distributed blockwise to the processes according to a blocksize n_{scb}. Theoretically, these blocks can have different sizes in the two dimensions. However, in practice usually quadratic blocks are used. At the end of the matrix smaller blocksizes can occur since otherwise the matrix size would be restricted to multiples of n_{scb}.
The processes to assign the data to are repeated in a cyclical way in both dimensions. Hence, the data is distributed in a 2D blockcyclic way. Figure 4.1 gives an illustration of how a matrix is cut into blocks and how they are distributed over the processes of the grid.

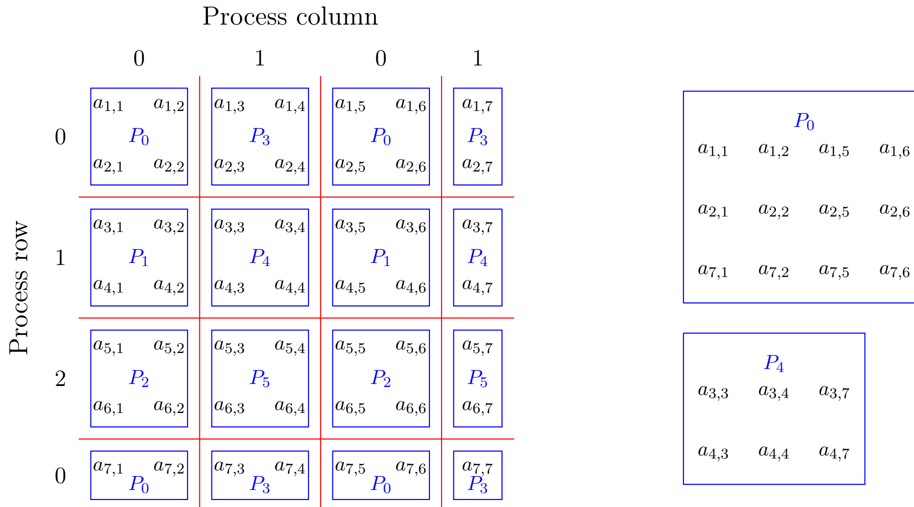


Figure 4.1: ScaLAPACK parallel data layout: The entries of the 7×7 matrix A are cut into blocks of size $n_{scb} \times n_{scb}, n_{scb} = 2$ as indicated in red. These blocks are distributed over the 3×2 process grid formed from the 6 available processes P_0 until P_5 in a 2D blockcyclic way. At the end of the matrix some blocks might have a reduced blocksize. In the right part of the picture the local data of process P_0 and process P_4 are shown.

4.1.2 Shared memory computation

Distributing computations and data targets mainly distributed memory systems and large computations. Due to the size of the data it will not fit into the memory of a single compute node. Thus, the data has to be distributed over several compute nodes and the computations are also distributed.
When, however, the data does not exceed the memory of a compute node, then the computation can be carried out in a shared memory setup. In a shared memory setup, the data is globally available, but the computation is performed locally by the single compute cores of the node independently.
Clearly, algorithms, that rely only on this approach are limited in the problem size. Another possibility is to follow a hybrid approach and break up the global computations into local subproblems and solve the subproblems by a shared memory approach.

4.1.3 Pipelining

Different from the two previous approaches where the focus lies on distributing data and computations, pipelining tries to organize the computations in an efficient way (which might also coincide with a tailored data distribution).
A pipeline consists of several workers that are arranged in a linear way. The output of worker one will become the input of worker two and so on.

Having one worker doing a task consisting of m steps (of equal duration), then he will be done with the task after m timesteps.

When having m workers for doing the task, every step can be performed by the m workers sharing the task. This would result (assuming full parallel efficiency) in finishing the task in one timestep. However, maybe the task cannot be distributed to m workers efficiently.
When additionally more tasks have to be run one after another, then the idea of pipelining becomes attractive. Every worker is specialized to one single step of the task and does only perform this step. Worker one starts with step one, hands the result over to worker two that performs step two on the input from worker one. In the meantime, worker one can start working on the next task. After m timesteps, the first task is completed since all workers did their step on this task. The second task has already $m - 1$ complete steps and so on for the later tasks. Hence, every following timestep another task is completed.
When starting the pipelining algorithm, a wind-up phase of m timesteps occurs until all workers are busy. Then the pipeline works fully efficient. Analogous to the wind-up phase, a wind-down phase begins when worker one has finished the last task. He will stay out of work from this time on and the parallel efficiency drops again.
Clearly, the throughput of the pipeline is limited by the slowest of its workers.

4.1.4 Splitting in independent subtasks

The idea of this approach is to split a task into smaller, independent subtasks. The subtasks itself are again divided into smaller sub-subtasks and so on until they can be solved easily by single workers. The results are stepwise combined and by that the solution of the result of the overall task can be obtained.
For the divide and conquer approach where each tasks is split in two subtasks this leads to a binary tree. Splitting the task according to a spacial distribution is another variant of this approach.

4.2 Twisted Crawford algorithm

4.2.1 Pipelining approach

The idea of pipelining was introduced in the previous section. The twisted Crawford algorithm consists of two operations in the transformation from the generalized to the standard eigenvalue problem as well as in the backtransformation of the eigenvectors: Applying an inversion step S_k and the following bulge chasing stages. In the following, a single bulge chasing stage $Q_k^{(i)}$ in the lower matrix half is going to be analyzed regarding pipelining.

The bulge chased by $Q_k^{(i)}$ in the lower triangle of the symmetric matrix A is located in the rows $(k-1)n_b + ib_A - b_B + 2 : kn_b + ib_A$ and columns $(k-1)n_b + (i-1)b_A - b_B + 1 : kn_b + (i-1)b_A - 1$. A QR decomposition of this bulge is computed and the obtained $Q_k^{(i)}$ is applied symmetrically to the matrix. As already mentioned, the bulge moves by b_A rows and columns towards the end. Thus, the bulge chasing seems to be suitable for a pipelining approach as it has a well-defined pattern. Every worker would receive a data package of $b_A \times b_A$ matrix entries.
However, applying the next inversion step $k-1$ will create fill-in in the rows $(k-2)n_b + ib_A - b_B + 2 : (k-1)n_b + ib_A$ and columns $(k-2)n_b + (i-1)b_A - b_B + 1 : (k-1)n_b + (i-1)b_A - 1$ in the lower triangle. This means the inversions steps move by n_b rows and columns towards the top.
Consequently, in every inversion step also the bulge chasing chain shifts by n_b columns and rows and hence additional matrix entries have to be transferred between the workers every inversion step. This will cause performance issues and has therefore to be avoided.
To overcome this issue, the free parameter n_b can be fixed to b_A. This aligns inversion steps and bulge chasing stages.

So far, the bulge has a size of $n_b + b_B$. This means, that even if the right end of the bulge is aligned to multiples of n_b, the left is not. Thus, incomplete blocks have to be processed. To get efficient block operations that work on full blocks, also b_B is fixed to b_A in the computations. Clearly, this might cause more computational work than necessary but the processing of full blocks will lead to more efficient operations which generally compensate the additional work.

To conclude, to achieve an efficient pipelining algorithm, the free choice of n_b is given up and it is bound to the bandwidth of the matrix A. Also b_B is potentially increased for this reason. Starting from here the following is fixed:

$$n_b = b_A = b_B \tag{4.1}$$

Figure 4.2 shows the simplified block structure of the matrices A, B and S. This uniform block structure allows to use a simplified notation. Since all operations are carried out on the block level it is sufficient to give the coordinates of the involved blocks instead of noting the exact row and column index. $M_{k,k}$ denotes the diagonal block in block row k which is located in the rows and columns $(k-1)n_b + 1 : kn_b$. $M_{k+1,k}$ is the block below $M_{k,k}$ in the rows $kn_b + 1 : (k+1)n_b$, $M_{k,k+1}$ is the block right of $M_{k,k}$ in the columns $kn_b + 1 : (k+1)n_b$. $M_{k:k+1,k}$ denotes the blocks $M_{k,k}$ and $M_{k+1,k}$.

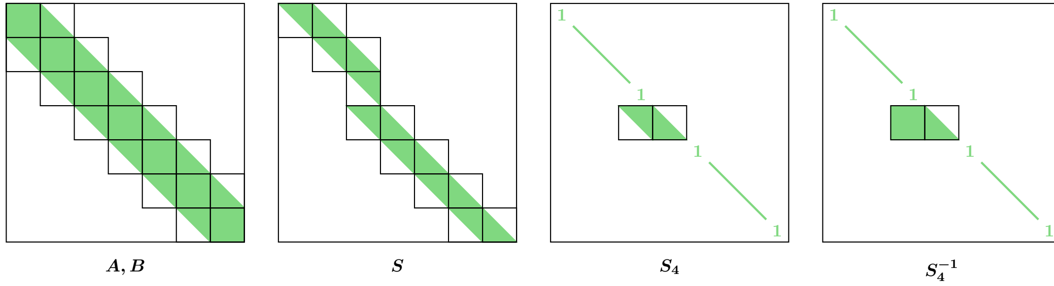


Figure 4.2: (first picture) Block structure of the matrices A and $B = S^T S$ with uniform bandwidth. (second picture) Block structure of the twisted factorization S. (third picture) One of its partial factors following Equation (3.13), S_4, and the inverse of it, S_4^{-1} (forth picture).

In Section 3.1.1 flipping of matrices was introduced. It allows to use the same algorithm for the upper and lower matrix half and hence reduces the coding effort by about 50%. In the following, only the lower matrix half algorithm is inspected since the upper matrix half algorithm can be mapped to it by flipping. Additionally, the matrix A is symmetric and all modifications to it are applied symmetrically. Hence, also all intermediate matrices and the resulting $\widetilde{C}$ are symmetric too. Thus, it is sufficient to restrict the further notes and the implementation to the lower triangle of the matrices.

As before, a short description of the simplified procedure of applying inversion steps and the following bulge chasing is presented. Applying inversion step S_k with $k > P$ (lower matrix half) from right updates the blocks $M_{k-1:k+1,k-1:k}$. The left-sided application updates the blocks $M_{k-1:k,k-1:k+1}$. The generated fill-in in the lower triangle is located in the blocks $M_{k:k+1,k-1}$ and $M_{k+1,k}$. The QR of $M_{k:k+1,k-1}$ is computed and the obtained $Q_k^{(1)}$ is applied symmetrically (involving block $M_{k+1,k}$ in the QR can be skipped since it would be filled up again by the right-sided part of the symmetric application).

The symmetric application of $Q_k^{(1)}$ can be performed in different ways. The left-sided application updates the blocks $M_{k:k+1,k-1:k+1}$, where $M_{k,k+1}$ is already in the upper triangle. The right-sided application updates the blocks $M_{k:k+2,k:k+1}$, where again $M_{k,k+1}$ is in the upper triangle. Four blocks are updated by the left and the right application, $M_{k:k+1,k:k+1}$. Two are only updated from left ($M_{k:k+1,k-1}$) and two are only updated from right ($M_{k+2,k:k+1}$).
Variant one would be to apply the symmetric update in a combined way as it is used in [4]. This variant works on the blocks $M_{k:k+2,k-1:k+1}$. A second variant performs first the left-sided update (where it generates $Q_k^{(1)}$). Afterwards, in a second step, the right-sided update is performed. Variant three would be to perform first a right-sided updated and then the left-sided update. This approach is not practical as working in the lower triangle, the $Q_k^{(1)}$ has to be obtained by a QR and not by a LQ of the upper matrix half bulge.
Hence, variant one and two remain. The advantage of variant two is that only two consecutive block rows or block columns are involved (see Figure 4.3). The right-sided update with $Q_k^{(1)}$ can be performed simultaneously to applying S_{k-1}, since the former updates block columns $k : k+1$, the latter $k-2 : k-1$. Additionally, further right-sided applications can be performed on pairs of block columns right of $k : k + 1$. This allows to work on all data simultaneously.

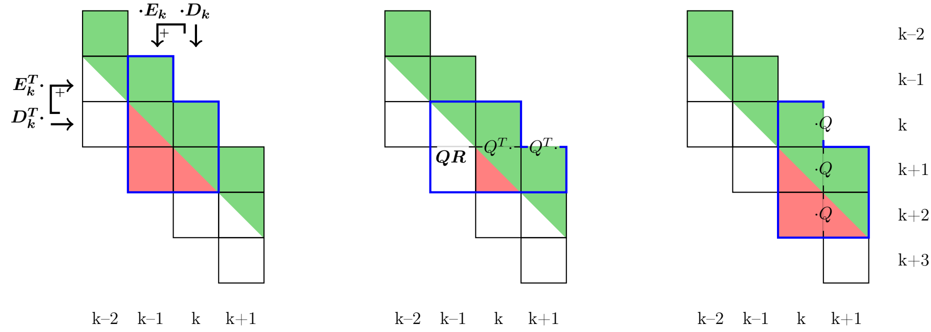


Figure 4.3: (left)The application of inversion step k updates the entries within the blue polygon. Marked in red is the bulge of newly created non-zeros outside the band. (middle) QR decomposition of the left blocks of the bulge and left-sided application of the obtained Q. (right) The right-sided application of Q creates new fill-in further down the matrix. The remaining part of the old bulge would be filled again in this step. (middle and right) The updated blocks are marked with a blue polygon. $Q^T\cdot$ and $\cdot Q$ on the border of two blocks indicate that this operation involves the two blocks. In the left as well as in the right-sided update, one block needs to interact with a block of the upper triangle of the matrix.

The same can be done regarding the left-sided updates.
The use of the combined application allows to use computational more efficient kernels for applying $Q_k^{(1)}$ for blocks undergoing a left and right-sided application. The blocks undergoing a one-sided application will usually be faster than the ones undergoing a symmetric application and remain idle until the symmetric blocks finish. The drawback here is that this variant uses 3 block rows and columns. Hence, less operations can be performed in parallel compared to variant two. Therefore, variant two with a splitting of the symmetric application of $Q_k^{(1)}$ in two separate steps has been chosen.

The splitting of the two-sided orthogonal transformations in a left-sided and a right-sided update can be generalized. The left-sided application of $Q_k^{(i)}$ updates the block rows $k+i-1:k+i$. Since $Q_{k+1}^{(i+1)}$ updates the block rows $k+i+1:k+i+2$, both work on different pairs of block rows and can be applied in parallel. More general, the left-sided application of all $Q_{k+j}^{(i+j)}$, $j>-i$, update disjunct pairs of block rows $k+i+2j-1:k+i+2j$ and hence, all $Q_{k+j}^{(i+j)}$ can be applied in parallel.
The same argument can be used for the right-sided application. The right-sided application of the $Q_{k+j}^{(i+j)}$, $j>-i$, update disjunct pairs of block columns $k+i+2j-1:k+i+2j$ and thus, also these right-sided applications can be run in parallel. Additionally to the application of the $Q_{k+j}^{(i+j)}$, S_{k-1} can be applied in parallel since it updates block columns $M_{k-2:k,k-2:k-1}$ (the most left update of $Q_{k+j}^{(i+j)}$ can be found for $j=-i+1$ and $i=1$; it updates the block columns $k:k+1$).
Figure 4.4 illustrates the obtained pipeline. Clearly, the number of parallel bulge chasing steps is limited by how far the inversion has moved on. The number of independent orthogonal transformations after applying inversion step S_k can be computed when considering the most left orthogonal transformation, $Q_k^{(1)}$, and the most right one, updating block rows and columns

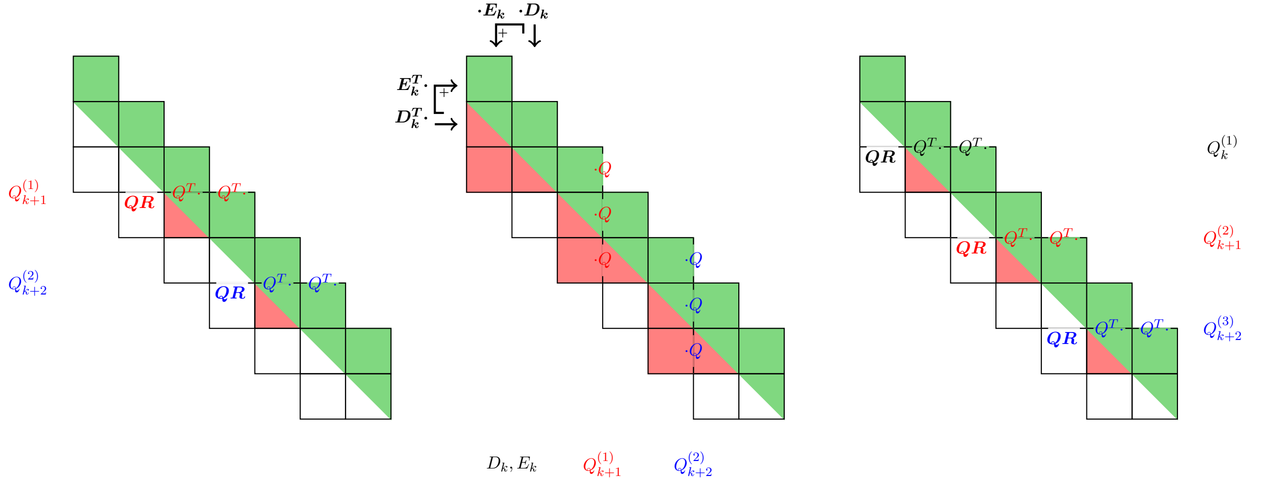


Figure 4.4: (left) Two QR decompositions are computed in parallel and applied from left. (middle) The right-sided application of the obtained Q completes the symmetric application of Q of the left picture. Simultaneously, inversion step S_k is applied. (right) The fill-in generated in the middle picture is removed by the next QR decompositions.

$N-2 : N-1$ or $N-1 : N$. The last orthogonal transformation that clears the bulge in row N and column $N-1$ is only performed in a very last step at the end of the algorithm and is hence not considered here. The number of block rows or columns between the most left and the most right orthogonal transformation determines the number of independent orthogonal transformations after applying S_k, which can be seen as the length of the pipeline. For the wind-up phase, the length is given by

$$l_{\text{pipe}}(k) = \lfloor \frac{N-k+1}{2} \rfloor. \tag{4.2}$$

The length of the pipeline rises every inversion step until applying the last inversion step. After this wind-up phase the length of the pipeline decreases every step by one in the wind-down phase until the algorithm terminates and the band is fully restored. The number of independent operations in the wind-down phase can be described by

$$l_{\text{pipe}}(i_{\min}) = \lfloor \frac{N-P-i_{\min}+1}{2} \rfloor, \tag{4.3}$$

where $i_{\min}$ denotes the smallest stage number within the row (which is held by $Q_{P+1}^{(i_{\min})}$).

Figure 4.5 shows the execution dependencies and clearly outlines the wind-up/wind-down character of the algorithm. All operations within one row are fully independent and can be executed at the same time. The operations within one diagonal belong to the same inversion step.

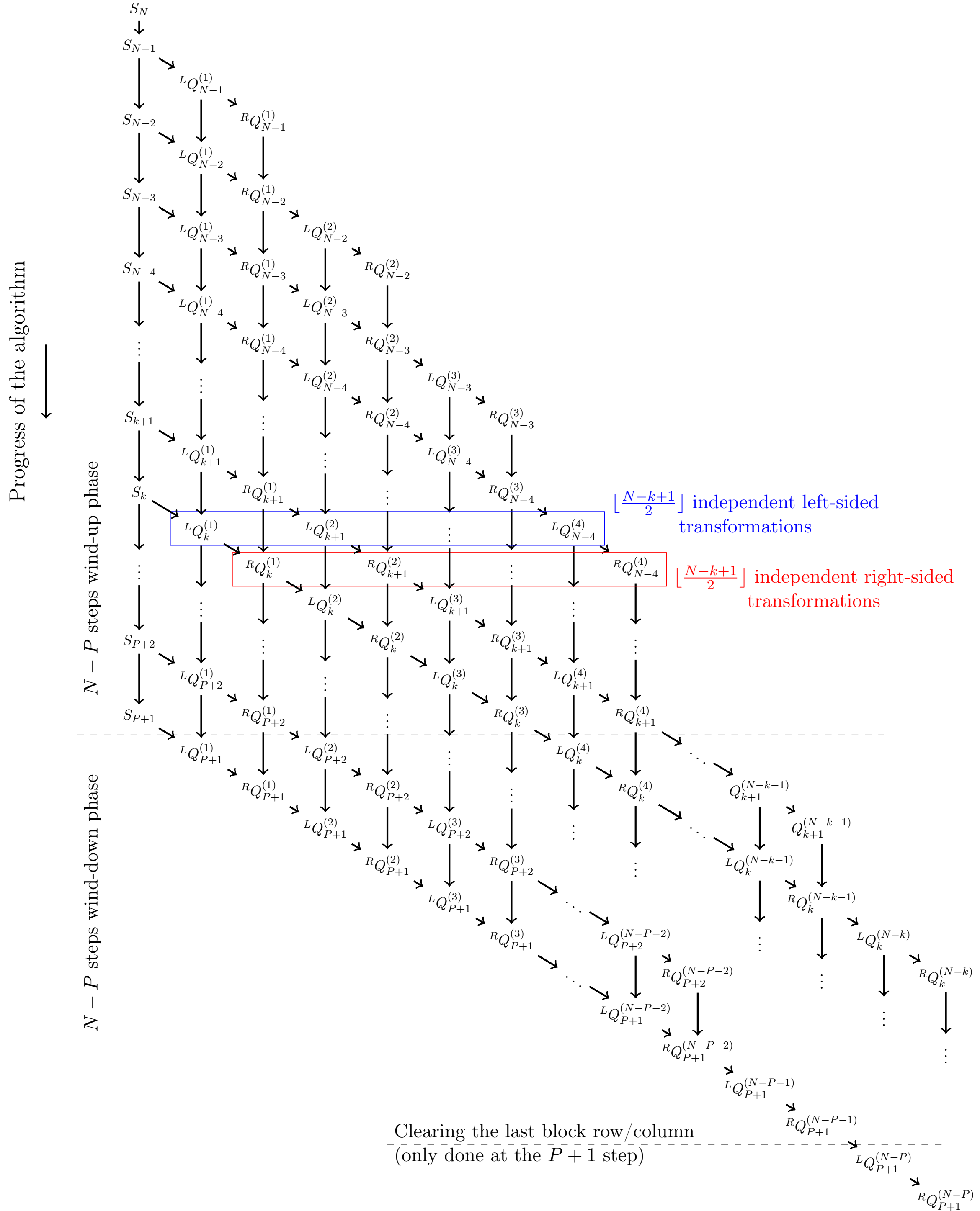


Figure 4.5: Execution dependencies of the different bulge chasing steps and stages: Every diagonal shows one inversion step and its following stages of bulge chasing. ${}^{L}Q_k^{(i)}$ denotes the left-sided application of the orthogonal transformation $Q_k^{(i)}$, ${}^{R}Q_k^{(i)}$ the right-sided application. All entries in one row are independent and can be executed at the same time. Only in the wind-up phase inversion steps are applied. The following wind-down phase only consists of bulge chasing steps. The last bulge chasing stage $Q_k^{(N-k+1)}$ is only applied in step $k = P + 1$.

4.2.2 Parallel execution of the upper and a lower matrix half

As described before, the upper and lower matrix half are widely decoupled. Especially, bulge chasing in the upper matrix half never updates the lower matrix half and bulge chasing in the lower matrix half never updates the upper matrix half. Regarding the inversion steps, this is not true. The last inversion step of the upper matrix half updates one block in the lower matrix half and vice versa. Figure 4.6 illustrates the last inversion steps of both matrix halves. The only block updated in the lower triangle of the upper matrix half by the lower matrix half is $A_{P,P}$. Since it is a diagonal block, no fill-in is generated. The only block in the lower matrix half updated by the upper matrix half is $A_{P+1,P}$. This block contains the band, but due to the multiplication with the upper triangular D_P no fill-in is introduced. Hence, fill-in appears only local to the matrix half applying the inversion step.
However, the blocks updated by the other matrix half have to be considered for the computation. Following the order in Equation (3.17), the lower matrix half has to be processed first and afterwards the upper matrix half is processed. This also means first running the lower matrix half algorithm, updating $A_{P,P}$ in the upper matrix half, and then running the upper matrix half algorithm updating $A_{P+1,P}$ in the lower matrix half.

Due to the widely independent operations in both matrix halves, both can be processed in parallel. This means that the lower matrix half starts with $k = N$ and runs until $k = P + 1$ while the upper matrix half starts from $k = 1$ and runs until $k = P - 2$. Then the exchange from the lower matrix half to the upper matrix half for updating $A_{P,P}$ takes place. After that, both halves can continue their computations. At some point in time D_P has to be applied to the lower matrix half. Because of Equation (3.17), this has to happen after $Q^{(1)}_{P+1}$ has been applied to $A_{P+1,P}$. From an algorithmic point of view, an application after the lower matrix half algorithm has fully cleaned the band is favorable since it omits additional extra case handling within the algorithm.

The possibility for parallel processing allows to use two fully independent groups of processes: one running the upper matrix half algorithm, one running the lower matrix half algorithm. The computational costs are roughly the same for both parts. Thus, neglecting the synchronization cost in the middle of the matrix and assuming the same amount of Flops per matrix half, a speedup of 2 can be expected by having two independent groups of processes.

Algorithm 30 provides the parallel algorithm for the lower matrix half. When flipping the upper matrix half it can also be used for the upper matrix half. The exchanging of data, which has been described above, has been simplified to keep the algorithm clearly arranged. The terms block column and block row are abbreviated by BR and BC in the algorithm. "Generating $Q^{(i)}_{k+i-1}$ and apply from left to BRs $k+i-1 : k+i$" means to run a QR decomposition on $A_{k+i-1:k+i,k+i-2}$ and to apply the obtained $Q^{(i)}_{k+i-1}$ to $A_{k+i-1:k+i,k+i-2:k+i}$.

4.2.3 Block and inter-block parallelization

Inter-block parallelization

Generating and applying orthogonal transformations $Q^{(i)}_k$ from left works on pairs of block rows. Applying inversion steps and applying $Q^{(i)}_k$ from right works on pairs of block columns.

Algorithm 30 Twisted Crawford algorithm.

$j \leftarrow N$; $k \leftarrow N$
windupPhase $\leftarrow$ true
while $j \leq N$ **do**
    if (windupPhase **and** (process has local data in BC $k-1:k$)) **then**
        apply D_k, E_k to BCs $k-1:k$
        if ($k = P+1$) **then**
            exchange data with other matrix half
        end if
    end if
    $i \leftarrow 1+(j-k)$; $i_0 \leftarrow 1+(j-k)$
    for $r_l \leftarrow j+1, j+3, ..., N$ **do**
        if (process has local data in BRs $r_l-1:r_l$) **then**
            generate $Q^{(i)}_{k+i-i_0}$ and apply $(Q^{(i)}_{k+i-i_0})^T$ from left to BRs $r_l-1:r_l$
        end if
        $i \leftarrow i+1$
    end for
    $i \leftarrow 1+(j-k)$; $i_0 \leftarrow 1+(j-k)$
    for $c_r \leftarrow j+1, j+3, ..., N$ **do**
        if (process has local data in BCs $c_r-1:c_r$) **then**
            apply $Q^{(i)}_{k+i-i_0}$ from right to BCs $c_r-1:c_r$
        end if
        $i \leftarrow i+1$
    end for
    if ($j = P+1$) **then**
        windupPhase $\leftarrow$ false
    end if
    if (windupPhase) **then**
        $k \leftarrow k-1$; $j \leftarrow j-1$
    else
        $j \leftarrow j+1$
    end if
end while
if (process has local data in BR N) **then**
    generate $Q^{(N-P+1)}_{P+1}$ and apply $(Q^{(N-P+1)}_{P+1})^T$ from left to BR N
end if
if (process has local data in BC N) **then**
    apply $Q^{(N-P+1)}_{P+1}$ from right to BC N
end if

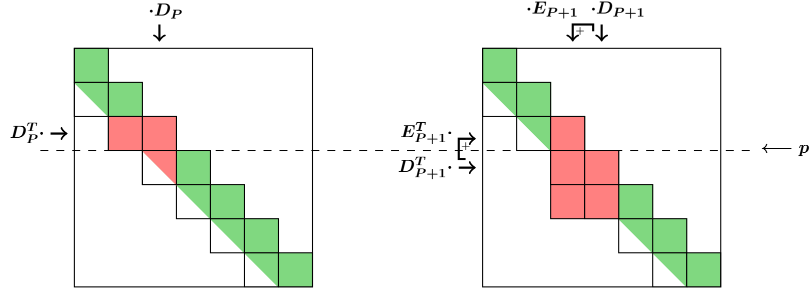


Figure 4.6: Application of the last inversion steps in the upper and lower matrix half when using a twisted factorization with twist block P. (left) The last inversion step of the upper matrix half updates the blocks indicated in red. (right) Last inversion step of the lower matrix half and its updated blocks.

In Section 4.2.1 the parallel capabilities of the pipeline have been discussed and the independence of pairs of block rows and pairs of block columns have been outlined. In the following, the work sharing within a pair of block rows and a pair of block columns is described.

The QR decomposition to obtain $Q_k^{(i)}$ is computed on the blocks $A_{k+i-1:k+i,k+i-2}$. The application of $Q_k^{(i)}$ to the BR pair $k+i-1 : k+i$ can be seen as a matrix multiplication from left with a square matrix of size two times two blocks. This means that every block pair $A_{k+i-1:k+i,k+i-2}$, $A_{k+i-1:k+i,k+i-1}$ and $A_{k+i-1:k+i,k+i}$ can be updated independently but every update involves the two blocks in the pair.
When assuming to have different processes in the different block rows, then a communicator between the two block rows has to be established. Having the same processes in two neighboring block rows is not practical since the pairs of block rows alternate: block row $k+i$ is collaborating with block row $k+i+1$ in the one step, in the other with $k+i-1$. See Figure 4.4 for an illustration of this behavior. Hence, neighboring block rows should not have the same processes.
The right-sided update works in the same way. It can be seen as a multiplication with a two times two blocks wide matrix $Q_k^{(i)}$ from right to a pair of block columns. Here, the different rows in the updated blocks are independent but the columns within the two neighboring blocks have to interact. Also here the interactions alters: block column $k+i$ is cooperating with $k+i-1$ in the one right-sided update, in the next with $k+i+1$. Hence, also two neighboring block columns should not have the same processes. For every block two local communicators along the columns have to be established, one to the left neighboring block, one to the right neighboring block.

Block parallelization

The coarse grained interactions between blocks in the left or right-sided updates have been described before. This will now be complemented by the inner-block parallelization and the computation strategies for these 2×3 and 3×2 block grids.

If the size of a block n_b is large enough, the data and hence the computation can further be

subdivided. For this, a 2-D process grid and a 2D blockcyclic data distribution as described in Section 4.1.1 is used. All operations work on full blocks and can hence be distributed easily. Detailed algorithms on the block and inter-block level are presented in Section 4.2.5.

Inner-block parallelization

When computing in shared memory environments, a further subdivision of the work on the process level to threads is possible. This can be achieved by using a multi-threaded LAPACK version instead of the standard version. The local calls to BLAS functions are then executed by more than one thread. A more extensive variant is the explicit use of OpenMP to distribute the block local work to threads explicitly. The latest MPI versions offer the possibility for shared memory programming directly within MPI. This work, however, restricts itself to a pure MPI implementation employing the possibility to plug-in a multi-threaded LAPACK version.

4.2.4 Process and data structures

Data structures

Due to the symmetry of the matrix it is sufficient to store only the lower triangle of the matrix. As illustrated before, for the application of an inversion step or the left or right-sided application of an orthogonal transformation, 2×3 or 3×2 blocks have to interact.
From Figure 4.4 it can be seen that three blocks in a block column are sufficient for the computation of the lower triangle if the upper triangle block can be surrogated. Block one will be the diagonal block, block two is the initially half filled block containing the border of the band and block three is solely for storing the occurring fill-in at this position.
Regarding the surrogate block, the pipeline can be inspected closer. The left-sided application of $Q_k^{(i)}$ updates the blocks $M_{k+i-1:k+i,k+i-2:k+i}$, the application of $Q_{k+1}^{(i+1)}$ happening in parallel updates the blocks $M_{k+i+1:k+i+2,k+i:k+i+2}$. Obviously, $M_{k+i+1,k+i-1}$, which is block three in block column $k+i-1$, is not involved in the operations. Since its fill-in has been cleared in the left-sided application of $Q_{k+1}^{(i)}$ before, it can serve as surrogate for the upper triangular $M_{k+i-1,k+i}$ in the left-sided application of $Q_k^{(i)}$.
Similar for the right-sided update where again the upper triangular $M_{k+i-1,k+i}$ is missing. The right-sided update of $Q_k^{(i)}$ updates the blocks $M_{k+i-1:k+i+1,k+i-1:k+i}$, the right-sided update with $Q_{k+1}^{(i+1)}$ updates $M_{k+i+1:k+i+3,k+i+1:k+i+2}$. In this case the block $M_{k+i+2:k+i+1,k+i}$ has been cleared by the previous left-sided update and is not involved in the right-sided update. Hence, it can act as surrogate for block $M_{k+i-1,k+i}$.
Inversion steps S_k can be applied without using a surrogate block since the missing block $M_{k-1,k}$ can be replaced by its lower triangular transpose, $M_{k,k-1}$. This block can be communicated to the processes holding $M_{k-1,k-1}$ where it is multiplied by E_k and added to $M_{k-1,k-1}$.
An illustration of the surrogate blocks and the changed communicators is given in Figure 4.7. Stored are only three blocks of size $n_b \times n_b$ per block column. In total, N block columns have to be stored. The blocks of the upper matrix half are already stored with flipped data and in a flipped arrangement. To simplify the computations, the diagonal block is not only stored as lower triangular block but as full block.

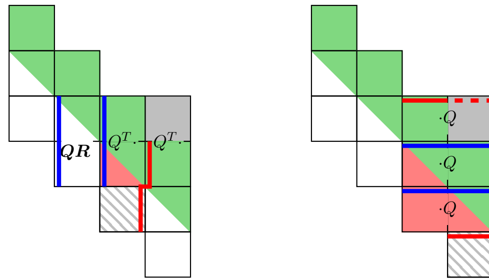


Figure 4.7: Sub-communicators that communicate across the block borders. The gray block illustrates the “missing” block in the upper triangle which is surrogated by a block that would normally not be involved in the computation (hatched). The thick lines illustrate the communicators, in blue the standard ones, in red the communicators that are used to communicate with the surrogate for the block in the upper matrix half.

Process layout

So far, different parallelization layers have been presented. The coarsest parallelization layer is the splitting of the matrix in an upper and a lower matrix half. Both parts work besides an exchange of one block fully independent and need only one synchronization point roughly in the middle of the runtime of the algorithm. Thus, the available processes are split in two groups: Processes for the upper matrix half and processes for the lower matrix half. The ratio of size of these groups should reflect the ratio of the number of block columns per matrix half.

Each of these groups is further subdivided equally into process groups (PG). A process group carries out the computation on the block level. The processes of a PG are arranged in a 2D process grid and the data is distributed to them in a 2D blockcyclic way. This distribution has been described in Section 4.1.1.

In the same way as three blocks are grouped together as block column, three process groups are grouped together as process group column (PGC). The process groups of a PGC are arranged in a fixed order as the blocks are arranged in a fixed order in a block column. The process group columns are numbered and have a well determined neighborhood. The block columns of a matrix half are assigned to the PGC of this matrix half in a cyclic way. Figure 4.8 illustrates the cyclic assignment: Block column 4 is assigned to PGC4 and block column 5 is again assigned to PGC1. This allows the PGC to have two fixed neighbors and establish local communicators with them. This local communicators allow to effectively communicate within the right-sided application of an orthogonal transformation.
For the communication in the left-sided application of an orthogonal transformation, the concept of a process group row (PGR) can be introduced. A process group row groups together the three blocks of block row. In the left-sided orthogonal transformation, similar to the right-sided transformation, a PGR has to interact with either its upper or lower neighbor PGR. Hence, also local communicators are established for pairs of PGRs.

The three main parallelization layers and the accompanying process distribution have been described. The most fine grained parallelization layer, work sharing via threads, is not further

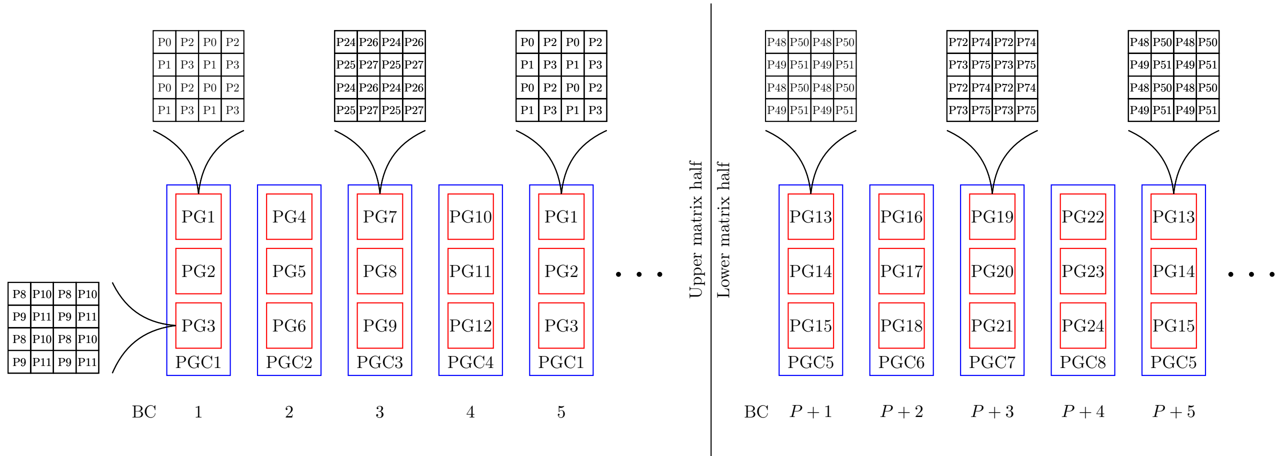


Figure 4.8: Different levels of parallelization in the algorithm and the distribution of the block columns (BC): Splitting of the matrix in upper and lower matrix half, process group columns (PGC), process groups (PG) and processes (P). Not shown here is the possible subdivision in threads.

described since the work sharing is handled by the LAPACK library within its BLAS routines.

4.2.5 Parallel algorithms on the level of pairs of block rows and block columns

Notation

To enable a precise description of the algorithms, a proper notation has to be introduced. For this work, the notation in [4] is adapted. It will be used in the rest of this work and is not restricted to a certain process layout.

In Auckenthaler's data layout the whole matrix is distributed in a 2D blockcyclic way to a 2D process grid. In this work the situation is different. Here, only the data of a block is distributed in a 2D blockcyclic way to a 2D process grid. The blocks itself collaborate within pairs of block columns or pairs of block rows, having disjunct processes in the underlying process groups. However, by setting up proper addressing methods, the 2×3 or 3×2 blocks in a pair of block rows or a pair of block columns can be seen as a continuum that is distributed to a set of processes. Hence, for the description of the algorithms on the level of pairs of block rows or pairs of block columns, a local index will be used.
In the 2×3 blocks setup, row indices $1 : n_b$ refer to the first block row, $n_b + 1 : 2n_b$ refer to the second block row. Similar for the block columns where the indices $n_b + 1 : 2n_b$ refer to the second block column. $M(1 : n_b, 2n_b + 1 : 3n_b)$ refers to block 3 in the upper block row which is located in the upper triangle. The fact that this block is surrogated by another block as described before will be implicitly incorporated to keep the descriptions compact.
The local indexing works the same for the 3×2 where the row indices last from $1 : 3n_b$ and the column indices from $1 : 2n_b$.

As already used before, $M(k, l)$ points to the entry in row k and column l of matrix M. To

point out the 2D blockcyclic distribution, the notation $M_{<p_r,p_c>}$ is used. $M_{<p_r,p_c>}(k,l)$ points thus to element (k,l) in a 2D distributed matrix where only the process owning the entry will act.
To indicate the replication of a matrix in one or more directions of the 2D process grid, "$*$" is used. This means that all processes in the row or column communicator, depending on whether "$*$" appears in the first or second coordinate, share the same data. For example $M_{<p_r,*>}$ is a matrix which is distributed over the rows of the process grid but replicated within the column communicator, hence replicated over the process columns.
Less general is the use of "$*_i$" which indicates that the data is replicated within block i of the asterisk direction. Hence, $M_{<p_r,*_1>}$ restricts the repetition of data to block column one. Still, the data is distributed over the process rows. Processes in the block columns two and three do not have copies of the data.
Finally, $M_{<*,*>}$ indicates that a matrix is replicated over all processes in the pair of PGC or PGR.

$M_{<p_r,p_c>}$ describes the distribution of the matrix over the processes. For local computations on single processes it is important to know about the row and column indices of the local matrices. These indices are added with squared brackets. $M_{<p_r,p_c>[i,j]}$ denotes a 2D distributed matrix where the data of the process at position $[i,j]$ is referenced. Some distributed computations require a reduction step at their end. This is indicated by adding another subscript after the squared brackets, describing the local data in the reduction direction (which can be seen by $< p_r, * >$. $M_{<p_r,*>[i]j}$ describes a matrix that has still to be reduced along the column communicator. A detailed example is presented in Figure 4.9. To shorten the notation, after already stating the distribution of a matrix, e.g. $< p_r, p_c >$, this part might be omitted in the rest of an algorithm.

Symmetric application of an inversion step

The parallel application of inversion step S_k is described in Algorithm 31. At first, the right block column has to be communicated to the left one. There, this communicated BC is multiplied by E_k and added to the local data. $A_{k-1,k}$, which would be used by $A_{k-1,k-1}$, is in the upper triangle of the matrix and hence not existing. This block is replaced by the lower triangular twin $A_{k,k-1}$. Afterwards, the right-sided application of S_k is completed by multiplying block column k from right by D_k. The missing left-sided update starts by transferring the updated block $A_{k,k-1}$ again to $A_{k-1,k-1}$, multiplying the transferred block by E_k^T and adding it to the local data. Finally, block row k is multiplied from left with D_k^T.
The two methods used for matrix multiplication in the algorithm, PDGEMM and PDTRMM , can be used from ScaLAPACK or be replace by some equivalent. They are used here to indicate that this parallel matrix multiplication is complete and not a local multiplication on local matrices.

QR decomposition and left-sided application

The parallel Householder vector generation ParallelHHgen is given in Algorithm 32. It extends the non-parallel Householder vector generation in Algorithm 1. Instead of having a globally

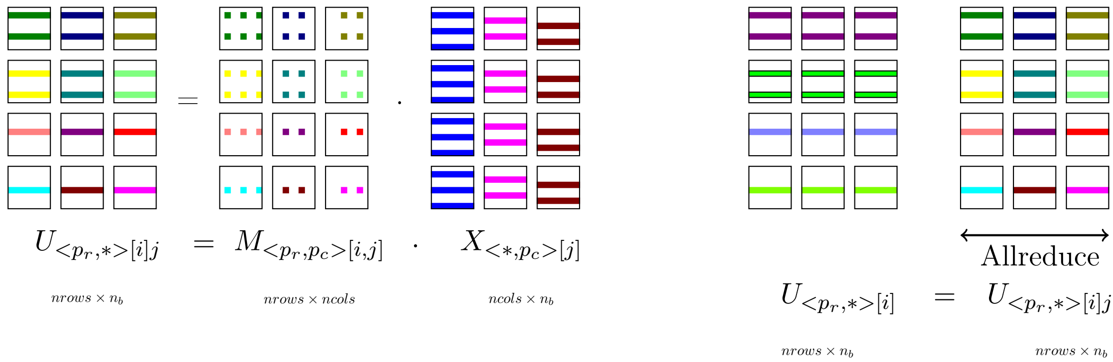


Figure 4.9: The distributed computation of the product of two matrices $U = M \cdot X$ illustrates the notation. The matrices are shown as a collection of quadrats, where each quadrat represents a process and its local view to the global matrix: The colored lines and rectangles indicates the local data. The quadrats for every matrix are arranged in the same way as the underlying processes are arranged in the process grid. The replication of data can be seen by having data at the same position with the same color (e.g. as for matrix X over the process rows). M is distributed in a 2D blockcyclic way, X is distributed in a 1D blockcyclic way over the process columns. Important for the multiplication is that the rows of X correspond to the columns of M. This is indicated by the subscript $[j]$. To compute the product, first local matrix multiplications are done on each process and the result is $U_{<p_r,*>[i]j}$. This is an intermediate result since a reduction step is necessary to complete the computation of $U_{<p_r,*>[i]}$. The $< p_r, * >$ in $U_{<p_r,*>[i]j}$ points to the resulting distribution over the process rows and the repetition over the columns. $[i]j$ points to the fact that this matrix is distributed over the process rows and the rows are the same as in M. Additionally, j without brackets indicates that the computation is still not complete since it is still distributed over the process columns and has hence to be reduced.

Algorithm 31 Parallel application of an inversion step.

function $A_{<p_r,p_c>[i,j]} \leftarrow$ ParallelDEapplication$(A_{<p_r,p_c>[i,j]},\ D_{<p_r,p_c>[i,j]},\ E_{<p_r,p_c>[i,j]})$
  if (process has local data in left BC) **then**
    if (process has local data in block 2 of the BC) **then**
      $b_{<p_r,p_c>[i,j]} \leftarrow A_{[i,j]}$
      MPI_SEND $(b_{<p_r,p_c>[i,j]})$ to block 1 in same BC
    else if (process has local data in block 1 of the BC) **then**
      $b_{<p_r,p_c>[i,j]} \leftarrow$ MPI_RECEIVE from block 2 in same BC
      $A_{[i,j]} \leftarrow A_{[i,j]} +$ PDGEMM $(b^T_{[i,j]},\ E_{[i,j]})$
    end if
    if (process has local data in block 2 or 3 of the BC) **then**
      $b_{<p_r,p_c>[i,j]} \leftarrow$ MPI_RECEIVE from block 1 or 2 in right BC
      $A_{[i,j]} \leftarrow A_{[i,j]} +$ PDGEMM $(b_{[i,j]},\ E_{[i,j]})$
      $b_{<p_r,p_c>[i,j]} \leftarrow A_{[i,j]}$
      MPI_SEND $(b_{<p_r,p_c>[i,j]})$ to block 1 in same BC
    else if (process has local data in block 1 of the BC) **then**
      $b_{<p_r,p_c>[i,j]} \leftarrow$ MPI_RECEIVE from block 2 in same BC
      $A_{[i,j]} \leftarrow A_{[i,j]} +$ PDGEMM $(E^T_{[i,j]},\ b_{[i,j]})$
    end if

  else
    if (process has local data in block 1 or 2 of the BC) **then**
      $b_{<p_r,p_c>[i,j]} \leftarrow A_{[i,j]}$
      MPI_SEND $(b_{<p_r,p_c>[i,j]})$ to block 2 or 3 in left BC
      $A_{[i,j]} \leftarrow$ PDTRMM $(A_{[i,j]},\ D_{[i,j]})$
    end if
  end if
  if (process has local data in left BC in block 2 or in right BC in block 1) **then**
    $A_{[i,j]} \leftarrow$ PDTRMM $(D^T_{[i,j]},\ A_{[i,j]})$
  end if
end function

available vector x, this vector is distributed over the process rows. Hence, the squared norm of x has to be obtained by a reduction operation. Also the first entry has to be distributed to all processes by the reduce. All computations besides the squared norm of x are repeated on every process in the process column. The resulting Householder vector v is distributed as the given vector x, the scalar factor τ is available on all processes in the process column.

Algorithm 32 Parallel Householder vector generation, the first entry of the given vector remains.

function $[x_{<p_r>[i]},\ v_{<p_r>[i]},\ \tau_{<*>}] \leftarrow \text{ParallelHHgen}(x_{<p_r>[i]})$
  $d_{<*>i} \leftarrow x_{[i]}^T \cdot x_{[i]}$
  $a_{<*>i} \leftarrow 0$
  if ($x(1)$ is local) **then**
    $a_{<*>i} \leftarrow x(1)$
  end if
  $[a_{<*>},\ d_{<*>}] \leftarrow \text{MPI_ALLREDUCE}\ (a_{<*>i},\ d_{<*>i})$
  $\beta \leftarrow \text{sign}(x(1)) \cdot \sqrt{d}$
  $\tau \leftarrow \frac{a+\beta}{\beta}$
  $v_{[i]} \leftarrow \frac{1}{a+\beta} x_{[i]}$
  $x_{[i]} \leftarrow 0$
  if ($x(1)$ is local) **then**
    $v(1) \leftarrow 1$
    $x(1) \leftarrow -\beta$
  end if
end function

To apply Householder vectors and the regarding factors, Algorithm 3 has been defined. This algorithm is extended to a parallel setup by ParallelApplyHHleft in Algorithm 33. The given Householder vector, which is distributed over the process rows, and the Householder factor, that is available on every process, are applied to the 2D distributed matrix A. The distribution of the matrix and the vectors requires an additional reduction step to complete the multiplication $A^T v$.

Algorithm 33 Parallel left-sided application of a Householder vector.

function $A_{<p_r,p_c>[i,j]} \leftarrow \text{ParallelApplyHHleft}(A_{<p_r,p_c>[i,j]},\ v_{<p_r,*>[i]},\ \tau_{<*,*>})$
  $z_{<p_c>i[j]} \leftarrow A_{[i,j]}^T \cdot v_{[i]}$
  $z_{<p_c>[j]} \leftarrow \text{MPI_ALLREDUCE}\ (z_{<p_c>i[j]})$
  $A_{[i,j]} \leftarrow A_{[i,j]} - \tau \cdot v_{[i]} \cdot z_{[j]}$
end function

Algorithms 32 and 33 are used in the parallel variant of Algorithm 5, ParallelQR, given in Algorithm 34. For all columns, if local on a process, the Householder vector is computed and then distributed to all process columns within the block. Afterwards, the Householder vector is applied from left and the next column follows.

Algorithm 34 Parallel QR decomposition.

function $[A_{<p_r,p_c>[i,j]},\ Y_{<p_r,*_1>[i]},\ \tau_{<*,*_1>}] \leftarrow \text{ParallelQR}(A_{<p_r,p_c>[i,j]})$
  $Y \leftarrow 0$
  for $k \leftarrow 1, 2, \ldots, n$ **do**
    if (k is local column) **then**
      $[A_{<p_r>[i]}(k:m,k),\ Y_{<p_r>[i]}(k:m,k),\ \tau_{<*>}(k)]$
        $\leftarrow \text{ParallelHHgen}(A_{<p_r>[i]}(k:m,k))$
    end if
    $[Y_{<p_r,*_1>[i]}(k:m,k),\ \tau_{<*,*_1>}(k)]$
      $\leftarrow$ MPI_BROADCAST $(Y_{<p_r>[i]}(k:m,k),\ \tau_{<*>}(k))$

    $A(k:m,k+1:n) \leftarrow \text{ParallelApplyHHleft}(A(k:m,k+1:n),\ Y_{[i]}(k:m,k),\ \tau(k))$
  end for
end function

During ParallelQR, the Householder vectors have been collected in the distributed matrix $Y_{<p_r,*>[i]}$ and the factors in global vector $\tau_{<*,*>}$. When using the compact WY formulation for applying Householder vectors in a blocked way, Algorithm 35 provides the distributed computation of $T_{<*,*>}$. Again, a matrix multiplication has to be completed by a reduce.

Algorithm 35 Parallel computation of T.

function $T_{<*,*>} \leftarrow \text{ParallelGenT}(Y_{<p_r,*>[i]}(1:m,1:n),\ \tau_{<*,*>}(1:n))$
  $T \leftarrow 0$
  $Z_{<*,*>i} \leftarrow Y_{[i]}^T \cdot Y_{[i]}$
  $Z_{<*,*>} \leftarrow$ MPI_ALLREDUCE $(Z_{<*,*>i})$
  for $k \leftarrow 1, n$ **do**
    $T(1:k-1,k) \leftarrow -\tau(k) \cdot T(1:k-1,1:k-1) \cdot Z(1:k-1,k)$
    $T(k,k) \leftarrow \tau(k)$
  end for
end function

For the blocked application of Householder vectors, a parallel version of Algorithm 11 is given in Algorithm 36. The Householder vector matrix $Y_{<p_r,*>[i]}$ as well as the matrix where the transformations are applied to, $A_{<p_r,pc>[i,j]}$, are distributed. The globally available $T_{<p_r,p_c>}$ completes the arguments for ParallelCWYleft. Due to global availability of T, only one reduction step is necessary to complete the distributed computation.

By using Algorithms 34 to 36, a blocked version of the parallel QR decomposition and the left-sided application to the pair of block rows can be formulated. Algorithm 37 provides the detailed description for that. First, the QR decomposition of a small tile is computed. After this step, all processes in block column one share parts of the obtained Householder vectors and factors. These Householder vectors are communicated to the blocks two and three in the block row via a MPI_SEND / MPI_RECEIVE . This communication takes place in a one-to-one fashion: Every process of the sending block sends its data to the process in the receiving block which has the same position in the 2D process grid. The same can be achieved by immediately

Algorithm 36 Parallel blocked application of Householder vectors from left.

function $A_{<p_r,p_c>[i,j]} \leftarrow \text{ParallelCWYleft}(A_{<p_r,p_c>[i,j]},\ Y_{<p_r,*>[i]},\ T_{<*,*>})$
  $Z_{<p_r,*>[i]} \leftarrow Y_{[i]} \cdot T$
  $U_{<*,p_c>i[j]} \leftarrow Z_{[i]}^T \cdot A_{[i,j]}$
  $U_{<*,p_c>[j]} \leftarrow \text{MPI_ALLREDUCE}\ (U_{<*,p_c>i[j]})$
  $A_{[i,j]} \leftarrow A_{[i,j]} - Y_{[i]} \cdot U_{[j]}$
end function

using a broadcast over the full PBR during ParallelHHgen. Afterwards, all processes compute redundantly T and finally, by calling ParallelCWYleft, apply the Householder vectors in a blocked way to the distributed matrix.

Algorithm 37 Parallel blocked QR decomposition.

function $[A_{<p_r,p_c>[i,j]},\ Y_{<p_r,*>[i]},\ \tau_{<*,*>}] \leftarrow \text{ParallelBlockedQR}(A_{<p_r,p_c>[i,j]})$
  for $j_1 \leftarrow 1, b+1, 2b+1, \ldots, n_b$ **do**
    $i_1 \leftarrow j_1;\qquad i_2 \leftarrow 2n_b$
    $j_2 \leftarrow \min(n_b,\ j_1+b-1)$
    if (is block column 1) **then**
      $[A_{[i,j]}(i_1:i_2, j_1:j_2),\ Y_{<p_r,*_1>[i]}(i_1:i_2,j_1:j_2),\ \tau_{<*,*_1>}(j_1:j_2)]$
          $\leftarrow \text{ParallelQR}(A_{[i,j]}(i_1:i_2,j_1:j_2))$

      MPI_SEND $(Y_{<p_r,*_1>[i]}(i_1:i_2,j_1:j_2),\ \tau_{<*,*_1>}(j_1:j_2))$ to BCs 2, 3
    else
      $[Y_{<p_r,*>[i]}(i_1:i_2,j_1:j_2),\ \tau_{<*,*>}(j_1:j_2)] \leftarrow$ MPI_RECEIVE from BC 1
    end if

    $T_{<*,*>} \leftarrow \text{ParallelGenT}(Y_{[i]}(i_1:i_2,j_1:j_2),\ \tau(j_1:j_2))$
    $A_{[i,j]}(i_1:i_2, j_2+1:3n_b) \leftarrow \text{ParallelCWYleft}(A_{[i,j]}(i_1:i_2,j_2+1:3n_b),$
        $Y_{[i]}(i_1:i_2,j_1:j_2),\ T)$
  end for
end function

Right-sided application of Q

In the generation of Q and the left-sided update, Y has been distributed over the rows and replicated over the columns. The right-sided application of Q is basically the transposed version of the left-sided, hence, also a transposed version of Y has to be obtained.
Y is distributed over two blocks. When computing block-local transposes, then these local transposes have just to be ordered in the right way to obtain the overall transpose. This means that the upper block becomes the left block and the lower block becomes the right block in the newly obtained transpose. Figure 4.10 illustrates the necessary communication of the parts of Y. The left block column has only to communicate block two since block three was already involved as surrogate for the upper triangular block in the left-sided transformation. In the right BC only one block has the lower part of Y from the left-sided transformation. Thus,

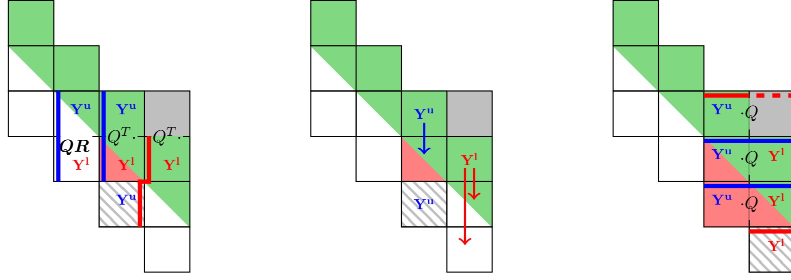


Figure 4.10: Redistribution of $Y_{<p_r,*>}$ after the left-sided update: (left) Initial distribution of Y with splitting $Y = [Y^u \ Y^l]^T$ in an upper and an lower part. (middle) Communication of Y^u and Y^l within the block columns. (right) Final distribution of Y^u and Y^l as needed for the right-sided update. The thick lines illustrate the communicators along which Y will be used in the right-sided update, in gray is the upper triangular block that is surrogated by the hatched block.

this block has to communicate it to the two neighboring blocks in the block column.
This communication step can again be done with point to point communication between the single processes in the regarding process grids in the process groups of the blocks. This communication step can take place for all right-sided updates at once before the right-sided update itself is carried out.

Algorithm 35 provided the computation of T in the description of the left-sided orthogonal transformation. There, $Y_{<p_r,*>}$ was used to obtain T. In the right-sided update, however, the transposed version of Y, $Y'_{<*,p_c>}$ is used. Transposed does not mean that the data distribution and the local data is transposed. Y' does only include the transpose of the data distribution, the local data itself stays untransposed (a column of Y' still contains the local parts of one Householder vector). Algorithm 38 provides the computation of T when using Y'. The changes are marginal as only the distribution direction of Y and hence the necessary reduction direction changes from rows to columns. If the matrix T is stored after the left-sided update, then this computation becomes obsolete.

Algorithm 38 Parallel computation of T.

function $T_{<*,*>} \leftarrow \text{ParallelGenT}(Y'_{<*,p_c>[j]}(1:m, 1:n),\ \tau_{<*,*>}(1:n))$
  $T \leftarrow 0$
  $Z_{<*,*>i} \leftarrow Y'^T_{[j]} \cdot Y'_{[j]}$
  $Z_{<*,*>} \leftarrow \text{MPI_ALLREDUCE}\ (Z_{<*,*>j})$
  for $k \leftarrow 1, n$ **do**
    $T(1:k-1,k) \leftarrow -\tau(k) \cdot T(1:k-1, 1:k-1) \cdot Z(1:k-1,k)$
    $T(k,k) \leftarrow \tau(k)$
  end for
end function

For the right-sided application of Householder vectors, in Algorithm 12 the compact WY representation was presented. It performs the blockwise application of a set of Householder vectors at once and employs only BLAS3 computations. Algorithm 39 provides the parallel

version of this algorithm which needs an additional reduction step to complete an intermediate product.

Algorithm 39 Parallel blocked application of Householder vectors from right.

function $A_{<p_r,p_c>[i,j]} \leftarrow \text{ParallelCWYright}(A_{<p_r,p_c>[i,j]},\ Y'_{<*,p_c>[j]},\ T_{<*,*>})$
  $Z_{<p_r,*>[j]} \leftarrow Y'_{[j]} \cdot T$
  $U_{<*,p_c>[i]j} \leftarrow A_{[i,j]} \cdot Z_{[j]}$
  $U_{<*,p_c>[i]} \leftarrow \text{MPI_ALLREDUCE}\ (U_{<*,p_c>[i]j})$
  $A_{[i,j]} \leftarrow A_{[i,j]} - U_{[i]} \cdot Y'_{[j]}$
end function

Algorithm 40 describes the right-sided application of an orthogonal transformation. As already mentioned before, a transposed version of Y has to be obtained first. The necessary communication steps to bring the right parts of Y in the right place has been described at Figure 4.10. When having the correct distribution of Y, the local "transpose" (see above) can be carried out to obtaine $Y'_{<*,p_c>}$. As for the QR decomposition and the left-sided application of Q, also for the right-sided application a tile-wise application is applied. The tile size b can be chosen arbitrarily and is not bound to the tile size in the QR step. For every tile Algorithm 38 can be used to compute T and by Algorithm 39 the application of the right-sided update is carried out.
Figure 4.10 additionally illustrates the blocks in the upper triangle of the matrix and their surrogates. By having a closer look, it can be seen that the upper triangular block is the same in the left and the right-sided update. This can be exploited at the right-sided update: Instead of transposing the lower triangular sibling, the content of the surrogate block after the left-sided transformation can be used. By that, no transpose is necessary.

So far, the computation of T is carried out only for the left-sided application of Q. For the right-sided application, the stored T of the left application is used. This means that a unique tile size is used and the computation of T during the right-sided application can be saved. Similar to Y, T has to be distributed to the blocks not having been involved in the left-sided update (see Figure 4.10 for details).

4.3 Backtransformation of eigenvectors

In the following the parallelization of the backtransformation of the eigenvectors is described. Equations (3.26) and (3.27) provided formulas for this computation. The latter is to use when employing a twisted factorization and shall be recalled here:

$$X = S^{-1} \cdot \widetilde{Q} \cdot \widetilde{Y}$$

The respective $\widetilde{Q}$ was provided in Equation (3.25) by

$$\widetilde{Q} = Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdots Q_{P+1}^{(1)} \cdots Q_{P+1}^{(\nu_{P+1})} \cdot Q_1^{(1)} \cdots Q_1^{(\nu_1)} \cdot \cdots Q_P^{(1)} \cdots Q_P^{(\nu_P)}.$$

Algorithm 40 Parallel blocked right-sided application of Q.

function $A_{<p_r,p_c>[i,j]} \leftarrow$ ParallelBlockedRightSidedApplQ($A_{<p_r,p_c>[i,j]}$, $Y_{<p_r,*>[i]}$, $T<*,*>$)
  if (is block column 1) **then**
    if (is block row 1) **then**
      MPI_SEND ($Y_{<p_r,*_1>[i]}$) to block row 2
    else if (is block row 2) **then**
      $Y_{<p_r,*_1>[i]} \leftarrow$ MPI_RECEIVE from block row 1
    end if
  else
    if (is block row 2) **then**
      MPI_SEND ($Y_{<p_r,*_2>[i]}$) to block row 1, 3
    else
      $Y_{<p_r,*_2>[i]} \leftarrow$ MPI_RECEIVE from block row 2
    end if
  end if
  $Y'_{<*,p_c>[j]} \leftarrow$ transpose($Y_{<p_r,*>[i]}$)
  for $i_1 \leftarrow 1, b+1, 2b+1, \ldots, n_b$ **do**
    $i_2 \leftarrow \min(n_b,\ i_1 + b - 1)$
    $j_1 \leftarrow i_1; \qquad j_2 \leftarrow 2n_b$
    $A_{[i,j]}(i_1 : 3n_b, j_1 : j_2)$
        $\leftarrow$ ParallelCWYright($A_{[i,j]}(i_1 : 3n_b, j_1 : j_2)$, $Y'_{[j]}(j_1 : j_2, i_1 : i_2)$, T)
  end for
end function

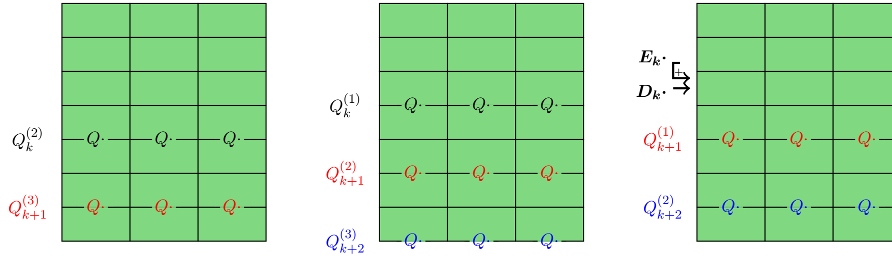


Figure 4.11: Three consecutive steps in the backtransformation: The applications of different Q can be done in parallel in the same way the Q have been created (see Figure 4.4). Finally, the D_k and E_k are applied.

The description will be limited to the case of using a twisted factorization since the non-twisted case is already covered by the lower matrix half part of the twisted case.

Two possibilities exist: setting up a pipelining algorithm as for the transformation of the generalized to the standard eigenvalue problem or accumulating a "backtransformation" matrix $\widetilde{S}^{-1} = S^{-1} \cdot \widetilde{Q}$.

4.3.1 Pipelining approach

Using a pipelining approach results in applying the Householder vectors and inversion steps in the order as given in Equation (3.27) to the matrix of eigenvectors. This means, that this matrix is updated in place without any additional storage requirements. The order of application can also be chosen according to Equation (3.21). Then, inversion steps and orthogonal transformations are not interchanged and are applied as in the transformation to the standard eigenvalue problem.

Since a uniform blocksize has been chosen before, applying the orthogonal transformations and the inversion steps to the eigenvectors is also updating pairs of block rows in the eigenvector matrix.
In particular, applying the orthogonal transformation $Q_k^{(i)}$ to the eigenvector matrix updates the block rows $k+i-1$ and $k+i$. As derived before, all $Q_{k+j}^{(i+j)}$, $j > -i$, update disjunct pairs of block rows $k+i+2j-1 : k+i+2j$ and can be applied in parallel.
When applying inversion step S_k to the eigenvectors, then the block rows $k-1$ and k are updated. This can immediately be carried out after the last $Q_k^{(i)}$ touched the block rows.

The D_k, and E_k as well as the parts to build $Q_k^{(i)}$ can be stored in the PGs they have been used in. This avoids a complete redistribution of this data.
So far, only the block rows have been fixed to the uniform blocksize n_b. For the backtransformation however, a uniform blocksize cannot be used. The reason lies in the arbitrary number of eigenvectors that have to be transformed. When using the same processor setup as for the transformation to the SEP, then three blocks in a block row should also be used here. This can be achieved by splitting the eigenvectors in three groups of roughly same size. Hence, a block in the backtransformation has a size of $n_b \times w_j$ with $j \in \{1, 2, 3\}$.

Figure 4.11 shows the parallel application of different bulge chasing stages and an inversion step. One important difference to the application of inversion steps and bulge chasing stages is that in the backtransformation the application is not transposed. For applying inversion steps, this leads to a different behavior: E_k is multiplied to block row $k-1$ and added to block row k. Afterwards block row k is multiplied by D_k. When applying Q, the only change is instead of T, T^T has to be applied.

Figure 4.12 shows the execution dependencies and the parallel capabilities of the backtransformation using this approach. Again, a wind-up phase is followed by a wind-down phase. For the wind-up phase the length of the pipeline can be computed by

$$l_{\text{pipe}}(i_{\min}) = \lfloor \frac{N - P - i_{\min} + 1}{2} \rfloor, \tag{4.4}$$

for the wind-down phase by

$$l_{\text{pipe}}(k) = \lfloor \frac{N - k + 2}{2} \rfloor. \tag{4.5}$$

Here, $i_{\min}$ refers to the minimum i-index in the row: Similar for k in the wind-down formula, which is obtained by the inversion step carried out.
The pipeline length rises until the last inversion step $P+1$ is applied and then decreases every step of the parallel backtransformation algorithm by one.

Parallel execution of the upper and a lower matrix half

When using a twisted factorization for B and computing the upper matrix half by flipping the matrix and using the lower matrix half algorithm, then this has also to be reflected in the backtransformation.
This means that the matrix of eigenvectors has to be flipped before the upper matrix half backtransformation is applied and afterwards the matrix is flipped back. This can be done in a reduced way since the vectors are independent of each other and hence, only

$$M^{\text{f}}(1 : n, :) = M(n : -1 : 1, :) \tag{4.6}$$

has to be done.

The interference of the upper and lower matrix half has been discussed in Section 4.2.2. The orthogonal transformations only update the matrix half where they origin in: Lower matrix half orthogonal transformations will never update the upper matrix half and vice versa. This holds also for the backtransformation.
Figure 4.6 provides an illustration of the situation for the inversion steps: The last inversion step in the upper matrix half updates $A_{P+1,P}$ in the lower matrix half. This update however, is conducted by the right-sided update. The left-sided update does not update any block in the lower matrix half. The last inversion step in the lower matrix half updates $A_{P,P}$ in the upper matrix half. This update happens by the left and the right-sided application of the inversion step. Hence, this update in the upper matrix half has also to be carried out in the backtransformation.
The interference of the upper and lower matrix half during backtransformation is illustrated

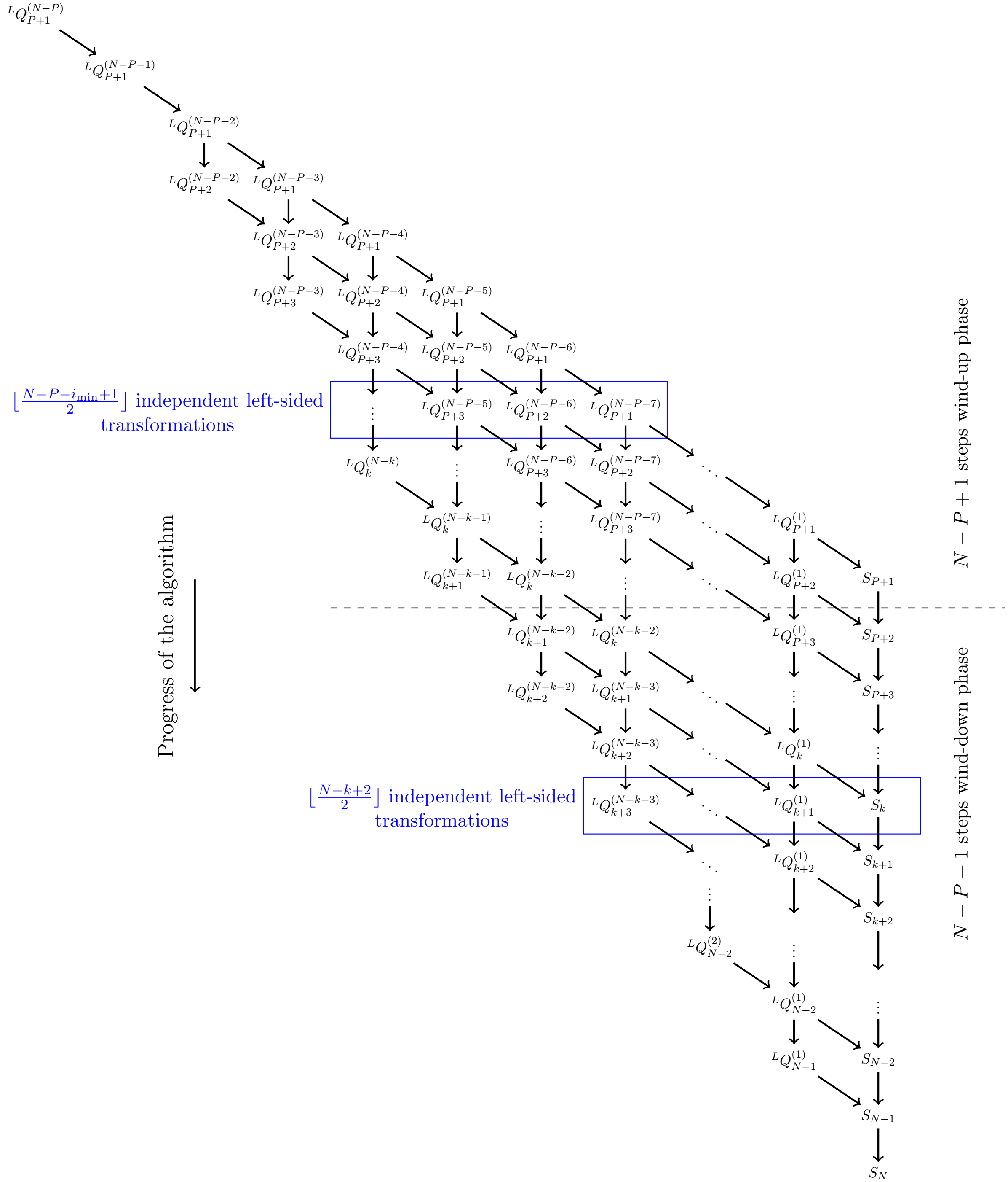


Figure 4.12: Execution dependencies in the backtransformation of the eigenvectors: Every diagonal shows one inversion step and the regarding stages of bulge chasing. To line out the left-sided updates, ${}^{L}Q_k^{(i)}$ is used for the left-sided application of the orthogonal transformation $Q_k^{(i)}$ (analogous to Figure 4.5). All entries in one row are independent and can be executed at the same time. The number of independent operations can be computed by the given formulas. In the wind-up phase, $i_{\min}$ refers to the minimum i-index in the row, k refers in the wind-down formula to the inversion step which is applied in this row.

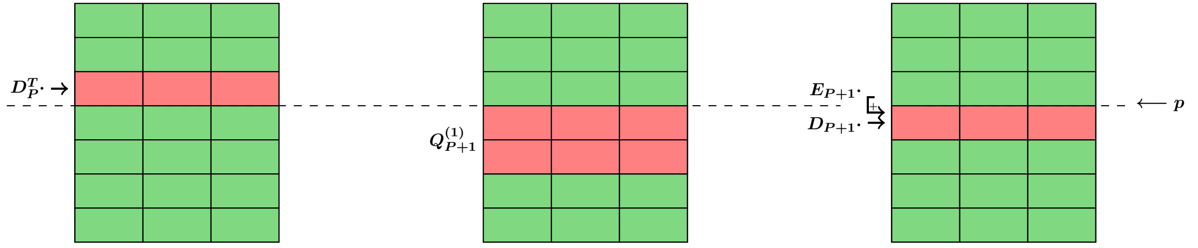


Figure 4.13: Application of the last inversion steps of the upper and lower matrix half to the eigenvector matrix and their dependencies. The application of the last inversion step of the lower matrix half, S_{P+1}, in shown in the right picture. It has to be carried out after the last inversion step in the upper matrix half and after applying $Q_k^{(1)}$ (left and middle pictures).

in Figure 4.13. The serial order is determined by Equations (3.26) and (3.27). For the parallel backtransformation, the order of the operations applied to a block row has still to be kept. Thus, the last inversion step of the upper matrix half, S_P, updates block row P. This block row will be used later by the last inversion step of the lower matrix half, S_{P+1}. Before applying S_{P+1} to the matrix, however, first $Q_{P+1}^{(1)}$ has to be applied since it also updates block row $P+1$ and the application of $Q_k^{(i)}$ comes first in Equation (3.26). Finally, S_{P+1} can be applied using the last block row of the upper matrix half.

Besides this synchronization point, both matrix halves are fully independent and can thus be applied in parallel to the eigenvectors. This allows also to use the splitting of the processes in upper matrix half processes and lower matrix half processes.

Algorithm 41 provides the parallel algorithm for the backtransformation of eigenvectors in the lower matrix half. By flipping the matrix this algorithm can also be used for the upper matrix half. The exchanging of data has been simplified in the algorithm. It only describes the view of the lower matrix half where the block row P of the upper matrix half is received before inversion step S_{P+1} is applied. The sending of the data in the upper matrix half, however, is performed after inversion step S_P is applied. The simplification was done for the sake of clarity.

Process and data structures

During the backtransformation the processes stay assigned in process groups and split in the upper and lower matrix half as before. The assignment in process group columns is not used here but the assignment in process group rows. This, however, does not coincide with a redistribution of blocks.

Figure 4.14 shows on the left the block structure as used during the twisted Crawford algorithm. There, three blocks form a block column. In the same way, three blocks form a block row. The block rows are as well as the block columns arranged in a staircase way. The blocks in a BC can be numbered from 1 (diagonal block in the BC) to 3 (sub-sub-diagonal block in the BC). The block index for blocks in a block row can be computed by $i_{\text{indexBR}} = 4 - i_{\text{indexBC}}$ when using the index in the block column i_{indexBC}.

Algorithm 41 Backtransformation of eigenvectors.

if (process has local data in BR N) **then**
    apply $Q_{P+1}^{(N-P+1)}$ from left to BR N
end if
$k \leftarrow P + 1$
$j \leftarrow N$
windupPhase $\leftarrow$ true
while $j \leq N$ **do**
    $i \leftarrow 1 + (j - k)$; $i_0 \leftarrow 1 + (j - k)$
    for $r_l \leftarrow j + 1, j + 3, ..., N$ **do**
        if (process has local data in BRs $r_l - 1 : r_l$) **then**
            apply $Q_{k+i-i_0}^{(i)}$ from left to BRs $r_l - 1 : r_l$
        end if
        $i \leftarrow i + 1$
    end for
    if (j=k **and** (process has local data in BC $k - 1 : k$)) **then**
        if ($k = P + 1$) **then**
            exchange data with other matrix half
        end if
        apply D_k, E_k to BC k, use BC $k - 1$
    end if
    if ($j = P + 1$) **then**
        windupPhase = false
    end if
    if (windupPhase) **then**
        $j \leftarrow j - 1$
    else
        $k \leftarrow k + 1$; $j \leftarrow j + 1$
    end if
end while

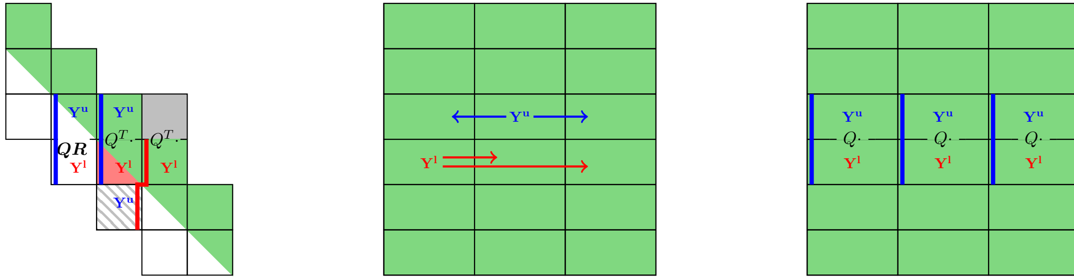


Figure 4.14: Redistribution of $Y_{<p_r,*>}$ for the backtransformation: (left) Initial distribution of Y with splitting $Y = [Y^u\ Y^l]^T$ in an upper and an lower part as stored during the left-sided orthogonal transformation. (middle) Communication of Y^u and Y^l within the block rows. (right) Final distribution of Y^u and Y^l as needed for the backtransformation. The thick lines illustrate the communicators along which Y will be used in the backtransformation.

For using the block row view during the backtransformation, all blocks with the same block row index are put in a column. This can be seen as pushing from left to the block rows until they are aligned on the right.

This setup does not change the processes within blocks and not the data distribution. It solely creates new connections between the block rows. In the twisted Crawford algorithm, during the left-sided update, block two in the upper block row was co-working with block one in the lower block row. In the backtransformation, only blocks of neighboring block rows with the same block row index will co-work.

Block and inter-block parallelization

For applying the inversion steps and the orthogonal transformations to the matrix of eigenvectors, the necessary data has to be in place. For inversion steps this requires that D and E are available in every block of the regarding block row. For the left-sided application of the orthogonal transformation, Y and T have to be in place respectively.

Y and τ of $Q_k^{(i)}$ are computed in the blocks $A_{k+i-1:k+i,k+i-2}$ and are used by the blocks $A_{k+i-1:k+i,k+i-1}$, $A_{k+i,k+i}$ and the surrogate block $A_{k+i+1,k+i-1}$. The blocks that compute Y and τ will also store Y and T for the backtransformation. It is possible to store them additionally in the blocks that use it during the left or right-sided application but this would again increase the memory footprint drastically.

Hence, Y and T are only stored in the blocks $A_{k+i-1:k+i,k+i-2}$. When considering the backtransformation block layout, then the upper part of Y is available in block two of the block row, the lower part is available in block one. Figure 4.14 illustrates the original distribution and the communication of the Y parts. On the right, additionally, the necessary inter-block communicators are indicated.

A similar situation can be found for applying inversion steps in the backtransformation.

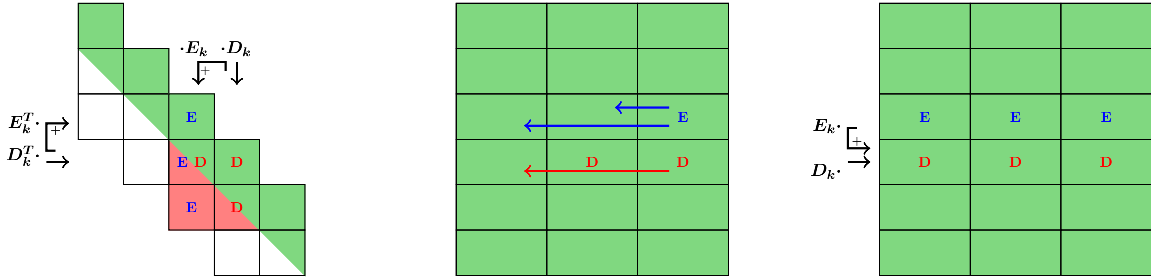


Figure 4.15: Redistribution of D and E for the backtransformation: (left) Initial distribution when applying the inversion step. (middle) Communication of D and E within the block rows. (right) Final distribution of D and E as needed for the backtransformation.

Initially, D_k and E_k are available in every block they have been used in during the inversion step (Figure 4.15 left picture). Since E_k has to be available in all three blocks of block row $k-1$, it has to be communicated to block one and two. Slightly different is the situation with D_k which is already available in block two and three of block row k. Here, block three is used to communicate D_k to block one. Block three is used instead of block two because when it comes to applying inversion step P, block two does not necessarily hold D_P.

Parallel algorithms on the level of pairs of block rows and block columns

Algorithm 41 described the coarse grained view on the algorithms stating left-sided applications of $Q_k^{(i)}$ and S_k to the matrix of eigenvectors. The necessary communication to distribute the data has been described before. Now, the single algorithms working on the level of pairs of block rows are going to be described.

To perform a QR decomposition and the left-sided application of Q, ParallelCWYleft was used. For the backtransformation, this cannot be used since it applies Q^T instead of Q. To apply Q from left instead of Q^T, the compact WY representation has to be revised. The application of Q^T is given by

$$\widetilde{M} = Q^T \cdot M = (I - Y \cdot T \cdot Y^T)^T \cdot M = (I - Y \cdot T^T \cdot Y^T) \cdot M.$$

Hence, the non-transposed application of Q is given by

$$\widetilde{M} = Q \cdot M = (I - Y \cdot T \cdot Y^T) \cdot M. \tag{4.7}$$

The only difference is using T in a transposed way or not. The algorithm for the blocked application of the Householder vectors from left in a non-transposed way is given by Algorithm 42.

As before in the transformation from the generalized to the standard eigenvalue problem, for backtransformation, Householder vectors are applied in a blocked way tile-wise. Algorithm 43 provides the algorithm for applying the Householder vectors by tiles of size b. It employs the ParallelCWYtransposedLeft as described in algorithm Algorithm 42 for the blocked ap-

Algorithm 42 Parallel blocked transposed application of Householder vectors from left.

function $A_{<p_r,p_c>[i,j]} \leftarrow$ ParallelCWYtransposedLeft($A_{<p_r,p_c>[i,j]},\ Y_{<p_r,*>[i]},\ T_{<*,*>}$)
  $Z_{<p_r,*>[i]} \leftarrow Y_{[i]} \cdot T^T$
  $U_{<*,p_c>i[j]} \leftarrow Z_{[i]}^T \cdot A_{[i,j]}$
  $U_{<*,p_c>[j]} \leftarrow$ MPI_ALLREDUCE ($U_{<*,p_c>i[j]}$)
  $A_{[i,j]} \leftarrow A_{[i,j]} - Y_{[i]} \cdot U_{[j]}$
end function

plication. If the T have not been stored during the QR generation, then they have to be recomputed by Algorithm 35.

Algorithm 43 Parallel blocked non-transposed application of orthogonal transformations.

function $X_{<p_r,p_c>[i,j]} \leftarrow$ ParallelBlockedLeftSidedApplQ($X_{<p_r,p_c>[i,j]}, Y_{<p_r,*>[i]}, T< *, * >$)
  for $j_1 \leftarrow 1, b+1, 2b+1, \dots, n_b$ **do**
    $i_1 \leftarrow j_1$; $i_2 \leftarrow 2n_b$
    $j_2 \leftarrow \min(n_b,\ j_1 + b - 1)$
    $X_{[i,j]}(i_1 : i_2, :) \leftarrow$ ParallelCWYtransposedLeft($X_{[i,j]}(i_1 : i_2, :),\ Y_{[i]}(i_1 : i_2, j_1 : j_2),\ T$)
  end for
end function

Algorithm 44 gives the algorithm for applying an inversion step to the eigenvector matrix. The method ParallelDEbacktrafo is called by processes of the two block rows that have to apply the inversion step. First, the upper block row is multiplied by E and the result is communicated to the lower block row. The lower block row is first multiplied by D and to the result the communicated result of the upper block row is added. The communication is organized in a way that a process transfers its local data to the process in the other block at the same place in the process grid.

Algorithm 44 Parallel application of an inversion step during the backtransformation.

function $X_{<p_r,p_c>[i,j]} \leftarrow$ ParallelDEbacktrafo($X_{<p_r,p_c>[i,j]},\ D_{<p_r,p_c>[i,j]},\ E_{<p_r,p_c>[i,j]}$)
  if (process has local data in upper BR) **then**
    $b_{<p_r,p_c>[i,j]} \leftarrow$ PDGEMM ($E_{[i,j]},\ X_{[i,j]}$)
    MPI_SEND ($b_{<p_r,p_c>[i,j]}$) to lower block row
  else
    $X_{[i,j]} \leftarrow$ PDTRMM ($D_{[i,j]},\ X_{[i,j]}$)
    $b_{<p_r,p_c>[i,j]} \leftarrow$ MPI_RECEIVE from upper block row
    $X_{[i,j]} \leftarrow X_{[i,j]} + b_{[i,j]}$
  end if
end function

4.3.2 Backtransformation matrix approach

Equation (3.19) with

$$X = \widetilde{S}^{-1}\widetilde{Y}$$

together with Equation (3.21) using the order of application

$$\begin{aligned}\widetilde{S}^{-1} = S_N^{-1} \cdot Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdots S_{P+1}^{-1} \cdot Q_{P+1}^{(1)} \cdots Q_{P+1}^{(\nu_{P+1})} \\ \cdot S_1^{-1} \cdot Q_1^{(1)} \cdots Q_1^{(\nu_1)} \cdots S_P^{-1} \cdot Q_P^{(1)} \cdots Q_P^{(\nu_P)}\end{aligned}$$

or Equation (3.23) for the interchanged order

$$\begin{aligned}\widetilde{S}^{-1} = S_N^{-1} \cdots S_{P+1}^{-1} \cdot S_1^{-1} \cdot \cdots S_P^{-1} \\ \cdot Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdots Q_{P+1}^{(1)} \cdots Q_{P+1}^{(\nu_{P+1})} \cdot Q_1^{(1)} \cdots Q_1^{(\nu_1)} \cdot \cdots Q_P^{(1)} \cdots Q_P^{(\nu_P)}\end{aligned}$$

provide the computations to obtain the original eigenvectors. The pipelining approach applies the orthogonal transformations and the inversion steps directly to the eigenvector matrix. Another possibility to perform the backtransformation is to accumulate the orthogonal transformations and inversion steps to obtain the backtransformation matrix $\widetilde{S}^{-1}$ explicitly. This matrix is then multiplied with the eigenvectors to obtain the original eigenvectors. The parallel implementation of this approach will be presented in the following.

The starting point is to set up an identity matrix before running the twisted Crawford algorithm and, when applying inversion steps or orthogonal transformations from right to A, also multiplying the backtransformation matrix from right. The inversion steps can be applied all at once (following Equation (3.23)) or in the same way they are applied to A (following Equation (3.21)). Hereinafter, the latter will be used as applying the inversion steps one by one perfectly follows the application of inversion steps to A. Computing S^{-1} before starting the transition from GEP to SEP as well as applying the inversion steps afterwards from left to the product of the orthogonal transformation would lack in parallelism (since two consecutive inversion steps overlap by one block row or column).

Parallel execution of the upper and a lower matrix half

When using a twisted factorization, the interference of the upper and lower matrix half has to be investigated to outline the necessary data exchange between the upper and lower matrix half.
At first, the application of an inversion step is reviewed. The right-sided application of S_k^{-1} multiplies block column k by E_k and adds the result to block column $k-1$. Afterwards, block column k is updated by its multiplication by D_k. Hence, block columns $k-1$ and k are updated.
Since starting with an identity matrix, everything but the diagonal is zero. For simplicity, the backtransformation matrix can be seen as a dense matrix which can be subdivided in a $N \times N$ grid of blocks of size n_b (the last block row and column can be smaller if the matrix size is not a multiple of n_b).
The application of inversion step k introduces additional fill-in in block column $k-1$, which is restricted to the block rows k till N. Thus, the fill-in by the inversion steps grows every step

but never exceeds the lower matrix half. The most left fill-in can be found in block column P. The application of the orthogonal transformation $Q_k^{(i)}$ from right updates the block columns $k+i-1$ and $k+i$. If one of the blocks has been filled before, both will be afterwards. $Q_k^{(1)}$ updates block columns k and $k+1$, which means that the block right of the diagonal will be filled. $Q_k^{(2)}$ will additionally fill block row $k+2$ and so on. Finally, all block columns right of the diagonal block will be filled starting from block row k.
Finally, a square in the lower right quadrant of the matrix is introduced. It starts from block column P and block row $P+1$.

After the lower matrix half, the upper matrix half has to be applied to the backtransformation matrix. The situation is the same, just flipped, until inversion step P. It does not have the E part and hence, only updates block column P. In this BC, however, the lower matrix half blocks are already filled. Hence, also the blocks in the lower matrix half have to be updated by D_P. Besides this "distant" update, only the upper matrix half is updated and also here a square in the top left quadrant of the matrix is introduced. Thus, only D_P has to be exchanged from the upper to the lower matrix half.

To be able to use only the lower matrix half algorithm as in the transition from GEP to SEP, the idea of flipping has also to be available for the backtransformation matrix. The initial matrix is an identity matrix and can hence easily be flipped. The application of inversion steps and orthogonal transformations can be translated in the same way as it is done for the applications to A (see Section 3.1.1 for details). Hence, flipping can be used to assemble the backtransformation matrix and a formulation of the algorithm in the lower matrix half is sufficient. For the flipping, the simplified version given in Equation (4.6) can be used. Since starting with an identity matrix, only flipping back is necessary. The identity matrix can already be set up in the flipped way.

As before, both matrix halves can be executed in parallel. The update of the lower matrix half by D_P does not even cause a synchronization since this transfer can be overlapped by computations as the application of D_P in the lower matrix half is only restricted to happen after applying S_{P+1}.
Since following the algorithmic structure of the right-sided update, the parallelism between pairs of block columns is the same as in the transition from GEP to SEP.

Algorithm 45 provides the extended version of Algorithm 30 that additionally generates the backtransformation matrix $\widetilde{S}^{-1}$ in passing. The right-sided application of the orthogonal transformations and the application of the inversion steps have been supplemented by the update of the backtransformation matrix $\widetilde{S}^{-1}$. The exchange of D_P is only mentioned for the lower matrix half since the sending from the upper matrix half can be done at any time.

Process and data structures

During the transition from GEP to SEP the blocks are arranged in a staircase way and a block column once computes together with the left neighboring BC, once with the right neighboring BC. This pattern is followed in the generation of the backtransformation matrix as illustrated in Figure 4.16. The middle and the right pictures show the application of Q in two consecutive

Algorithm 45 Twisted Crawford Algorithm with accumulation of backtransformation matrix.

$\widetilde{S}^{-1} \leftarrow I$
$j \leftarrow N; \quad k \leftarrow N$
windupPhase $\leftarrow$ true
while $j \leq N$ **do**
  if (windupPhase **and** (process has local data in BC $k-1:k$)) **then**
    apply D_k, E_k to BCs $k-1:k$ of A
    apply D_k, E_k from right to BCs $k-1:k$ of $\widetilde{S}^{-1}$
    if ($k = P+1$) **then**
      exchange data of A with other matrix half
      if (lower matrix half) **then**
        exchange D_P with other matrix half and apply D_P to BC P
      end if
    end if
  end if
  $i \leftarrow 1 + (j-k); \quad i_0 \leftarrow 1 + (j-k)$
  for $r_l \leftarrow j+1, j+3, ..., N$ **do**
    if (process has local data in BRs $r_l - 1 : r_l$) **then**
      generate $Q^{(i)}_{k+i-i_0}$ and apply from left to BRs $r_l - 1 : r_l$
    end if
    $i \leftarrow i+1$
  end for
  $i \leftarrow 1 + (j-k); \quad i_0 \leftarrow 1 + (j-k)$
  for $c_r \leftarrow j+1, j+3, ..., N$ **do**
    if (process has local data in BCs $c_r - 1 : c_r$) **then**
      apply $Q^{(i)}_{k+i-i_0}$ from right to BCs $c_r - 1 : c_r$ of A
      apply $Q^{(i)}_{k+i-i_0}$ from right to BCs $c_r - 1 : c_r$ of $\widetilde{S}^{-1}$
    end if
    $i \leftarrow i+1$
  end for
  if ($j = P+1$) **then**
    windupPhase $\leftarrow$ false
  end if
  if (windupPhase) **then**
    $k \leftarrow k-1; \quad j \leftarrow j-1$
  else
    $j \leftarrow j+1$
  end if
end while
if (process has local data in BR N) **then**
  generate $Q^{(N-P+1)}_{P+1}$ and apply from left to BR N
end if
if (process has local data in BC N) **then**
  apply $Q^{(N-P+1)}_{P+1}$ from right to BC N of A
  apply $Q^{(N-P+1)}_{P+1}$ from right to BC N of $\widetilde{S}^{-1}$
end if

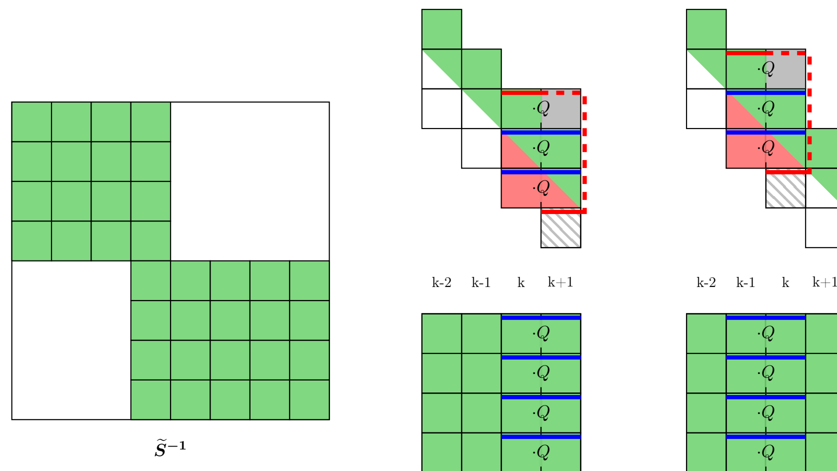


Figure 4.16: Shape of the backtransformation matrix and two steps of the combined right-sided update / backtransformation matrix update: (left) Non-zero pattern of the backtransformation matrix. (middle upper) right-sided application of an orthogonal step (middle lower) Regarding update of the backtransformation matrix. (right upper and lower) The following right-sided transformation and backtransformation matrix update. Given in blue and red are the necessary communicators between the blocks.

stages of the bulge chasing. The left picture shows the structure of the backtransformation matrix, which consists of the upper left quadrant and the extended lower right quadrant as described before.
In Figure 4.16, the block columns are already subdivided into quadratic blocks. When doing so, the blocks can be distributed cyclically over the process groups of the owning process group column. Thus, there might be an imbalance of one block between the process groups.
Another possibility is to distribute the rows of the blocks to three more or less equal sized blocks and distribute this blocks to the three process groups. An advantage of this approach is the more equal distribution of the rows of the backtransformation matrix (imbalance of one row between the process groups). However, by giving up the uniform blocksize, computational routines written for uniform blocks have to be rewritten.

The block columns of the backtransformation matrix are distributed in the same way to the process group columns as the block columns of matrix A. Since the doubled block column P does also exist in A (see Figure 4.6), this does not cause problems.

To omit an additional synchronization point between the process groups of a pair of process group columns, the blocks of the backtransformation matrix have to be distributed carefully to the process groups. It has to be achieved that the process groups collaborating when updating A are also collaborating when updating $\widetilde{S}^{-1}$. An assignment pattern achieving this is given in Figure 4.17 in the right illustration. It repeats every three block columns and the collaboration partners of the process groups are the same as in the left and middle illustration.

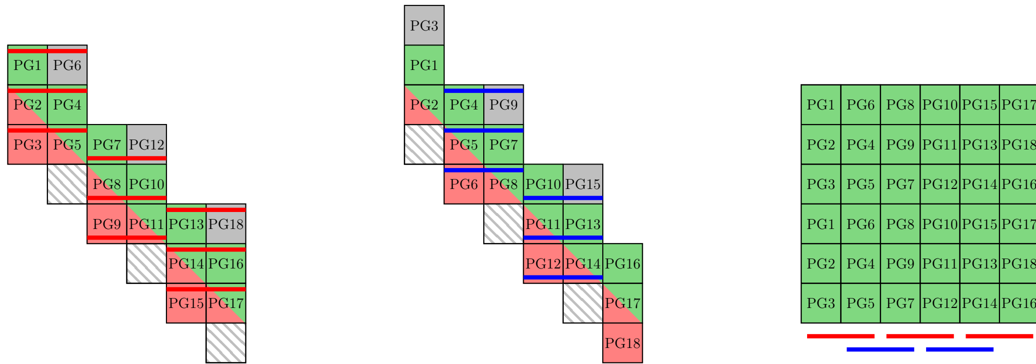


Figure 4.17: Collaboration of the blocks and process groups in the right-sided update during two consecutive pipeline steps (left and middle) and block to process group assignment in the regarding part of the backtransformation matrix to achieve interaction of the same process groups as in the right-sided updates (right). The blue and red lines indicate the collaboration (and the regarding communicator).

Reducing computational work by considering the zero patterns

The backtransformation matrix is initially an identity matrix. When applying inversion step k, then all $\widetilde{S}^{-1}_{k:N,k-1:k}$ are updated. All blocks above are not changed and can hence be neglected. Inversion steps introduce fill-in below the diagonal and stepwise fill up to the left. The application of Householder transformations generates fill-in above the diagonal. When following the order in Equation (3.21), then after the application of S_k, no fill-in exists in block rows smaller than k. The fill-in above the diagonal is introduced stepwise and has a staircase shape during the wind-up phase. This can be exploited when applying the Householder transformations. In the wind-down phase the quadrant is filled completely and the full matrix half has to be updated.
Another possibility to save computational work is the application of E. In the original inversion step application, the right BC is multiplied by E and added to the left BC. When assembling the backtransformation matrix, however, the left BC is zero all the time and hence the addition can be saved.
During the multiplication of $\widetilde{S}^{-1}$ with the matrix of eigenvectors, obviously the zero pattern of $\widetilde{S}^{-1}$ can be exploited.

Parallel algorithms on the level of pairs of block rows and block columns

Algorithm 45 provided the extended algorithm that applies inversions steps and right-sided Householder transformations to A and additionally generates the backtransformation matrix $\widetilde{S}^{-1}$.

To apply the Householder transformations, Algorithm 39 can be used. Since the application to A is done with the same process group setup, all data is already in place and no additional data transfer is necessary.
For applying the inversion steps, Algorithm 46 provides a parallel version. Comparing to Algorithm 31, this version is much shorter since it does not have to handle non-existing

blocks in the upper triangle of the matrix or similar. However, compared to the original data distribution, also block three in the right block column has to hold D.

Algorithm 46 Parallel application of an inversion step to the backtransformation matrix.

function $\widetilde{S}^{-1}_{<p_r,p_c>[i,j]} \leftarrow$ ParallelDEapplication($\widetilde{S}^{-1}_{<p_r,p_c>[i,j]},\ D_{<p_r,p_c>[i,j]},\ E_{<p_r,p_c>[i,j]}$)
  if (process has local data in right BC) **then**
    $b_{<p_r,p_c>[i,j]} \leftarrow \widetilde{S}^{-1}_{[i,j]}$
    MPI_SEND ($b_{<p_r,p_c>[i,j]}$) to neighbor block in left BC
    $\widetilde{S}^{-1}_{[i,j]} \leftarrow$ PDTRMM ($\widetilde{S}^{-1}_{[i,j]},\ D_{[i,j]}$)
  else
    $b_{<p_r,p_c>[i,j]} \leftarrow$ MPI_RECEIVE from neighbor block in right BC
    $\widetilde{S}^{-1}_{[i,j]} \leftarrow$ PDGEMM ($b_{[i,j]},\ E_{[i,j]}$)
  end if
end function

The multiplication can be performed in various ways. One possibility is to generate a ScaLAPACK distributed version of $\widetilde{S}^{-1}$ and multiply it with the matrix of eigenvectors. Another possibility is to generate two ScaLAPACK distributed matrices representing the upper left and lower right quadrant of non-zeros in $\widetilde{S}^{-1}$ and apply them to the eigenvector matrix. A third option would be to do the multiplication in the data distribution format of $\widetilde{S}^{-1}$. This variant, however, causes high effort for the transformation between the data distributions since the eigenvectors have to be transformed twice.

4.3.3 Summary

For the pipelining approach, at least Y and the τ have to be stored. In Table 3.1 the number of chasing steps was given with $0.25N^2 + 0.5N$. This means that even when ignoring the repetition along the columns of a process group, $0.5N^2 + N$ blocks of storage are required (assuming $P = \frac{N}{2}$). For including the repetition, this number has to be multiplied by p_c. Instead of the τ, T can also be stored. This allows to save computations but costs additional storage.
The backtransformation matrix approach requires $P^2 + (N - P + 1)(N - P)$ blocks of storage to hold the matrix. Simplifying with $P = \frac{N}{2}$, $0.5N^2 + 0.5N$ blocks are necessary. Hence, assembling the backtransformation matrix is slightly less demanding regarding storage. So far, these considerations have been made on a non-distributed view of data. Including data distribution and the immanent repetition of some data, the pipelining approach becomes even less competitive.

To compare the Flops, all necessary steps have to be evaluated. This means for the backtransformation matrix approach to consider the multiplication of the matrix with the eigenvector matrix, the application of the inversion steps and the application of the Householder transformations. For the pipelining approach, the application of the inversion steps and the application of the Householder vectors have to be taken into account.

For the following Flop computations, $n = Nn_b$, $P = \frac{N}{2}$ and $p = Nn_b$ are fixed. The tile size when applying Householder vectors is given by b. The backtransformation of w eigenvectors via the pipelining approach in a serial setting causes computational costs of $(1.5 - \frac{1}{4n_b} + \frac{b}{2n_b})n^2w + O(n^2)$ Flops. When utilizing the backtransformation matrix, then $(0.5 + \frac{b}{6n_b} - \frac{1}{12n_b})n^3 + n^2w + O(n^2)$ Flops are necessary for the backtransformation step.

It can be seen that only when transforming almost all eigenvectors, the backtransformation matrix approach becomes equally expensive (both approximately $1.5n^3$). However, in the most cases not all eigenvectors are required and hence the pipeline approach will usually be favorable.

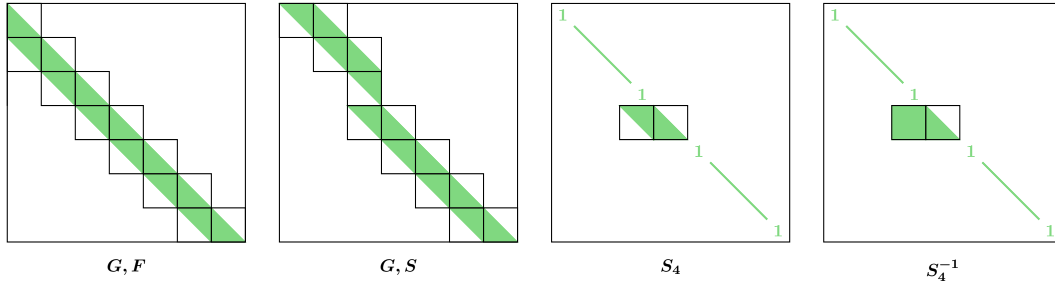


Figure 4.18: (first picture) Block structure of matrices with uniform bandwidth as it appears in F and can appear in G when not using a twisted factorization for G. (second picture) Block structure of the twisted factorization as it appears in S and can appear in G. (third picture) One of the partial factors of S following Equation (3.13), S_4, and the inverse of it, S_4^{-1} (forth picture).

4.4 Crawford SVD algorithm

4.4.1 Pipelining approach

In the same way as in Section 4.2.1, the pipelining approach can be motivated for the Crawford SVD algorithm. Contrary to the eigenvalue algorithm the orthogonal transformations and the inversion steps are not applied from left and right. Half of the orthogonal transformations are applied only from left, half only from right. The inversion steps are solely applied from right. The use of the uniform block $n_b = b_A = b_B$ as justified in Section 4.2.1 for the eigenvalue algorithm allows also in the SVD variant the use of an efficient pipelining approach. Then, all operations are performed on the block level and one bulge chasing stage moves the bulge by n_b rows or columns towards the end of the matrix.

In Figure 4.18 the simplified block structure of the matrices G, F and S is presented. Again, a simplified notation for indices on the block level becomes available. As declared in Section 4.2.1, $M_{k,k+1}$ refers to the block in block row k and block column $k + 1$, which is located in the rows $(k - 1)n_b + 1 : kn_b$ and the columns $kn_b + 1 : (k + 1)n_b$.

Contrary to the eigenvalue algorithm, where flipping allowed to limit all descriptions to the lower matrix half, this is not possible for the Crawford SVD algorithm. It can only be applied to the case of having G and S in twisted shape. Hence, only in this case the descriptions are limited to the lower matrix half. Additionally, the SVD algorithm is not operating on symmetric matrices. This requires to describe the operations in the lower as well as the upper triangle of the matrix G.

Similarly to the Crawford eigenvalue algorithm also for the SVD algorithm a brief description using the simplified block structure shall be given. The application of inversion step S_k with $k > P$ (lower matrix half) updates the blocks $M_{k:k+1,k-1:k}$. The blocks $M_{k:k+1,k}$ are multiplied by E_k and added to $M_{k:k+1,k-1}$. Afterwards, $M_{k:k+1,k}$ are multiplied by D_k. This introduces fill-in in the lower triangle in the blocks $M_{k:k+1,k-1}$ and $M_{k+1,k}$ (see Figure 4.19 left illustration). A QR decomposition of the blocks $M_{k:k+1,k-1}$ is computed and the obtained $Q_k^{(1)}$ applied from left to the block rows $k : k + 1$. This removes the fill-in in

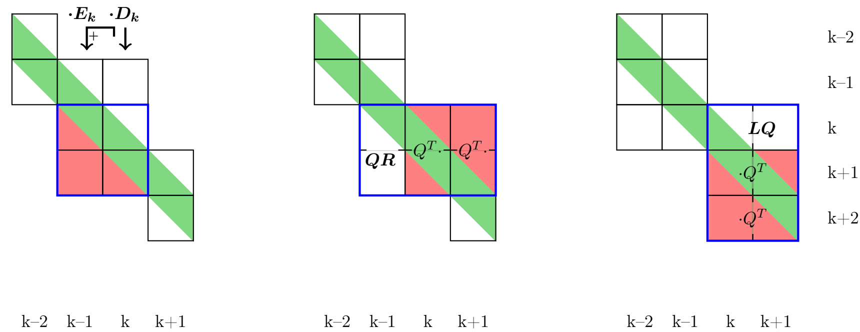


Figure 4.19: Application of an inversion step in the lower matrix half: (left)The application of inversion step k updates the entries within the blue rectangle. Marked in red is the bulge of newly created non-zeros outside the band. (middle) QR decomposition of the left bulge blocks of the bulge and left-sided application of the obtained Q. This generates new fill-in above the diagonal. (right) LQ decomposition of the newly generated fill-in and right-sided application of the obtained Q. New fill-in is introduced below the diagonal. (middle and right) The updated blocks are marked with a blue rectangle. Q^T on the border of two blocks indicate that this operation involves the two blocks.

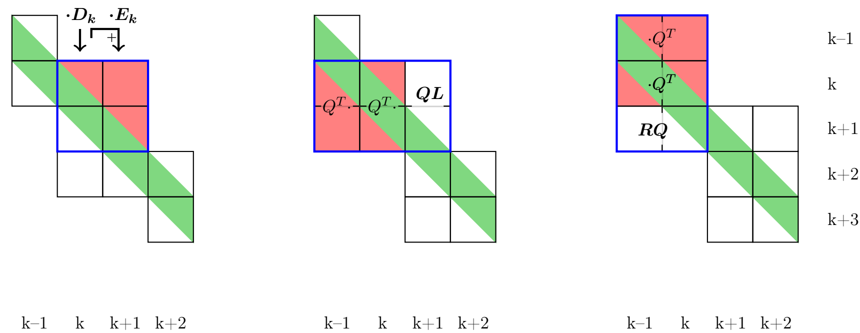


Figure 4.20: Application of an inversion step in the upper matrix half (with untwisted G): (left)The application of inversion step k updates the entries within the blue rectangle. Marked in red is the bulge of newly created non-zeros outside the band. (middle) QL decomposition of the right bulge blocks and left-sided application of the obtained Q. This generates new fill-in below the diagonal. (right) RQ decomposition of the newly generated fill-in and right-sided application of the obtained Q. New fill-in is introduced above the diagonal. (middle and right) The updated blocks are marked with a blue rectangle. Q^T on the border of two blocks indicate that this operation involves the two blocks.

$M_{k:k+1,k-1}$ and introduces new fill-in above the diagonal in the blocks $M_{k:k+1,k+1}$ and $M_{k,k}$. The fill-in in $M_{k+1,k}$ remains since by the following LQ and the regarding right-sided update it would be filled up again anyway. A LQ decomposition is computed of $M_{k,k:k+1}$ and the obtained $Q_k^{(2)}$ is applied from right to the block columns $k : k+1$, which introduces again fill-in below the diagonal in the blocks $M_{k+1:k+2,k}$ and $M_{k+2,k+1}$. Thus, the bulge moved by every QR application by one block to the right and by every LQ application by one block towards the bottom of the matrix. This procedure is applicable in the lower matrix half and for the case of twisted G when exploiting the flipping idea as well in the flipped upper matrix half.

In general, when mentioning the upper matrix half without additional note, it can be assumed that G is in non-twisted shape.
The application of inversion step S_k in the upper matrix half updates the blocks $M_{k:k+1,k:k+1}$. Similar as in the lower matrix half, block column k is multiplied by E and added to block column $k+1$. Afterwards, block column k is multiplied by D_k. This application of the inversion step introduces fill-in above the diagonal in the blocks $M_{k:k+1,k+1}$ and $M_{k,k}$. The chasing has now to be performed towards the top left end of the matrix and hence a QL decomposition of $M_{k:k+1,k+1}$ is computed. The obtained $Q_k^{(1)}$ is applied to the block rows $k : k+1$, removes the fill-in in $M_{k:k+1,k+1}$ and introduces new fill-in below the diagonal in the blocks $M_{k:k+1,k-1}$ and $M_{k+1,k}$. Again, one block of fill-in ($M_{k,k}$) is not cleared since it would be filled up in the following RQ step anyway. The RQ step is computed on $M_{k+1,k-1:k}$ and applied from right to the block columns $k-1 : k$. This introduces new fill-in above the diagonal (blocks $M_{k-1,k-1:k}$ and $M_{k,k}$) but clears the blocks the RQ was computed on. Thus, this sequence moved the bulge by one block row and one block column towards the top left end of the matrix. An illustration is given in Figure 4.20.

Pairs of block rows or pairs of block columns can again be executed independently of each other. In the lower matrix half, the application of $Q_k^{(i)}$, $i = 1+2j$, generated by a QR decomposition, updates the block rows $k + \frac{i-1}{2} : k + \frac{i-1}{2} + 1$. In the upper matrix half, $Q_k^{(i)}$ with $i = 1+2j$ (generated by a QL decomposition) updates the block rows $k - \frac{i-1}{2} : k - \frac{i-1}{2} + 1$. From this expressions the independence of all $Q_{k+j}^{(1+2j)}$, $0 \leq j \leq \lfloor \frac{N-k}{2} \rfloor$, in the lower matrix half and all $Q_{k-j}^{(1+2j)}$, $0 \leq j \leq \lfloor \frac{k-1}{2} \rfloor$ in the upper matrix half can be derived. This means that after applying inversion step S_k, all $Q_{k+j}^{(1+2j)}$ generated by QR or, respectively, all $Q_{k-j}^{(1+2j)}$ generated by QL, can be executed in parallel.
Analogously, the independence of the right-sided transformation with RQ and LQ can be derived. $Q_k^{(i)}$, $i = 2j$, generated by a LQ decomposition updates the block columns $k + \frac{i}{2} - 1 : k + \frac{i}{2}$. In the upper matrix half, $Q_k^{(i)}$, $i = 2j$, generated by a RQ decomposition updates the block columns $k - \frac{i}{2} : k - \frac{i}{2} + 1$. The independence of all $Q_{k+j}^{(2j)}$, $1 \leq j \leq \lfloor \frac{N-k}{2} \rfloor$, in the lower matrix half and all $Q_{k-j}^{(2j)}$, $1 \leq j \leq \lfloor \frac{k-1}{2} \rfloor$ in the upper matrix half can be derived from that. The application of inversion step S_k in the lower matrix half updates the block columns $k-1 : k$ and is therefore independent of the application of all $Q_{k+j}^{(2j)}$. Similar for the upper matrix half where inversion step k updates the block columns $k : k+1$ and the independence of the applications of $Q_{k-j}^{(2j)}$ is given. This means that together with inversion step S_k, all $Q_{k+j}^{(2j)}$ generated by LQ or, respectively, all $Q_{k-j}^{(2j)}$ generated by RQ can be executed in parallel.
Figures 4.21 and 4.22 illustrate the independence of the operations and outline the pipeline

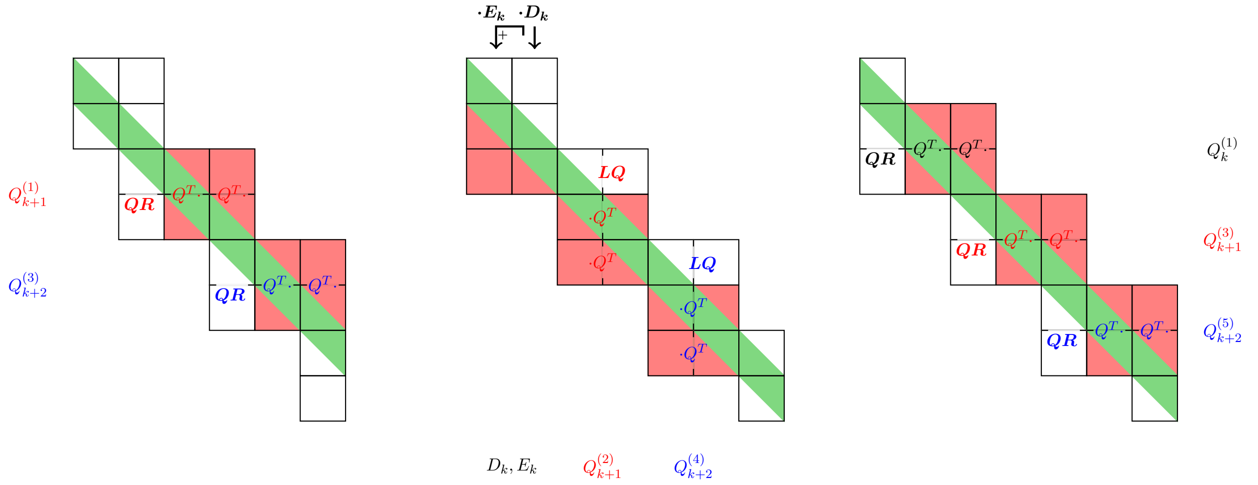


Figure 4.21: (left) Two QR decompositions are computed in parallel and applied from left. (middle) Inversion step S_k is applied and parallel to it two LQ decompositions are computed. The obtained Q is applied from right. (right) The fill-in generated in the middle picture is shifted by the next QR decompositions.

character of the procedure.

Figure 4.23 for the lower matrix half and Figure 4.24 for the upper matrix half show the execution dependencies between inversion steps and the single bulge chasing stages. The wind-up/wind-down character of the algorithm can be seen clearly. All operations within one row are fully independent and can be executed at the same time. The operations within one diagonal belong to the same inversion step.

The length of the pipeline is, as already mentioned, limited by how far the application of inversion steps has progressed. It can be computed in the lower matrix half with the top most block row used during the QR phase resulting in

$$l_{\text{pipe}}^{QR}(k) = \lfloor \frac{N-k+1}{2} \rfloor \tag{4.8}$$

or the left most block column during the LQ phase resulting in

$$l_{\text{pipe}}^{LQ}(k) = \lfloor \frac{N-k}{2} \rfloor. \tag{4.9}$$

The difference to the eigenvalue algorithm, where both pipeline length values have been the same is originated in the reference k: In the eigenvalue algorithm, the left-sided application of $Q_k^{(1)}$ following the application of inversion step k was the reference point. The right-sided application of $Q_k^{(1)}$ still refers to the index k, even if it is applied in parallel to the application of inversion step $k-1$. In the SVD algorithm, however, the left and right-sided applications of orthogonal transformations are split in left-sided QR and right-sided LQ. The former has the same reference frame while the latter refers now to the parallel inversion step $k-1$.
In the upper matrix half is the length of the QL pipeline determined by the bottom most

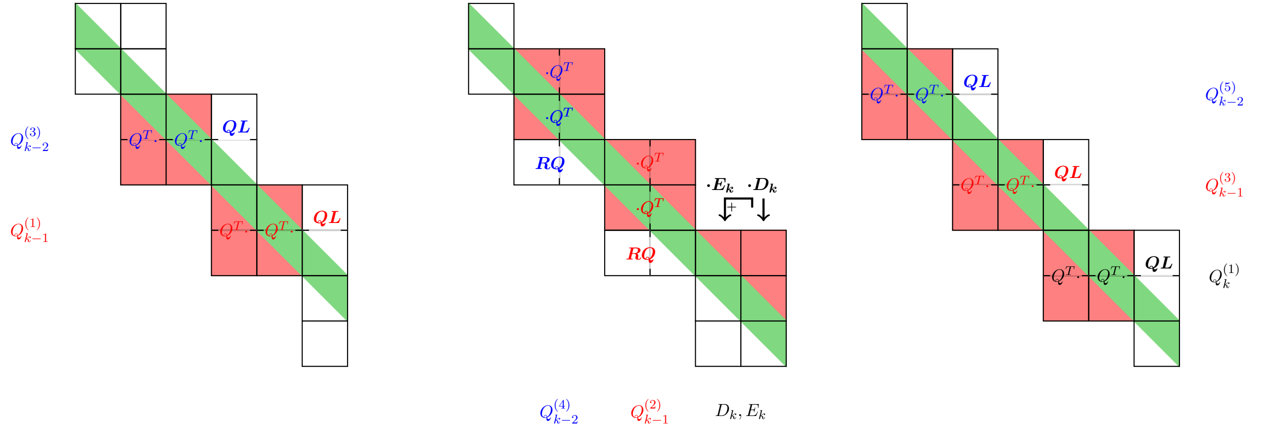


Figure 4.22: (left) Two QL decompositions are computed in parallel and applied from left. (middle) Inversion step S_k is applied and parallel to it two RQ decompositions are computed. The obtained Q is applied from right. (right) The fill-in generated in the middle picture is shifted by the next QL decompositions.

block row in the QL phase resulting in

$$l_{\text{pipe}}^{QL}(k) = \lfloor \frac{k+1}{2} \rfloor \tag{4.10}$$

and the length of the RQ pipeline is computed by the right most block column in the RQ phase resulting in

$$l_{\text{pipe}}^{RQ}(k) = \lfloor \frac{k-1}{2} \rfloor. \tag{4.11}$$

During the LQ and RQ phase, additionally the inversion step is applied. The length of the pipeline raises every inversion step until all inversion steps are applied. After this wind-up phase, the wind-down phase with a decrease of the pipeline length every step follows. During the wind-down phase, the length of the pipeline can be computed by

$$l_{\text{pipe}}^{l}(i_{\text{min}}) = \lfloor \frac{N - P - \lfloor \frac{i_{\text{min}}+1}{2} \rfloor + 1}{2} \rfloor \tag{4.12}$$

in the lower matrix half with i_{min} being the smallest stage index in the inspected row in Figure 4.21. Analogous for the upper matrix half:

$$l_{\text{pipe}}^{u}(i_{\text{min}}) = \lfloor \frac{P - \lfloor \frac{i_{\text{min}}}{2} \rfloor + 1}{2} \rfloor \tag{4.13}$$

4.4.2 Parallel execution of the upper and a lower matrix half

For inspecting the interference of the upper and lower matrix half the two cases G being in twisted or non-twisted shape have to be distinguished. Again, only the last inversion steps of each matrix half, P and $P+1$, have to be inspected.

In both cases, the lower matrix half inversion step S_{P+1} updates the blocks $G_{P+1:P+2,P:P+1}$.

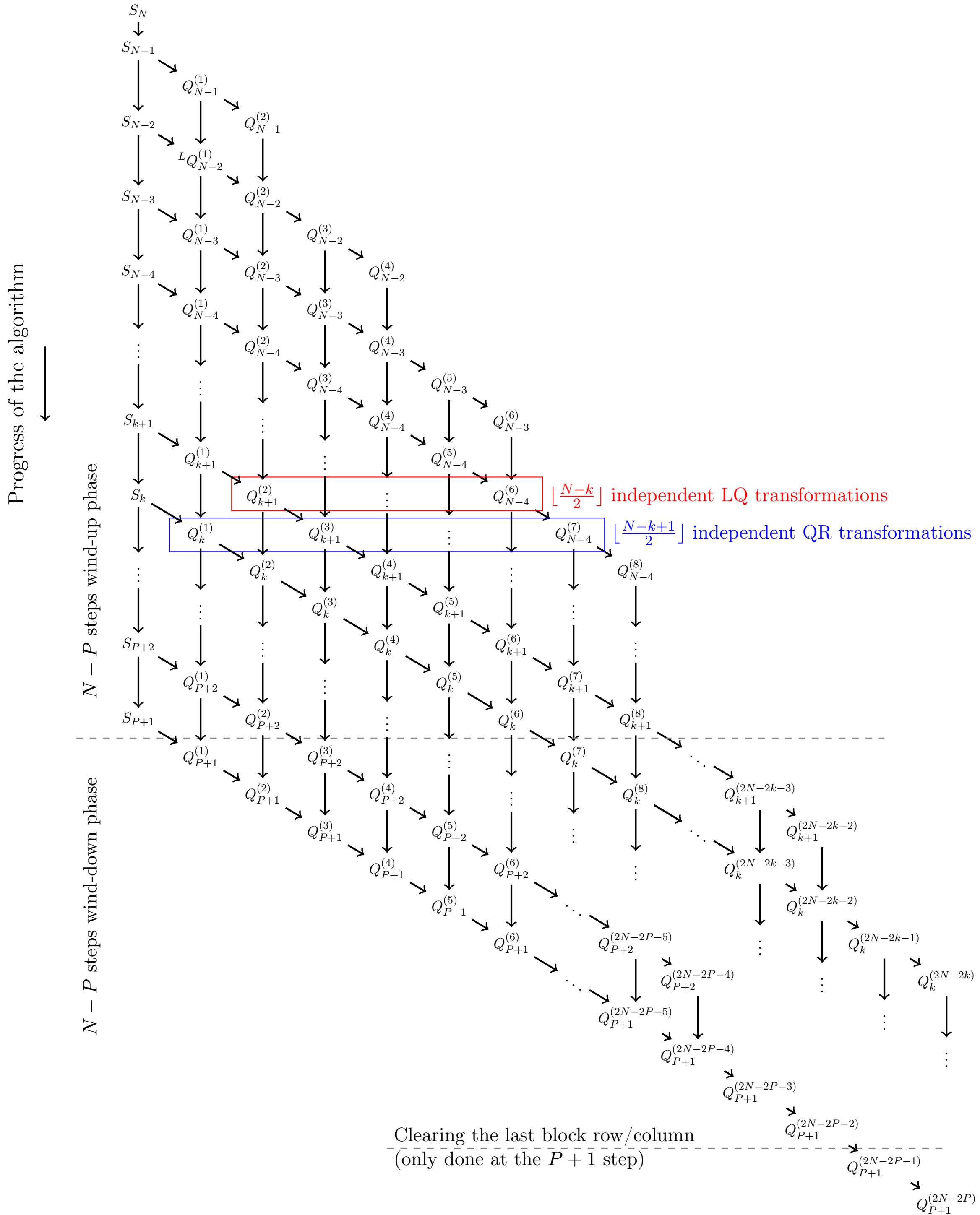


Figure 4.23: Execution dependencies of the different bulge chasing steps and stages: Every diagonal shows one inversion step and its following stages of bulge chasing. $Q_k^{(i)}$ with odd i are generated by QR decompositions and are applied from left, $Q_k^{(i)}$ with even i are generated by LQ decompositions and are applied from right. All entries in one row are independent and can be executed at the same time.

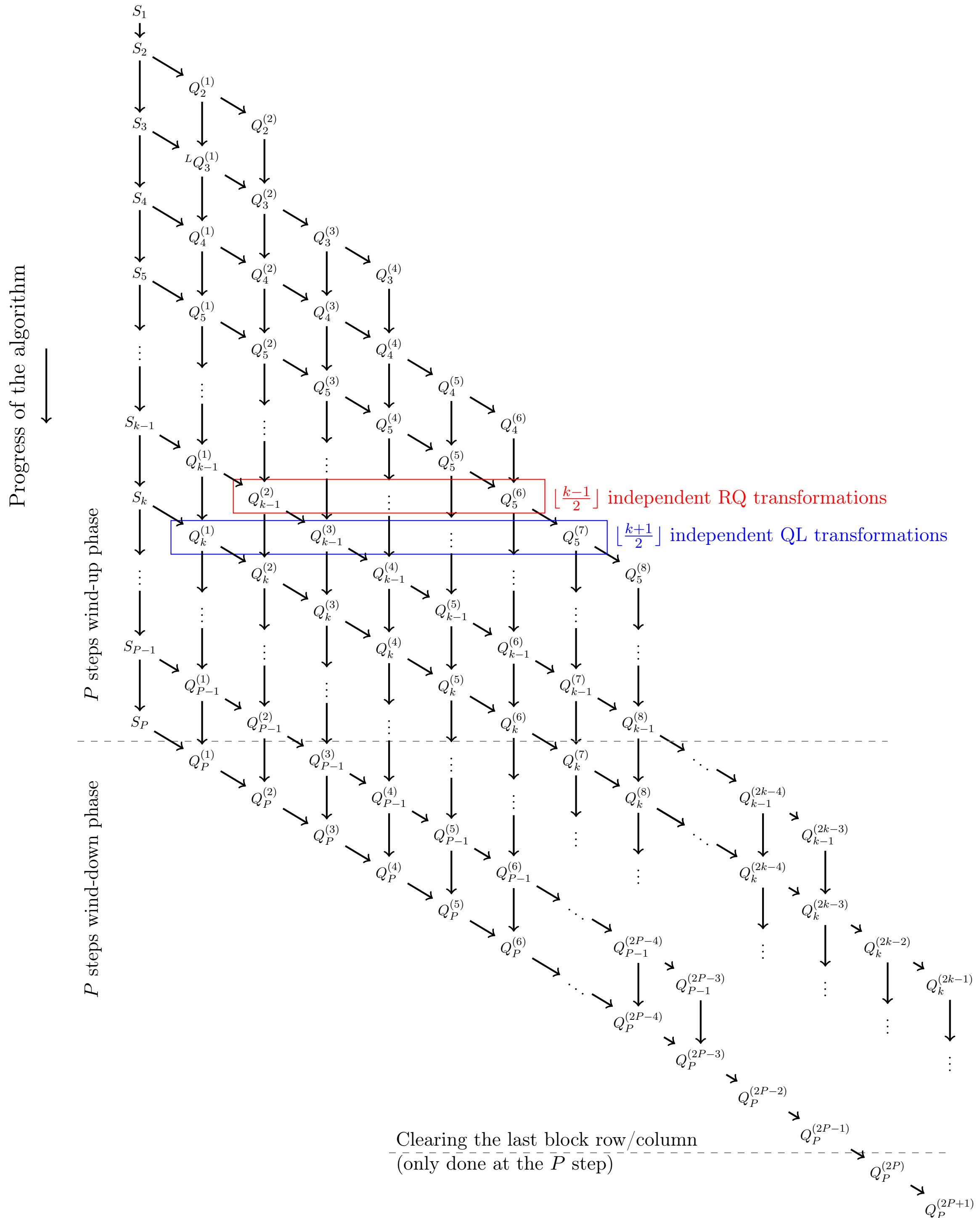


Figure 4.24: Execution dependencies of the different bulge chasing steps and stages: Every diagonal shows one inversion step and its following stages of bulge chasing. $Q_k^{(i)}$ with odd i are generated by QL decompositions and are applied from left, $Q_k^{(i)}$ with even i are generated by RQ decompositions and are applied from right. All entries in one row are independent and can be executed at the same time.

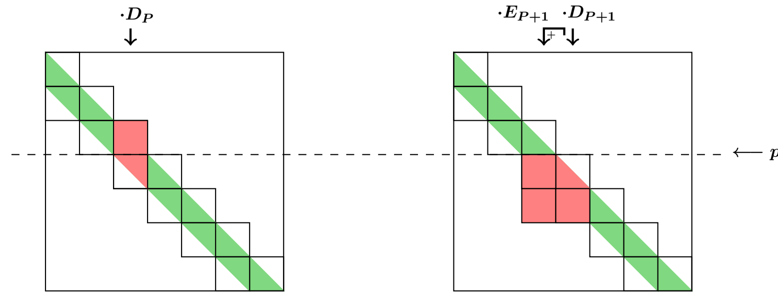


Figure 4.25: Last inversion steps when having S as twisted matrix with twist block P and G as banded lower triangular matrix. On the left, the blocks updated by the last inversion step of the upper matrix half, are highlighted in red color. On the right, the blocks updated by the last inversion step of the lower matrix half, are highlighted.

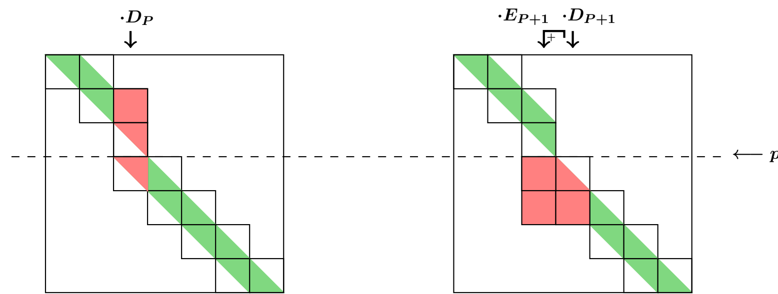


Figure 4.26: Last inversion steps when having G and S as twisted matrix with twist block P. On the left, the blocks updated by the last inversion step of the upper matrix half, are highlighted in red color. On the right, the blocks updated by the last inversion step of the lower matrix half, are highlighted.

The updated blocks are fully in the lower matrix half and hence no interference with the upper matrix half is given.
When applying S_P the situation is different for the twisted and the non-twisted G. For G being non-twisted, inversion step P updates the $G_{P,P:P+1}$. The upper block is in the upper matrix half, the lower in the lower matrix half (see Figure 4.25). This update has to be applied at some point in time after the lower matrix half has performed the QR decomposition $Q^{(1)}_{P+1}$ but does not imply a synchronization point. For G being a twisted matrix, three blocks are updated by S_P: $G_{P-1:P,P}$ in the upper matrix half are updated as well as $G_{P+1,P}$ in the lower matrix half. Like in the case of non-twisted G this update in the lower matrix half can be performed at any time and does not imply synchronization. Figure 4.26 illustrates the updated blocks for the case of two twisted matrices.

The bulge chasing is restricted to the respective matrix half and hence does not imply any interference between the upper and the lower matrix half. Hence, both matrix halves can be executed fully independent and parallel. This allows to split the processes again in groups for the upper and groups for the lower matrix half.

Algorithm 47 gives the parallel algorithm for running the algorithm in the lower matrix half. It can also be used on the flipped upper matrix half when having G as twisted matrix. For non-twisted G, Algorithm 48 provides the parallel procedure. When using the lower matrix half algorithm for the flipped upper matrix half, clearly, the application of D_P as noted in the end of Algorithm 47 does not take place. This step is only performed in the lower matrix half. In both cases the parallelism is hidden in having local data on block columns or block rows. The single loop runs containing the QR, QL, LQ or RQ can be executed in parallel. Further details on these algorithms are provided in the following chapters.

4.4.3 Block and inter-block parallelization

Inter-block parallelization

The orthogonal transformations in the Crawford SVD algorithm are an extension of the algorithms used for the twisted Crawford eigenvalue algorithm. There, only QR decomposition has to be used since by flipping the upper matrix half, QL decomposition need not be implemented. The algorithms in the eigenvalue algorithm are applied symmetrically and not as in the SVD algorithm from left or right. Hence, the number of necessary transformations grows from one to four: QR and LQ in the lower matrix half and QL and RQ in the upper matrix half.

QR and QL work on pairs of block rows, where the updated block grid has a size of 2×3. Similar for the right-sided orthogonal transformations LQ and RQ which work on block grids of size 3×2.

A QR decomposition is computed by the left two blocks and applied to all blocks in the 2×3 block grid from left. A QL decomposition is computed on the right two blocks in the grid and again applied to all blocks from left. For the generation of the orthogonal matrix as well as for the application of $Q_k^{(i)}$, two blocks in the same block column have to interact.

For a LQ decomposition the computation is performed on the top two blocks of the 3×2 block grid and applied from right to all blocks in the grid. The RQ computes $Q_k^{(i)}$ on the lower two blocks and applies $Q_k^{(i)}$ to all blocks from right. In this right-sided updates two blocks in the same block row have to interact.

The interacting pairs of blocks can generally perform their update independent of the other block pairs in the grid of six blocks. However, the generation of $Q_k^{(i)}$ implies some synchronization between the blocks of the block grid.

Block parallelization

On the block level the interaction during left and right-sided orthogonal transformations has been described. The work, however, can be further distributed within a block if n_b is large enough. For this, the data can be distributed in a 2D blockcyclic way as described in Section 4.1.1. The resulting parallel algorithms on the block and inter-block level are presented in Section 4.4.5.

Algorithm 47 Crawford SVD algorithm lower matrix half.

$j \leftarrow N; \quad k \leftarrow N$
windupPhase $\leftarrow$ true
while $j \leq N$ **do**
  if (windupPhase **and** (process has local data in BC $k-1:k$)) **then**
    apply D_k, E_k to BCs $k-1:k$
  end if
  $i \leftarrow 1+(j-k); \quad i_0 \leftarrow 1+(j-k)$
  for $r_l \leftarrow j,\ j+2,\ \ldots,\ N-1$ **do**
    if (process has local data in BRs $r_u : r_u+1$) **then**
      generate $Q^{(i)}_{k+\frac{i-i_0}{2}}$ by QR and apply $(Q^{(i)}_{k+\frac{i-i_0}{2}})^T$ from left to BRs $r_u : r_u+1$
    end if
    $i \leftarrow i+2$
  end for
  $i \leftarrow 2+(j-k); \quad i_0 \leftarrow 2+(j-k)$
  for $c_r \leftarrow j,\ j+2,\ \ldots,\ N-1$ **do**
    if (process has local data in BCs $c_l : c_l+1$) **then**
      generate $Q^{(i)}_{k+\frac{i-i_0}{2}}$ by LQ and apply $(Q^{(i)}_{k+\frac{i-i_0}{2}})^T$ from right to BCs $c_l : c_l+1$
    end if
    $i \leftarrow i+2$
  end for
  if $(j = P+1)$ **then**
    windupPhase $\leftarrow$ false
  end if
  if (windupPhase) **then**
    $k \leftarrow k-1; \quad j \leftarrow j-1$
  else
    $j \leftarrow j+1$
  end if
end while
if (process has local data in BR N) **then**
  generate $Q^{(2N-2P-1)}_{P+1}$ by QR and apply $(Q^{(2N-2P-1)}_{P+1})^T$ from left to BR N
end if
if (process has local data in BC N) **then**
  generate $Q^{(2N-2P)}_{P+1}$ by LQ and apply $(Q^{(2N-2P)}_{P+1})^T$ from right to BCs N
end if
if (process has local data in BC P) **then**
  apply D_k to BCs P
end if

Algorithm 48 Crawford SVD algorithm upper matrix half and non-twisted G.

$j \leftarrow 1; \quad k \leftarrow 1$
windupPhase $\leftarrow$ true
while $j \geq 1$ **do**
  if (windupPhase **and** (process has local data in BC $k : k+1$)) **then**
    apply D_k, E_k to BCs $k : k+1$
  end if
  $i \leftarrow 1 + (k-j); \quad i_0 \leftarrow 1 + (k-j)$
  for $r_l \leftarrow j+1,\ j-1,\ \ldots,\ 2$ **do**
    if (process has local data in BRs $r_l - 1 : r_l$) **then**
      generate $Q^{(i)}_{k-\frac{i-i_0}{2}}$ by QL and apply $(Q^{(i)}_{k-\frac{i-i_0}{2}})^T$ from left to BRs $r_l - 1 : r_l$
    end if
    $i \leftarrow i + 2$
  end for
  $i \leftarrow 2 + (k-j); \quad i_0 \leftarrow 2 + (k-j)$
  for $c_r \leftarrow j,\ j-2,\ \ldots,\ 2$ **do**
    if (process has local data in BCs $c_r - 1 : c_r$) **then**
      generate $Q^{(i)}_{k-\frac{i-i_0}{2}}$ by RQ and apply $(Q^{(i)}_{k-\frac{i-i_0}{2}})^T$ from right to BCs $c_r - 1 : c_r$
    end if
    $i \leftarrow i + 2$
  end for
  if $(j = P)$ **then**
    windupPhase $\leftarrow$ false
  end if
  if (windupPhase) **then**
    $k \leftarrow k+1; \quad j \leftarrow j+1$
  else
    $j \leftarrow j - 1$
  end if
end while
if (process has local data in BC 1) **then**
  generate $Q^{(2N-2P)}_{P+1}$ by RQ and apply $(Q^{(2N-2P)}_{P+1})^T$ from right to BCs 1
end if
if (process has local data in BR 1) **then**
  generate $Q^{(2N-2P+1)}_{P+1}$ by QL and apply $(Q^{(2N-2P+1)}_{P+1})^T$ from left to BR 1
end if

Inner-block parallelization

As for the Crawford eigenvalue algorithm also for the SVD algorithm a shared memory approach can be used on the block level. This can easily be achieved by using a multi-threaded LAPACK version instead of the standard version. Most computations are carried out by calls to LAPACK and hence this provides an easy way to introduce another parallelization layer.

4.4.4 Process and data structures

Data structures

Figures 4.21 and 4.22 have shown that three blocks in a row or three blocks in a column are sufficient to store the data of the matrix. During the Crawford eigenvalue algorithm one block was used as surrogate for an upper triangular block. In the SVD algorithm, the upper matrix half blocks are fully stored since the underlying matrices are not symmetric anymore. Therefore no surrogate block is necessary anymore.

When closely inspecting Figure 4.21 or Figure 4.22 it can be seen that some blocks appear and some disappear between the single illustrations. That left and right-sided orthogonal transformations work on different block setups has already been seen in the eigenvalue algorithm. In the SVD algorithm, however, even between two left or two right-sided transformations, the block setup is not the same. After four orthogonal transformations (QR-LQ-QR-LQ or QL-RQ-QL-RQ) the block positions are the same again. These different block layouts are referred to as "states" one to four in the following.

Within a state, a block that will disappear after the state is zeroed by an orthogonal transformation. The newly appearing block is empty at the beginning of the state and filled by the application of an orthogonal transformation. The appearing and disappearing at the same time can be exploited by using the processes and data structures of a disappearing block for the appearing block. This changes the blocks collaborating with each other and thus, for every state a collaboration pattern has to be introduced. Figure 4.27 depicts the block shifting and illustrates an exemplary orthogonal transformation in every state.
It can be seen that state 1 starts in the upper and lower matrix half with a 3×2 block setup. In the lower matrix half, the end of the matrix has only shortened 2×2 version of it whereas in the upper matrix half, the first two block columns of the matrix have a full 3×2 block setup. The moving block is different in both matrix halves. In the upper matrix the block in position $3, 1$ of the 3×2 block setup moves to the virtual position $4, 2$. In the lower matrix half block $1, 2$ moves to the virtual position $2, 3$. In state 2 the blocks are again moving down along the diagonal. Block $1, 3$ of the 2×3 moves to the virtual position $2, 4$ in the upper matrix half and block $2, 1$ moves to position $3, 2$. In state 3 and 4 the blocks are moving up along the diagonal. In state 3 the blocks position of state 1 moves, in state 4 the blocks in the same block setup position as in state 2. After this four states the initial block setup is restored and the stages of an exemplary orthogonal transformation have moved by two block rows and columns.

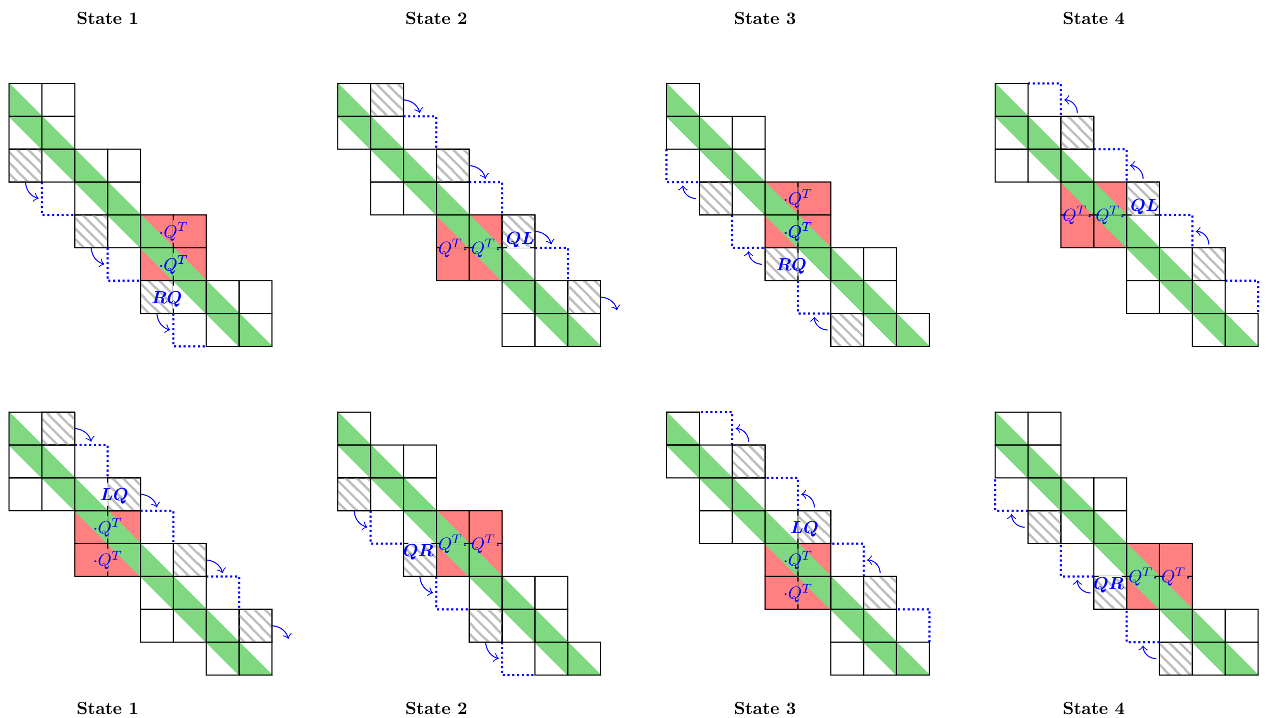


Figure 4.27: Changes in the block structure at states transitions (Upper row: upper matrix half, lower row: lower matrix half). The hatching marks the blocks that are zero when leaving the state, the arrow and the blue dotted rectangle show where this blocks will be used in the next state. An exemplary orthogonal transformation in the respective state is shown and the fill-in outside the band is indicated in red.

Process layout

The algorithm in the upper as well as in the lower matrix half starts with state 1 in the block setup. This allows to use the process setup as described in Section 4.2.4 and depicted in Figure 4.8.
Processes are split in a group for the upper and a group for the lower matrix half. They can operate independently on their regarding matrix half. The processes of a matrix half are equally subdivided into process groups (PG). Three PG form a process group columns (PGC) as well as three PG form a process group row (PGR). The grouping in PGC and PGR is more an organizational construct to organize the collaboration between the process groups. Which of them is used depends on the state (PGC for state 1 and 3, PGR for state 2 and 4). Block columns of a matrix half are assigned cyclically to process group columns in state 1. The data of a single block in a block column is assigned to a process group in the owning PGC. The ordering of the blocks in a block column is also reflected by the order of the process groups in a PGC. The processes of a process group are arranged in a 2D grid and the data is distributed block-cyclically to them as described in Section 4.1.1.

The later states 2 - 4 are derived from state 1 by shifting the moving block and the underlying process group and redefining the PGCs or PGRs. This leads finally to having four different process setups and also having four different local communicator setups for the 3×2 and 2×3 process grid.

This process setup allows to have the same parallelization layers as in the eigenvalue algorithm: splitting in upper and lower matrix half, splitting in pairs of PGC or pairs of PGR, splitting the data within a block and using threaded BLAS routines.

4.4.5 Parallel algorithms on the level of pairs of block rows and block columns

In the following the parallel algorithms on the level pairs of block columns and pairs of block rows are described. The coarse grained algorithms for the upper and lower matrix half have already been described above. The notation used in this section was described in Section 4.2.5.

Application of an inversion step

Applying an inversion step differs slightly in the upper and lower matrix half (when not having G as twisted matrix). Basically, the upper matrix half algorithm is a flipped version of the lower matrix half algorithm. Algorithms 49 and 50 provide the parallel implementations of the application of an inversion step. Due to having only a right-sided application, this procedures are simplified versions of Algorithm 31. Block column k is multiplied by E_k and added to block column $k-1$ ($k+1$ in the upper matrix half). Afterwards, block column k is multiplied by D_k.

QR and QL decomposition and left-sided application

The QR decomposition and the regarding left-sided application is described in detail in Section 4.2.5. In the SVD algorithm the situation is exactly the same.

Algorithm 49 Parallel application of an inversion step (SVD - lower matrix half).

```
function G_{<p_r,p_c>[i,j]} ← ParallelDEapplication(G_{<p_r,p_c>[i,j]}, D_{<p_r,p_c>[i,j]}, E_{<p_r,p_c>[i,j]})
    if (process has local data in left BC ) then
        if (process has local data in block 2 or 3 of the BC ) then
            b_{<p_r,p_c>[i,j]} ← MPI_RECEIVE from block 2 or 3 in right BC
            G_{[i,j]} ← G_{[i,j]} + PDGEMM (b_{[i,j]}, E_{[i,j]})
        end if
    else
        if (process has local data in block 2 or 3 of the BC ) then
            b_{<p_r,p_c>[i,j]} ← G_{[i,j]}
            MPI_SEND (b_{<p_r,p_c>[i,j]}) to block 2 or 3 in left BC
            G_{[i,j]} ← PDTRMM (G_{[i,j]}, D_{[i,j]})
        end if
    end if
end function
```

Algorithm 50 Parallel application of an inversion step (SVD - upper matrix half).

```
function G_{<p_r,p_c>[i,j]} ← ParallelDEapplication(G_{<p_r,p_c>[i,j]}, D_{<p_r,p_c>[i,j]}, E_{<p_r,p_c>[i,j]})
    if (process has local data in left BC ) then
        if (process has local data in block 1 or 2 of the BC ) then
            b_{<p_r,p_c>[i,j]} ← G_{[i,j]}
            MPI_SEND (b_{<p_r,p_c>[i,j]}) to block 1 or 2 in right BC
            G_{[i,j]} ← PDTRMM (G_{[i,j]}, D_{[i,j]})
        end if
    else
        if (process has local data in block 1 or 2 of the BC ) then
            b_{<p_r,p_c>[i,j]} ← MPI_RECEIVE from block 1 or 2 in left BC
            G_{[i,j]} ← G_{[i,j]} + PDGEMM (b_{[i,j]}, E_{[i,j]})
        end if
    end if
end function
```

For the QL decomposition the algorithm works in the same way but has to consider some differences: Instead of block column one, the decomposition is carried out in block column three and not the entries below a certain index are cleared but above.
Hence, a modified version of ParallelHHgen as provided in Algorithm 51 becomes necessary. All entries of the initial vector x until index $m-1$ are set to zero by the procedure. The resulting Householder vector has, contrary to the QR decomposition, the one at the last entry and not at the first entry.

Algorithm 51 Parallel Householder vector generation, the last entry of the given vector remains.

function $[x_{<p_r>[i]},\ v_{<p_r>[i]},\ \tau_{<*>}] \leftarrow \text{ParallelHHgen}(x_{<p_r>[i]})$
  $d_{<*>i} \leftarrow x_{[i]}^T \cdot x_{[i]}$
  $a_{<*>i} \leftarrow 0$
  if ($x(m)$ is local) **then**
    $a_{<*>i} \leftarrow x(m)$
  end if
  $[a_{<*>},\ d_{<*>}] \leftarrow \text{MPI_ALLREDUCE}\ (a_{<*>i},\ d_{<*>i})$
  $\beta \leftarrow \text{sign}(x(m)) \cdot \sqrt{d}$
  $\tau \leftarrow \frac{a+\beta}{\beta}$
  $v_{[i]} \leftarrow \frac{1}{a+\beta} x_{[i]}$
  $x_{[i]} \leftarrow 0$
  if ($x(m)$ is local) **then**
    $v(m) \leftarrow 1$
    $x(m) \leftarrow -\beta$
  end if
end function

The left-sided unblocked application of a Householder vector can be performed by ParallelApplyHHleft given in Algorithm 33. Using ParallelApplyHHleft and Algorithm 51 allows to formulate ParallelQL which computes a parallel unblocked QL decomposition. It steps from right to left through the given matrix and eliminates column wise entries until the lower triangular part remains.

For the blocked application of Householder vectors, in the same manner as in the QR decomposition, a compact WY formulation can be used. Algorithm 35 is employed to compute the required T. T and the matrix of Householder vectors Y are used to apply the Householder vectors in a blocked way as described in Algorithm 36.

With Algorithms 35, 36 and 52, a blocked QL algorithm can be derived. This parallel blocked QL is given in Algorithm 53. Starting from right, a QL decomposition is computed for a tile and applied to the rest of the matrix by the compact WY formulation. After completion, the next tile is computed and applied from left until block column 3 is fully decomposed. For the communication between the block columns the same pattern is used as in the blocked QR algorithm. Just block column three and block column one change their roles and hence block column three sends the data.

Algorithm 52 Parallel QL decomposition.

function $[A_{<p_r,p_c>[i,j]},\ Y_{<p_r,*_1>[i]},\ \tau_{<*,*_1>}] \leftarrow \text{ParallelQL}(A_{<p_r,p_c>[i,j]})$
  $Y \leftarrow 0$
  for $k \leftarrow n, n-1, \ldots, 1$ **do**
    $l \leftarrow n - k + 1$
    if (k is local column) **then**
      $[A_{<p_r>[i]}(1:k,k),\ Y_{<p_r>[i]}(1:k,l),\ \tau_{<*>}(l)] \leftarrow \text{ParallelHHgen}(A_{<p_r>[i]}(1:k,k))$
    end if
    $[Y_{<p_r,*_1>[i]}(1:k,l),\ \tau_{<*,*_1>}(l)] \leftarrow$ MPI_BROADCAST $(Y_{<p_r>[i]}(1:k,l),\ \tau_{<*>}(l))$

    $A(1:k,1:k-1) \leftarrow \text{ParallelApplyHHleft}(A(1:k,1:k-1),\ Y_{[i]}(1:k,l),\ \tau(l))$
  end for
end function

Algorithm 53 Parallel blocked QL decomposition.

function $[A_{<p_r,p_c>[i,j]},\ Y_{<p_r,*>[i]},\ \tau_{<*,*>}] \leftarrow \text{ParallelBlockedQL}(A_{<p_r,p_c>[i,j]})$
  for $j_2 \leftarrow n_b, n_b - b, \ldots, 1$ **do**
    $i_1 \leftarrow 1;\quad i_2 \leftarrow n_b + j_2$
    $j_1 \leftarrow \max(1,\ j_2 - b + 1)$
    $l_1 \leftarrow n_b - j_2 + 1;\quad l_2 \leftarrow n_b - j_1 + 1$
    if (is block column 3) **then**
      $[A_{[i,j]}(i_1:i_2, j_1:j_2),\ Y_{<p_r,*_1>[i]}(i_1:i_2, l_1:l_2),\ \tau_{<*,*_1>}(l_1:l_2)]$
        $\leftarrow \text{ParallelQL}(A_{[i,j]}(i_1:i_2, j_1:j_2))$

      MPI_SEND $(Y_{<p_r,*_1>[i]}(i_1:i_2, l_1:l_2),\ \tau_{<*,*_1>}(l_1:l_2))$ to BCs 1, 2
    else
      $[Y_{<p_r,*>[i]}(i_1:i_2, l_1:l_2),\ \tau_{<*,*>}(l_1:l_2)] \leftarrow$ MPI_RECEIVE from BC 3
    end if

    $T_{<*,*>} \leftarrow \text{ParallelGenT}(Y_{[i]}(i_1:i_2, l_1:l_2),\ \tau(l_1:l_2))$
    $A_{[i,j]}(i_1:i_2, 1:j_1-1) \leftarrow \text{ParallelCWYleft}(A_{[i,j]}(i_1:i_2, 1:j_1-1),$
      $Y_{[i]}(i_1:i_2, l_1:l_2),\ T)$
  end for
end function

LQ and RQ decomposition and right-sided application

Every left-sided orthogonal transformation is followed by a right-sided transformation. For the lower matrix half, this will be a LQ decomposition, for the upper matrix half a RQ decomposition. The left-sided transformations work on columns of the underlying matrix. The right-sided transformations, however, process the underlying matrix row-wise. Regarding the computational efficiency, this is a clear drawback when using a column-mayor data layout.

During the left-sided transformations, Y has been filled in the following way:

$$Y = [v_1, v_2, \dots v_n] \tag{4.14}$$

The v_k denote the Householder vectors in the order of their creation. For left-sided orthogonal transformations the Householder vectors are column vectors.
For the right-sided orthogonal transformations the order or the vectors shall stay the same:

$$Y' = [v_1^T, v_2^T, \dots v_n^T] \tag{4.15}$$

Since the Householder vectors are row vectors, the transpose of them is used in this notation to indicate that every Householder vector is stored as column vector. The single Householder vectors are distributed over the process columns of a process group. Hence, also the collection of the Householder vectors is distributed in the same way and thus, Y' is used to indicate the different distribution over the processes.

The generation of Householder vectors does not differ from Algorithms 32 and 51 besides the different distribution of the vector to transform, x, and in the same way the different distribution of the resulting Householder vector v. For completeness and to avoid confusion, the versions for the right-sided orthogonal transformations are provided in Algorithms 54 and 55.

Algorithm 54 Parallel Householder vector generation for row vector, the first entry of the given vector remains.

function $[x_{<p_c>[j]},\ v_{<p_c>[j]},\ \tau_{<*>}] \leftarrow \text{ParallelHHgenRow}(x_{<p_c>[j]})$
  $d_{<*>j} \leftarrow x_{[j]}^T \cdot x_{[j]}$
  $a_{<*>j} \leftarrow 0$
  if ($x(1)$ is local) **then**
    $a_{<*>j} \leftarrow x(1)$
  end if
  $[a_{<*>},\ d_{<*>}] \leftarrow \text{MPI_ALLREDUCE}\ (a_{<*>j},\ d_{<*>j})$
  $\beta \leftarrow \text{sign}(x(1)) \cdot \sqrt{d}$
  $\tau \leftarrow \frac{a+\beta}{\beta}$
  $v_{[j]} \leftarrow \frac{1}{a+\beta} x_{[j]}$
  $x_{[j]} \leftarrow 0$
  if ($x(1)$ is local) **then**
    $v(1) \leftarrow 1$
    $x(1) \leftarrow -\beta$
  end if
end function

During the right-sided orthogonal transformations the Householder vectors have to be applied from right. Algorithm 56 can be used to apply a single Householder vector that is distributed over the process columns, to a given matrix.

By using the creation of Householder vectors in Algorithm 54 and the right-sided application

Algorithm 55 Parallel Householder vector generation for row vector, the last entry of the given vector remains.

function $[x_{<p_r>[j]},\ v_{<p_c>[j]},\ \tau_{<*>}] \leftarrow \text{ParallelHHgen}(x_{<p_c>[j]})$
  $d_{<*>j} \leftarrow x_{[j]}^T \cdot x_{[j]}$
  $a_{<*>j} \leftarrow 0$
  if $(x(m)$ is local$)$ **then**
    $a_{<*>j} \leftarrow x(m)$
  end if
  $[a_{<*>},\ d_{<*>}] \leftarrow \text{MPI_ALLREDUCE}\ (a_{<*>j},\ d_{<*>j})$
  $\beta \leftarrow \text{sign}(x(m)) \cdot \sqrt{d}$
  $\tau \leftarrow \frac{a+\beta}{\beta}$
  $v_{[j]} \leftarrow \frac{1}{a+\beta} x_{[j]}$
  $x_{[j]} \leftarrow 0$
  if $(x(m)$ is local$)$ **then**
    $v(m) \leftarrow 1$
    $x(m) \leftarrow -\beta$
  end if
end function

Algorithm 56 Parallel right-sided application of a Householder vector.

function $A_{<p_r,p_c>[i,j]} \leftarrow \text{ParallelApplyHHright}(A_{<p_r,p_c>[i,j]},\ v_{<*,p_c>[j]},\ \tau_{<*,*>})$
  $z_{<p_r>[i]j} \leftarrow A_{[i,j]} \cdot v_{[j]}^T$
  $z_{<p_r>[i]} \leftarrow \text{MPI_ALLREDUCE}\ (z_{<p_r>[i]j})$
  $A_{[i,j]} \leftarrow A_{[i,j]} - \tau \cdot z_{[i]} \cdot v_{[j]}$
end function

in Algorithm 56, the LQ decomposition of a matrix can be formulated as in Algorithm 57.

Algorithm 57 Parallel LQ decomposition.

function $[A_{<p_r,p_c>[i,j]},\ Y_{<*_1,p_c>[j]},\ \tau_{<*_1,*>}] \leftarrow \text{ParallelLQ}(A_{<p_r,p_c>[i,j]})$
  $Y \leftarrow 0$
  for $k \leftarrow 1, 2, \ldots, m$ **do**
    if (k is local row) **then**
      $[A_{<p_c>[j]}(k, k:n),\ Y_{<p_c>[j]}(k:n,k),\ \tau_{<*>}(k)]$
        $\leftarrow \text{ParallelHHgen}(A_{<p_c>[j]}(k, k:n))$
    end if
    $[Y_{<*_1,p_c>[j]}(k:n,k),\ \tau_{<*_1,*>}(k)]$
      $\leftarrow$ MPI_BROADCAST $(Y_{<p_c>[j]}(k:n,k),\ \tau_{<*>}(k))$

    $A(k+1:m, k:n) \leftarrow \text{ParallelApplyHHright}(A(k+1:m, k:n),\ Y_{[j]}(k:n,k),\ \tau(k))$
  end for
end function

The same can be composed for the RQ decomposition. ParallelRQ as given in Algorithm 58 using Algorithms 55 and 56 computes the RQ decomposition of a given matrix in an unblocked manner.

Algorithm 58 Parallel RQ decomposition.

function $[A_{<p_r,p_c>[i,j]},\ Y_{<*_3,p_c>[i]},\ \tau_{<*_3,*>}] \leftarrow \text{ParallelRQ}(A_{<p_r,p_c>[i,j]})$
  $Y \leftarrow 0$
  for $k \leftarrow m, m-1, \ldots, 1$ **do**
    $l \leftarrow m - k + 1$
    if (k is local row) **then**
      $[A_{<p_c>[j]}(k, 1:n_b+k),\ Y_{<p_c>[j]}(1:n_b+k, l),\ \tau_{<*>}(l)]$
        $\leftarrow \text{ParallelHHgen}(A_{<p_c>[j]}(k, 1:n_b+k))$
    end if
    $[Y_{<*_1,p_c>[j]}(1:n_b+k, l),\ \tau_{<*_1,*>}(l)]$
      $\leftarrow$ MPI_BROADCAST $(Y_{<p_c>[j]}(1:n_b+k, l),\ \tau_{<*>}(l))$

    $A(1:k-1, 1:n_b+k) \leftarrow \text{ParallelApplyHHright}(A(1:k-1, 1:n_b+k),$
      $Y_{[j]}(1:n_b+k, l),\ \tau(l))$
  end for
end function

For applying the Householder vectors in a blocked way, the compact WY formulation is also used for the right-sided transformations. Algorithm 39 provides the necessary algorithm which requires T and Y'. The former can be computed by Algorithm 38. The latter, Y', is obtained by Algorithms 57 and 58.

So far, all parts for the blocked computation and application of the LQ and RQ decomposition have been described. ParallelBlockedLQ given in Algorithm 59 and ParallelBlockedRQ

given in Algorithm 60 provide these implementations. For the LQ decomposition, block row one performs the decomposition and block rows two and three apply the decomposition. In the RQ case, block row three computes the RQ and block rows one and two only apply the Householder transformations.

Algorithm 59 Parallel blocked LQ decomposition.

function $[A_{<p_r,p_c>[i,j]},\ Y_{<*,p_c>[j]},\ \tau_{<*,*>}] \leftarrow \text{ParallelBlockedLQ}(A_{<p_r,p_c>[i,j]})$
  for $i_1 \leftarrow 1, b+1, 2b+1, \ldots, n_b$ **do**
    $i_2 \leftarrow \max(n_b,\ i_1 + b - 1)$
    $j_1 \leftarrow i_1; \qquad j_2 \leftarrow 2n_b$
    if (is block row 1) **then**
      $[A_{[i,j]}(i_1 : i_2, j_1 : j_2),\ Y_{<*_1,p_c>[j]}(j_1 : j_2, i_1 : i_2),\ \tau_{<*_1,*>}(i_1 : i_2)]$
        $\leftarrow \text{ParallelLQ}(A_{[i,j]}(i_1 : i_2, j_1 : j_2))$

      MPI_SEND $(Y_{<*_1,p_c>[j]}(j_1 : j_2, i_1 : i_2),\ \tau_{<*_1,*>}(i_1 : i_2))$ to BRs 2, 3
    else
      $[Y_{<*,p_c>[j]}(j_1 : j_2, i_1 : i_2),\ \tau_{<*,*>}(i_1 : i_2)] \leftarrow$ MPI_RECEIVE from BR 1
    end if

    $T_{<*,*>} \leftarrow \text{ParallelGenT}(Y_{[j]}(j_1 : j_2, i_1 : i_2),\ \tau(i_1 : i_2))$
    $A_{[i,j]}(i_2 + 1 : 3n_b, j_1 : j_2) \leftarrow \text{ParallelCWYleft}(A_{[i,j]}(i_2 + 1 : 3n_b, j_1 : j_2),$
      $Y_{[j]}(j_1 : j_2, i_1 : i_2),\ T)$
  end for
end function

Again, for all blocked orthogonal transformations, the MPI_SEND and MPI_RECEIVE can be omitted when immediately using a MPI_BROADCAST to distribute every single Householder vector after creation. With the described version, the Householder vectors are immediately distributed within the own block and after having completed the decomposition of a tile, all Householder vectors of this tile are sent together to the other blocks. This increases the message size, reduces the overall message count and has shown to be superior during testing.

4.5 Backtransformation of singular vectors

Equations (3.39) and (3.41) provide the formulas for computing the singular vectors of the original singular value problem:

$$U = {}^{\mathrm{o}}Q_N^{(\cdot)} \cdots {}^{\mathrm{o}}Q_{P+1}^{(\cdot)} \cdot {}^{\mathrm{o}}Q_1^{(\cdot)} \cdots {}^{\mathrm{o}}Q_P^{(\cdot)} \cdot \widetilde{U}$$
$$V^T = \widetilde{V}^T \cdot {}^{\mathrm{e}}Q_P^{(\cdot)} \cdots {}^{\mathrm{e}}Q_1^{(\cdot)} \cdot {}^{\mathrm{e}}Q_{P+1}^{(\cdot)} \cdots {}^{\mathrm{e}}Q_N^{(\cdot)}$$

To perform the backtransformations of the left and the right singular vectors, the same approaches as in the eigenvector algorithm are available: A pipelining approach and the assembly of backtransformation matrices. For both approaches, the computation of the transposed right

Algorithm 60 Parallel blocked RQ decomposition.

function $[A_{<p_r,p_c>[i,j]},\ Y_{<*,p_c>[j]},\ \tau_{<*,*>}] \leftarrow \text{ParallelBlockedRQ}(A_{<p_r,p_c>[i,j]})$
  for $i_2 \leftarrow 3n_b, 3n_b - b, \ldots, 2n_b + 1$ **do**
    $i_1 \leftarrow \max(2n_b + 1,\ i_2 - b + 1)$
    $j_1 \leftarrow 1; \qquad j_2 \leftarrow i_2 - n_b$
    $l_1 \leftarrow 3n_b - i_2 + 1; \qquad l_2 \leftarrow 3n_b - i_1 + 1$
    if (is block row 3) **then**
      $[A_{[i,j]}(i_1 : i_2, j_1 : j_2),\ Y_{<*_3,p_c>[j]}(j_1 : j_2, l_1 : l_2),\ \tau_{<*_3,*>}(l_1 : l_2)]$
        $\leftarrow \text{ParallelRQ}(A_{[i,j]}(i_1 : i_2, j_1 : j_2))$

      MPI_SEND $(Y_{<*_3,p_c>[j]}(j_1 : j_2, l_1 : l_2),\ \tau_{<*_3,*>}(l_1 : l_2))$ to BRs 1, 2
    else
      $[Y_{<*,p_c>[j]}(j_1 : j_2, l_1 : l_2),\ \tau_{<*,*>}(l_1 : l_2)] \leftarrow$ MPI_RECEIVE from BR 3
    end if

    $T_{<*,*>} \leftarrow \text{ParallelGenT}(Y_{[j]}(j_1 : j_2, l_1 : l_2),\ \tau(l_1 : l_2))$
    $A_{[i,j]}(1 : i_1 - 1, j_1 : j_2) \leftarrow \text{ParallelCWYright}(A_{[i,j]}(1 : i_1 - 1, j_1 : j_2),$
      $Y_{[j]}(j_1 : j_2, l_1 : l_2),\ T)$
  end for
end function

singular vectors V^T instead of V is more favorable since it uses the same block pattern as the generation and application to G.

4.5.1 Pipelining approach

The pipelining approach follows the application of the orthogonal transformations to G in reverse order. In the following, the descriptions are limited to the case of having G as non-twisted matrix and S as twisted matrix. The case of having additionally G in twisted shape is handled by the lower matrix half part of the algorithms together with a flipping of the upper matrix half. The case of having non-twisted matrices G and F is also covered by the lower matrix half algorithm.

Starting point are the singular vectors of $\widetilde{W}$, $\widetilde{U}$ and $\widetilde{V}^T$. The orthogonal transformations given by Equations (3.39) and (3.41) are applied to this singular vector matrices. This application updates the singular vectors in place and does not require any additional storage.

Since the orthogonal transformations work on pairs of block columns or pairs of block rows, the singular vector matrices should also reflect this. This means that the left singular vectors, to which the orthogonal transformations are applied to from left, should be subdivided in block rows of size n_b. Similar for the right singular vectors, which undergo right-sided orthogonal transformations and hence are subdivided in block columns of size n_b. Three blocks in a block column or a block row co-work in an orthogonal transformation. This can be kept for the backtransformation. Then, for the left singular vectors, a process group holds a block of size $n_b \times w_j$ with w being the number of singular vectors to transform and $w_j \approx \frac{w}{3}$. For the right

singular vectors, a process group holds a block of size $w_i \times n_b$ with $w_i \approx \frac{w}{3}$. Another possibility is to keep the uniform blocksize $n_b \times n_b$ and distribute the blocks cyclically to the process groups. The former approach, however, distributes the singular vectors in a more equal way to the processes.

The independence of the orthogonal transformations and their parallel capabilities is already shown in Section 4.4.1. Hence, the parallelism in the pairs of block rows and pairs of block columns is only exemplarily shown in Figure 4.28 for the backtransformation of the right singular vectors in the lower matrix half. They undergo the before generated LQ transformations. The upper matrix half of the right singular vectors undergoes a series of RQ transformations and in the same way the left singular vectors undergo series of QR and QL transformations.

Parallel execution of the upper and a lower matrix half

During the backtransformation of singular vectors, contrary to the eigenvector backtransformation, no inversion steps have to be applied to the matrices of singular vectors. And since the orthogonal transformation are limited to their matrix half, no interference between both matrix halves is given. Hence, both can be executed fully independent.

Algorithm 61 provides the backtransformation of the left singular vectors. The initial left singular vector matrix $\widetilde{U}$ is updated by all left-sided orthogonal transformations computed during the bulge chasing. In the same way Algorithm 62 gives the implementation for obtaining the right singular vectors V^T from the initial right singular vector matrix $\widetilde{V}^T$. In both cases, the backtransformation algorithms follow the pipeline used during the application of the inversion steps and the regarding bulge chasing in a reverse order. In both cases, the upper and the lower matrix half can be executed in parallel as well as all transformations in the for loops.

Process and data structures

Y and T (or τ) have to be stored to be available for the backtransformation. Due to the moving blocks between the states and the constant mapping of process groups to blocks of the singular vectors matrices, the situation becomes more complicated. Figure 4.29 illustrates the situation for the right singular vectors. The blocks of the singular vectors are assigned to the process groups according to state 1. In state 3, however, the block neighborhood changes by the moving blocks, but not in the singular vector matrices. Figure 4.29 illustrates the problem: PG4 holds a right block in state 1 and will hence be assigned to block column 2 in the singular vector matrix. In state 3, the underlying block has moved and PG4 still holds a right block. However, in the singular vector matrix, PG4 holds block column 2 and thus a left block. Therefore, PG4 has the wrong part of Y. Consequently, for state 1 all processes can hold the correct parts of Y and T, but for state 3 at least the moving blocks have to receive Y. T is the same in the left and right block column and hence need not be communicated.
For the left singular vectors this situation is the same and thus, also there the moving blocks have to receive the correct Y parts by communication (see Figure 4.30).

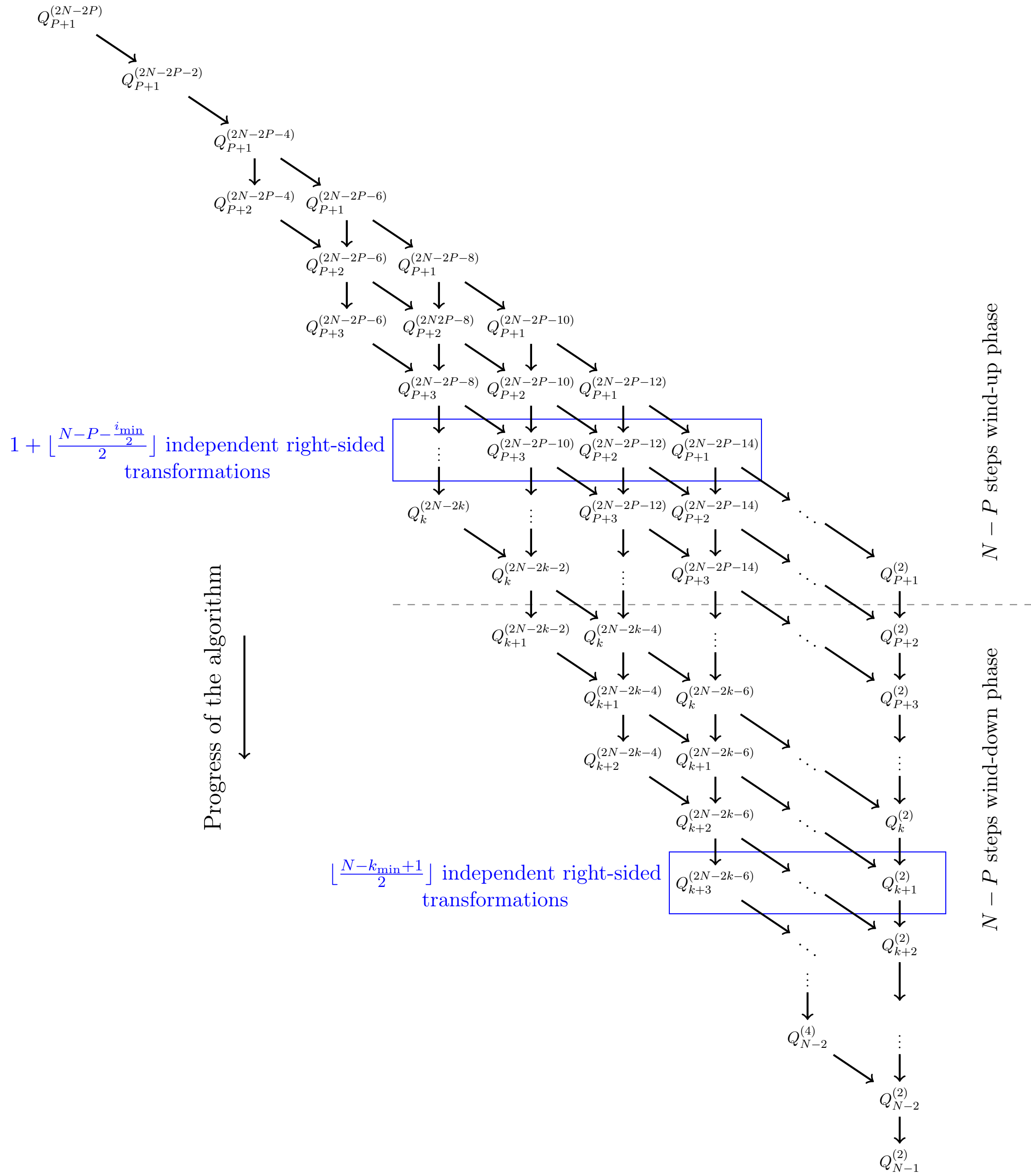


Figure 4.28: Execution dependencies in the backtransformation of the right singular vectors in the lower matrix half: Every diagonal shows the application of LQ transformations caused by one inversion step. All entries in one row are independent and can be executed at the same time. The number of independent operations can be computed by the given formulas. In the wind-up phase, $i_{\min}$ refers to the minimum i-index in the row, $k_{\min}$ refers to the minimum k-index in a row.

Algorithm 61 Backtransformation of left singular vectors.

```
windupPhase ← true
if (Upper matrix half) then
    if (process has local data in BR 1) then
        apply Q_P^(2P) from left to BR 1
    end if
    k ← P;    j ← 1
    while j ≥ 1 do
        i ← 1 + (k − j);    i_0 ← 1 + (k − j)
        for r_l ← j + 1, j − 1, ..., 2 do
            if (process has local data in BRs r_l − 1 : r_l) then
                apply Q^(i)_{k − (i−i_0)/2} from left to BRs r_l − 1 : r_l
            end if
            i ← i + 2
        end for
        if (j = P) then
            windupPhase ← false
        end if
        if (windupPhase) then
            j ← j + 1
        else
            k ← k − 1;    j ← j − 1
        end if
    end while
else
    if (process has local data in BR N) then
        apply Q_{P+1}^(2N−2P−1) from left to BR N
    end if
    k ← P + 1;    j ← N
    while j ≤ N do
        i ← 1 + (j − k);    i_0 ← 1 + (j − k)
        for r_l ← j, j + 2, ..., N − 1 do
            if (process has local data in BRs r_u : r_u + 1) then
                apply Q^(i)_{k + (i−i_0)/2} from left to BRs r_u : r_u + 1
            end if
            i ← i + 2
        end for
        if (j = P + 1) then
            windupPhase ← false
        end if
        if (windupPhase) then
            j ← j − 1
        else
            k ← k + 1;    j ← j + 1
        end if
    end while
end if
```

Algorithm 62 Backtransformation of right singular vectors.

```
windupPhase ← true
if (Upper matrix half) then
    if (process has local data in BC 1) then
        apply Q_P^(2P+1) from right to BC 1
    end if
    k ← P;    j ← 1
    while j ≥ 1 do
        i ← 2 + (k − j);    i_0 ← 2 + (k − j)
        for c_r ← j, j − 2, ..., 2 do
            if (process has local data in BCs c_r − 1 : c_r) then
                apply Q_{k−(i−i_0)/2}^(i) from right to BCs c_r − 1 : c_r
            end if
            i ← i + 2
        end for
        if (j = P) then
            windupPhase ← false
        end if
        if (windupPhase) then
            j ← j + 1
        else
            k ← k − 1;    j ← j − 1
        end if
    end while
else
    if (process has local data in BC N) then
        apply Q_{P+1}^(2N−2P) from right to BC N
    end if
    k ← P + 1;    j ← N
    while j ≤ N do
        i ← 2 + (j − k);    i_0 ← 2 + (j − k)
        for c_r ← j, j + 2, ..., N − 1 do
            if (process has local data in BCs c_l : c_l + 1) then
                apply Q_{k+(i−i_0)/2}^(i) from right to BCs c_l : c_l + 1
            end if
            i ← i + 2
        end for
        if (j = P + 1) then
            windupPhase ← false
        end if
        if (windupPhase) then
            j ← j − 1
        else
            k ← k + 1;    j ← j + 1
        end if
    end while
end if
```

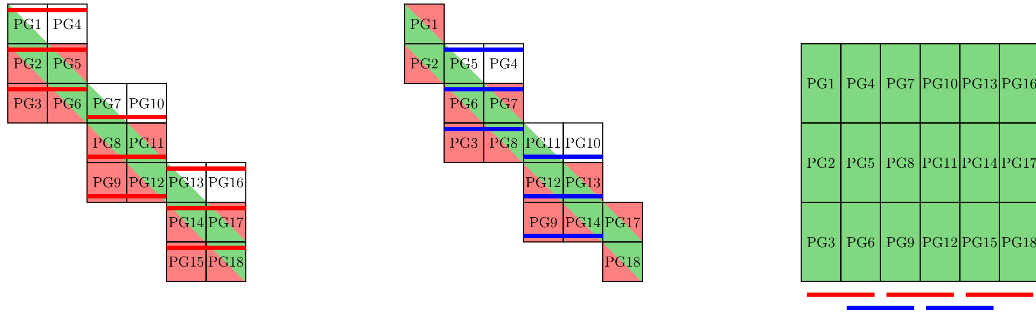


Figure 4.29: State 1 and 3 in right-sided orthogonal transformations and the matrix of right singular vectors. The gray and red lines indicate the collaboration between the block columns (and the regarding communicator). The block to process group allocation follows the process groups in state 1.

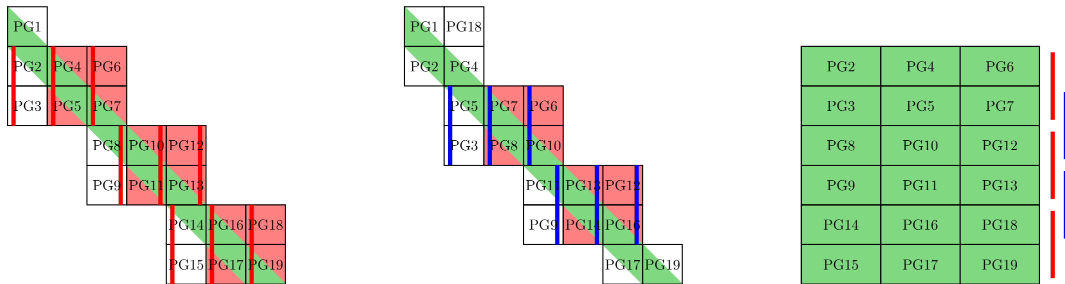


Figure 4.30: State 2 and 4 in left-sided orthogonal transformations and the matrix of left singular vectors. The gray and red lines indicate the collaboration between the block rows (and the regarding communicator). The block to process group allocation follows the process groups in state 2.

The process setup as used before for applying inversion steps and performing the bulge chasing can be fully reused in the backtransformation. Therefore, the same parallelization levels are available as before and the backtransformation should show the same parallel behavior.

Block and inter-block parallelization

Parallel algorithms on the level of pairs of block rows and block columns

Algorithms 61 and 62 have given the overall, parallel procedure to apply the backtransformation to the left and right singular vector matrices. The more fine grained algorithms performing the single updates on the level of pairs of block columns or pairs of block rows will be presented in the following.

Comparing to the SVD bulge chasing, a different variant of the CWY application has to be used for the backtransformation since in bulge chasing all orthogonal transformations are applied transposed. Algorithm 42 is presented in the context of the eigenvectors backtransformation and can also be used for the backtransformation of the left singular vectors. For the backtransformation of the right singular vectors, a modified version of Algorithm 39 as given in Algorithm 63 is used. The only difference is the transposed use of T which originates in the non-transposed application of $Q_k^{(i)}$ needed for the backtransformation of the right singular vectors.

Algorithm 63 Parallel blocked transposed application of Householder vectors from right.

function $A_{<p_r,p_c>[i,j]} \leftarrow \text{ParallelCWYtransposedRight}(A_{<p_r,p_c>[i,j]},\ Y'_{<*,p_c>[j]},\ T_{<*,*>})$
  $Z_{<p_r,*>[j]} \leftarrow Y'_{[j]} \cdot T^T$
  $U_{<*,p_c>[i]j} \leftarrow A_{[i,j]} \cdot Z_{[j]}$
  $U_{<*,p_c>[i]} \leftarrow \text{MPI_ALLREDUCE}\ (U_{<*,p_c>[i]j})$
  $A_{[i,j]} \leftarrow A_{[i,j]} - U_{[i]} \cdot Y'_{[j]}$
end function

Compact CWY formulations are used in the tile-wise application of the Householder vectors. For the eigenvalue backtransformation, Algorithm 43 gives an implementation which can be used in the backtransformation of the left singular vectors for the lower matrix half. This can be extended to be usable for the upper matrix half as well by changing some indices as given in Algorithm 64. Clearly, this change could have been incorporated in Algorithm 43 but might cause confusion there. Depending on whether applying a transformation originated by QL or QR, different rows in the singular vector matrix U and in the Householder vector matrix Y have to be addressed. If storing only τ instead of T, an additional computation of T has to be added before the compact WY is used.

Similar for the right singular vectors and the right-sided application of Householder vectors The application to a pair of block columns is described by Algorithm 65. The columns to update are adapted depending on if a LQ or RQ decomposition is applied to the matrix of right singular vectors V^T.

Algorithm 64 Parallel blocked non-transposed left-sided application of orthogonal transformations to left singular vectors.

function $U_{<p_r,p_c>[i,j]} \leftarrow$ ParallelBlockedLeftSidedApplQ($U_{<p_r,p_c>[i,j]},\ Y_{<p_r,*>[i]},\ T_{<*,*>}$)
  for $j_1 \leftarrow 1,\ b+1,\ 2b+1,\ \ldots,\ n_b$ **do**
    if (Upper matrix half) **then**
      $i_1 \leftarrow 1; \quad i_2 \leftarrow 2n_b - j_1 + 1$
    else
      $i_1 \leftarrow j_1; \quad i_2 \leftarrow 2n_b$
    end if
    $j_2 \leftarrow \min(n_b,\ j_1 + b - 1)$
    $U_{[i,j]}(i_1 : i_2, :) \leftarrow$ ParallelCWYtransposedLeft($U_{[i,j]}(i_1 : i_2, :), Y_{[i]}(i_1 : i_2, j_1 : j_2),\ T$)
  end for
end function

Algorithm 65 Parallel blocked non-transposed right-sided application of orthogonal transformations to right singular vectors.

function $V^T_{<p_r,p_c>[i,j]} \leftarrow$ ParallelBlockedRightSidedApplQ($V^T_{<p_r,p_c>[i,j]},\ Y'_{<*,p_c>[j]},\ T_{<*,*>}$)
  for $j_1 \leftarrow 1,\ b+1,\ 2b+1,\ \ldots,\ n_b$ **do**
    if (Upper matrix half) **then**
      $i_1 \leftarrow 1; \quad i_2 \leftarrow 2n_b - j_1 + 1$
    else
      $i_1 \leftarrow j_1; \quad i_2 \leftarrow 2n_b$
    end if
    $j_2 \leftarrow \min(n_b,\ j_1 + b - 1)$
    $V^T_{[i,j]}(:, i_1 : i_2) \leftarrow$ ParallelCWYtransposedRight($V^T_{[i,j]}(:, i_1 : i_2), Y'_{[j]}(i_1 : i_2, j_1 : j_2),\ T$)
  end for
end function

The necessary distribution of Y has been sparred out in the algorithmic descriptions of Algorithms 64 and 65 for simplicity. It has been described during the process and data structures part, Section 4.5.1.

4.5.2 Backtransformation matrix approach

As in the backtransformation of eigenvectors, accumulating backtransformation matrices is also a possibility for the backtranformation of singular vectors. The singular vectors can be obtained by multiplying the backtransformation matrices ${}^{\mathrm{o}}\widetilde{Q}$ and ${}^{\mathrm{e}}\widetilde{Q}^T$ with the singular vectors of the transformed problem $\widetilde{W}$:

$$U = {}^{\mathrm{o}}\widetilde{Q} \cdot \widetilde{U} \tag{4.16}$$

$$V^T = \widetilde{V}^T \cdot {}^{\mathrm{e}}\widetilde{Q} \tag{4.17}$$

The backtransformation matrices can be obtained by

$${}^{\mathrm{o}}\widetilde{Q} = I \cdot {}^{\mathrm{o}}Q_N^{(\cdot)} \cdots {}^{\mathrm{o}}Q_{P+1}^{(\cdot)} \cdot {}^{\mathrm{o}}Q_1^{(\cdot)} \cdots {}^{\mathrm{o}}Q_P^{(\cdot)} \tag{4.18}$$

$${}^{\mathrm{e}}\widetilde{Q} = {}^{\mathrm{e}}Q_P^{(\cdot)} \cdots {}^{\mathrm{e}}Q_1^{(\cdot)} \cdot {}^{\mathrm{e}}Q_{P+1}^{(\cdot)} \cdots {}^{\mathrm{e}}Q_N^{(\cdot)} \cdot I. \tag{4.19}$$

This leads to accumulating the orthogonal transformations during the bulge chasing. However, even for the eigenvectors, this approach is only favorable if all or almost all eigenvectors are computed. Therefore, following this approach is not seen as an practical alternative to the pipelining approach and hence the backtransformation matrix approach is not further elaborated. However, all steps are analogous to the procedure of eigenvalue computation.

5 Numerical Analysis

In this chapter the parallel implementations are analyzed on a theoretical basis. The findings can be used to estimate and evaluate the computational results on supercomputers, for the optimal choice of parameters like P and for the optimal distribution of the available processes.

5.1 Twisted Crawford algorithm

5.1.1 Modeling of the speedup of the block and inter-block parallelization

To estimate the expectable speedups during the generation of one $Q_k^{(i)}$ and the respective left and right-sided application, the single operations have to be analyzed. The QR decomposition and the left-sided update work on a 2×3 block grid with six independent process groups, the right-sided update on a 3×2 block grid. Every process group consists of a 2D process grid of $p_r \times p_c$ processes. The local size of a block on a process can hence be estimated by $\frac{n_b}{p_r} \times \frac{n_b}{p_c}$.

Model for the estimation of the runtime of parallel algorithms

To model the runtime of the parallel algorithms in Chapter 4, a simple model is used. It comprises modeling Flops, memory transfer and communication.
For modeling communication the simple latency-bandwidth model is used. The time to send a message of size m from one process to another consists of the constant latency time and the message-size-dependent time for the actual sending process:

$$t = m \cdot t_{\text{Word}} + t_{\text{Msg}} \tag{5.1}$$

Latency includes all parts that do not scale with the size of the message, e.g. setting up the message, handshake between the processes and further more. Clearly, this model includes many assumptions: It presumes to have constant bandwidth for all messages, it ignores the topology of the network and it disregards the possible saturation of the network path.
For collective communication, the model gets an additional factor to reflect the number of involved processes, p, [19]:

$$t = (m \cdot t_{\text{Word}} + t_{\text{Msg}}) \lceil \log(p) \rceil \tag{5.2}$$

Since MPI_ALLREDUCE consists of a reduce and a broadcast operation, MPI_ALLREDUCE is modeled as $t = 2(m \cdot t_{\text{Word}} + t_{\text{Msg}}) \lceil \log(p) \rceil$.

The cache hierarchy and memory transfer is modeled in a very much simplified way by the external memory model [1]. The hierarchy consists of an infinitely fast, internal memory of limited size to which the CPU has direct access. Additionally, an external memory of infinite size exists in which all data is located initially. Data has to be transferred before usage from

the external to the internal memory. This data transfer is limited by a certain memory bandwidth which leads to the time to load a double number from slow to fast memory, t_{Mem}.

The time of the CPU to compute one Flop is given by t_{Flops}. All the hardware parameters used for modeling, t_{Flops}, t_{Mem}, t_{Word} and t_{Msg}, reduce complex hardware features in an extensive way. Thus, the runtime model delivers a simplified view on the algorithms and their performance but will provide necessary insights to get a deeper understanding of the algorithms and derive strategies on how to invest the available computational resources.

The time to generate one Householder vector following Algorithm 32 can be estimated by

$$t_{\text{HH}}^{\text{row}} \approx 3\frac{n_b}{p_r}t_{\text{Flops}} + 2\frac{n_b}{p_r}t_{\text{Mem}} + 4\lceil\log(2p_r)\rceil t_{\text{Word}} + 2\lceil\log(2p_r)\rceil t_{\text{Msg}}. \tag{5.3}$$

The application of a single Householder vector to a matrix in the QR context takes place during computing the QR of a tile. The vector is applied to the trailing tile which has m columns. Hence, the maximal local size of the trailing tile on a process is $\frac{n_b}{p_r} \times \frac{m}{p_c}$. The time to carry out this transformation, which is described in Algorithm 33, can be approximated by

$$\begin{aligned} t_{\text{HHAppl}}^{\text{row}}(m) \approx\ & 4\frac{n_b m}{p_r p_c}t_{\text{Flops}} + (\frac{n_b m}{p_r p_c} + \frac{n_b}{p_r} + \frac{m}{p_c})t_{\text{Mem}} \\ & + 2\frac{m}{p_c}\lceil\log(2p_r)\rceil t_{\text{Word}} + 2\lceil\log(2p_r)\rceil t_{\text{Msg}}. \end{aligned} \tag{5.4}$$

Both formulas can be used to compute the approximated time for computing a QR decomposition of b columns:

$$\begin{aligned} t_{\text{QR}} &= b \cdot t_{\text{HH}}^{\text{row}} + \sum_{k=1}^{b-1} t_{\text{HHAppl}}^{\text{row}}(b-k) \\ &\approx (2\frac{n_b(b^2-b)}{p_r p_c} + 3\frac{n_b b}{p_r})t_{\text{Flops}} + (2\frac{n_b(b^2-b)}{p_r p_c} + 2\frac{n_b b}{p_r})t_{\text{Mem}} \\ &+ (\ (\frac{b^2-b}{p_c} + 4b)\lceil\log(2p_r)\rceil + \frac{n_b b}{p_r}\lceil\log(p_c)\rceil)t_{\text{Word}} \\ &+ (4b\lceil\log(2p_r)\rceil + b\lceil\log(p_c)\rceil)t_{\text{Msg}} \end{aligned} \tag{5.5}$$

The time for the generation of T, as described in Algorithm 35, can be estimated by

$$\begin{aligned} t_{\text{genT}}^{\text{row}} \approx\ & \frac{1}{3}b^3 + \frac{n_b b^2}{p_r} - \frac{5}{4}b^2 - \frac{1}{12}b)t_{\text{Flops}} + (\frac{n_b b}{p_r} + 2b^2)t_{\text{Mem}} \\ & + b^2\lceil\log(2p_r)\rceil t_{\text{Word}} + 2\lceil\log(2p_r)\rceil t_{\text{Msg}}. \end{aligned} \tag{5.6}$$

For applying b Householder vectors by CWY from left (Algorithm 36) to a matrix of maximal

local size $\frac{n_b}{p_r} \times \frac{m}{p_c}$, the time can be estimated by

$$\begin{aligned} t_{\text{CWYleft}} \approx & (4\frac{n_b m b}{p_r p_c} + \frac{n_b b^2}{p_r} - b(\frac{0.5 n_b}{p_r} + \frac{m}{p_c}) t_{\text{Flops}} \\ & + (\frac{n_b m}{p_r p_c} + 2\frac{n_b b}{p_r} + \frac{m b}{p_c} + b^2) t_{\text{Mem}} \\ & + 2\frac{m b}{p_c} \lceil \log(2p_r) \rceil t_{\text{Word}} + 2\lceil \log(2p_r) \rceil t_{\text{Msg}} \end{aligned} \tag{5.7}$$

and for applying b Householder vectors by CWY from right (Algorithm 39) to a matrix of maximal local size $\frac{m}{p_r} \times \frac{n_b}{p_c}$, the following estimate can be used:

$$\begin{aligned} t_{\text{CWYright}} \approx & (4\frac{n_b m b}{p_r p_c} + \frac{n_b b^2}{p_c} - b(\frac{m}{p_r} + \frac{0.5 n_b}{p_c}) t_{\text{Flops}} \\ & + (\frac{n_b m}{p_r p_c} + 2\frac{n_b b}{p_c} + \frac{m b}{p_r} + b^2) t_{\text{Mem}} \\ & + 2\frac{m b}{p_r} \lceil \log(2p_c) \rceil t_{\text{Word}} + 2\lceil \log(2p_c) \rceil t_{\text{Msg}} \end{aligned} \tag{5.8}$$

The algorithm to perform the QR decomposition of the k-th tile of a block and to apply the obtained Householder vectors by CWY to the trailing block is described in Algorithm 37. The runtime, neglecting the time for sending the Householder vectors to the right blocks, can be estimated by

$$\begin{aligned} t_{\text{QRblocks}}(k) \approx & (\frac{4n_b^2 b + n_b b^2(2-4k)}{p_r p_c} + \frac{2n_b b^2}{p_r} + \frac{b^2 k}{p_c} + \frac{b^3}{3}) t_{\text{Flops}} \\ & + (\frac{n_b^2 + n_b b^2 - n_b b k + n_b b}{p_r p_c} + \frac{5 n_b b}{p_r} + \frac{n_b b - b^2 k}{p_c} + 3b^2) t_{\text{Mem}} \\ & + (\ (\frac{2n_b b - b^2(2k-1)}{p_c} + b^2) \lceil \log(2p_r) \rceil + \frac{n_b b}{p_r} \lceil \log(p_c) \rceil) t_{\text{Word}} \\ & + (\ (4b+4) \lceil \log(2p_r) \rceil + b \lceil \log(p_c) \rceil) t_{\text{Msg}}. \end{aligned} \tag{5.9}$$

The runtime of the right blocks, which solely apply the Householder vectors, stays constant over the tiles and is estimated as

$$\begin{aligned} t_{\text{blocks}}^{\text{ri}} \approx & (\frac{4n_b^2 b}{p_r p_c} + \frac{n_b b^2}{p_r} + \frac{b^3}{3}) t_{\text{Flops}} + (\frac{n_b^2}{p_r p_c} + \frac{3 n_b b}{p_r} + \frac{n_b b}{p_c} + 3b^2) t_{\text{Mem}} \\ & + (\frac{2 n_b b}{p_c} + b^2) \lceil \log(2p_r) \rceil t_{\text{Word}} + 4\lceil \log(2p_r) \rceil t_{\text{Msg}}. \end{aligned} \tag{5.10}$$

This estimate is again not considering the time for waiting and receiving Y and τ.

For putting together the pieces for the overall QR step according to Algorithm 37, the following considerations have to be made:

- Only the left blocks perform the QR, the other four blocks solely apply the Householder

transformations.

- While for the four right blocks the number of columns to process in $CWYleft$ stays constant, it decreases in the QR blocks.
- The two QR blocks do not have to compute T and do not have to run $CWYleft$ after processing the last tile.
- The four blocks that only apply the transformation have to wait for receiving Y from the QR blocks.

It turns out that the runtime of the QR blocks is initially higher than the runtime of the right blocks and, by the decreasing number of columns for $CWYleft$ in the QR blocks, the situation changes at a certain point. After this point, the runtime of the right blocks dominates the steps and hence determines the runtime of a QR step.

Using Equations (5.9) and (5.10), an estimate for the overall time of the QR step can be derived. Necessary for that is an estimation when the right blocks start dominating the runtime. This estimate is denoted by o and gives the percentage of tiles of the block after which the QR blocks do not dominate the runtime anymore. It turns out that o depends highly on nb, b, the process grid of a PG and on the numbers t_{Flops}, t_{Mem}, t_{Word} and t_{Msg}. On SuperMuc-NG, o can be found between 0.33 and 0.5 for n_b of size 2000 with quadratic process group setups and $b = 175$, for $n_b = 4000$ and the same conditions between 0.5 and 0.75.
The resulting formula to estimate the runtime of the QR step using o is given by

$$t_{\text{Qleft}} \approx \sum_{k=1}^{o\frac{n_b}{b}} t_{\text{QRblocks}}(k) + (1-o)\frac{n_b}{b} t_{\text{blocks}}^{\text{ri}} + \frac{n_b b}{p_r} t_{\text{Word}} + \frac{n_b}{b} t_{\text{Msg}}. \tag{5.11}$$

For the right-sided updates (Algorithm 40), no QR decomposition has to be computed and all blocks work simultaneously. T does not need to be computed again since it has been stored during the left-sided application of the orthogonal transformation. Hence, besides running ParallelCWYright, Y and T have only to be communicated. For the right-sided update one block acts as surrogate block. This block does not need to receive Y and T since it acted in the left-sided update as surrogate block as well. Thus, the runtime for the right-sided application of an orthogonal transformation can be estimated by

$$\begin{aligned} t_{\text{Qright}} \approx & \left(\frac{4n_b^3}{p_r p_c} + \frac{n_b^2 b}{p_c}\right) t_{\text{Flops}} + \left(\frac{n_b^3/b}{p_r p_c} + \frac{n_b^2}{p_r} + \frac{2n_b^2}{p_c} + n_b b\right) t_{\text{Mem}} \\ & + \left(\frac{2n_b^2}{p_r}\lceil\log(2p_c)\rceil + \frac{n_b b}{p_r}\lceil\log(2p_r)\rceil + \frac{n_b^2 + 0.5 n_b b^2}{p_c} + 2n_b/b\lceil\log(2p_c)\rceil\right) t_{\text{Word}} \\ & + \left(2n_b/b\lceil\log(2p_c)\rceil + \frac{n_b}{p_r} + p_r\right) t_{\text{Msg}}. \end{aligned} \tag{5.12}$$

This formula uses an adapted version of the runtime estimation for transposing Y, which is given in [4] by $\lceil\log(2p_r)\rceil(\frac{n_b b}{p_r} t_{\text{Word}} + \frac{lcm}{p_c} t_{\text{Msg}})$. lcm in the formula is the least common multiple of p_r and p_c and hence $\frac{lcm}{p_c} \leq p_r$. Thus, the runtime for transposing Y can be estimated by

$\lceil\log(2p_r)\rceil(\frac{n_b b}{p_r}t_{\text{Word}} + p_r t_{\text{Msg}})$.

The backtransformation of eigenvectors (Algorithm 43) consists of distributing Y and T to the regarding blocks and by using a modified version of the compact WY formulation to apply the Householder vectors from left. The estimated runtime for transforming w eigenvectors is given by

$$t_{\text{backtrafo}} \approx (\frac{4n_b^2 w}{p_r p_c} + \frac{n_b^2 b}{p_r})t_{\text{Flops}} + (\frac{n_b^2 w/b}{p_r p_c} + \frac{2n_b^2}{p_r} + \frac{n_b w}{p_c} + n_b b)t_{\text{Mem}} + (\frac{2n_b w}{p_c}\lceil\log(2p_r)\rceil + \frac{n_b^2}{p_r} + n_b b)t_{\text{Word}} + (\frac{2n_b}{b}\lceil\log(2p_r)\rceil + \frac{n_b}{p_r})t_{\text{Msg}}. \quad (5.13)$$

Inspecting Equations (5.11) to (5.13), it can be seen that most of the terms scale in p_r, p_c or both directions. Naturally, this leads to the use of a 2D process grid. Some of the terms do not scale at all or only rarely. Analyzing the derived formulas with the hardware characteristics of SuperMuc-NG allows to obtain that Flops usually scale very well and memory transfer is in the most cases comparably cheap and thus does not have a huge impact. Similar for the time to initiate communication which barely scales but has only small impact. The communication of data however, scales generally the worst and will hence become the bottleneck in block scaling.

5.1.2 Modeling of the speedup of the pipelining approach

Pairs of block rows during the generation of the QR and the regarding left-sided application and pairs of block columns during the right-sided application of the QR are independent of each other and can be executed in parallel. The parallel execution is achieved by distributing block columns cyclically to process group columns. A pair of block columns is operated by a pair of PGCs, the next pair of block columns by another pair of PGC and so on. Hence, both can update their block columns independently.

Having more PGCs available allows therefore to execute more operations simultaneously. The pipelining approach, however, consists of a wind-up and a wind-down phase. During these phases, the number of block column pairs to work on grows or reduces by every step and hence, also the computational work that can be done in parallel rises and falls. Since the number of PGCs is constant during the algorithm, some of the PGC might stay idle for some steps until they start working.

Formulas for the length of the pipeline are developed in Section 4.2.1. Equation (4.2) gives the length of the pipeline in the wind-up phase after applying inversion step k by

$$l_{\text{pipe}}(k) = \lfloor\frac{N-k+1}{2}\rfloor.$$

The length of the pipeline in the wind-down phase is described in Equation (4.3) by

$$l_{\text{pipe}}(i_{\min}) = \lfloor\frac{N-P-i_{\min}+1}{2}\rfloor.$$

Since the length of the pipeline determines the number of parallel executable left or right-sided orthogonal transformations, the accumulation of all lengths of the pipeline within the algorithm can be used to count the overall number of left or right-sided applications of orthogonal transformations:

$$c^{\mathrm{QR}} = 1 + \sum_{k=P+1}^{N} l_{\mathrm{pipe}}(k) + \sum_{i_{\min}=2}^{N-P-1} l_{\mathrm{pipe}}(i_{\min}) \tag{5.14}$$

Starting the second sum at 2 is a consequence of counting the parallel orthogonal transformations after inversion step k in $l_{\mathrm{pipe}}(k)$. This already includes $Q_{P+1}^{(1)}$ in step $P+1$. The additional 1 is a consequence of clearing the last block row and column only for $k = P+1$. The formula computes the number of orthogonal transformations in the lower matrix half. It can be used for the upper matrix half by flipping the matrix or by adapting the indices: k has to run from 1 to P and $i_{\min}$ from 2 to $P-1$.
In the upper matrix half, one QR step and its application can be saved since inversion step P is reduced and contains only of the application of D_P, no fill-in is generated by E_P. Hence, $Q_P^{(1)}$ can be skipped and the chasing of this inversion step can start with $Q_P^{(2)}$.

To compute the speedup gained by using more PGCs, all orthogonal transformations that can be executed in parallel have to be distributed to the available pairs of PGCs n_{PGCpairs}:

$$t_{\mathrm{pipe}}(k, n_{\mathrm{PGCpairs}}) = \lceil \frac{l_{\mathrm{pipe}}(k)}{n_{\mathrm{PGCpairs}}} \rceil \cdot t_{\mathrm{orth}} \tag{5.15}$$

and

$$t_{\mathrm{pipe}}(i_{\min}, n_{\mathrm{PGCpairs}}) = \lceil \frac{l_{\mathrm{pipe}}(i_{\min})}{n_{\mathrm{PGCpairs}}} \rceil \cdot t_{\mathrm{orth}} \tag{5.16}$$

compute the time it takes to apply l_{pipe} orthogonal transformations when using n_{PGCpairs} pairs of PGCs. $t_{\mathrm{orth}} = t_{\mathrm{Qleft}} + t_{\mathrm{Qright}}$ denotes the time to generate and apply an orthogonal transformation from left and right.

The time for running the full pipeline with n_{PGCpairs} PGC pairs can therefore be computed by

$$t^{\mathrm{QR}}(n_{\mathrm{PGCpairs}}) \approx (1 + \sum_{k=P+1}^{N} \lceil \frac{l_{\mathrm{pipe}}(k)}{n_{\mathrm{PGCpairs}}} \rceil + \sum_{i_{\min}=2}^{N-P-1} \lceil \frac{l_{\mathrm{pipe}}(i_{\min})}{n_{\mathrm{PGCpairs}}} \rceil) \cdot t_{\mathrm{orth}}. \tag{5.17}$$

An explicit upper matrix half formulation can be given by

$$t^{\mathrm{QR}}(n_{\mathrm{PGCpairs}}) \approx (1 + \sum_{k=1}^{P} \lceil \frac{l_{\mathrm{pipe}}(k)}{n_{\mathrm{PGCpairs}}} \rceil + \sum_{i_{\min}=2}^{P-1} \lceil \frac{l_{\mathrm{pipe}}(i_{\min})}{n_{\mathrm{PGCpairs}}} \rceil) \cdot t_{\mathrm{orth}} \tag{5.18}$$

using

$$l_{\mathrm{pipe}}(k) = \lfloor \frac{k}{2} \rfloor \tag{5.19}$$

and

$$l_{\mathrm{pipe}}(i_{\min}) = \lfloor \frac{P - i_{\min} + 1}{2} \rfloor. \tag{5.20}$$

The model $t^{\mathrm{QR}}(n_{\mathrm{PGCpairs}})$ ignores the time for applying the inversion steps and the reduced

n_{PGCpairs}	1	2	4	5	10	20	30	40	50
$n_b/n = 1\%$									
Relative time	100%	51.0%	26.5%	21.6%	11.8%	7.11%	5.55%	4.75%	4.00%
Speedup	1	1.96	3.77	4.62	8.45	14.1	18.0	21.1	25.0
Efficiency	100%	98.0%	94.2%	92.5%	84.4%	70.3%	60.0%	52.7%	50.0%
$n_b/n = 2\%$									
Relative time	100%	52.0%	28.1%	23.3%	14.0%	9.38%	7.99%	7.99%	7.99%
Speedup	1	1.92	3.56	4.29	7.13	10.7	12.5	12.5	12.5
Efficiency	100%	96.1%	89.1%	85.7%	71.3%	53.3%	41.7%	31.3%	25.0%
$n_b/n = 5\%$									
Relative time	100%	55.5%	33.5%	28.8%	19.9%	19.9%	19.9%	19.9%	19.9%
Speedup	1	1.80	2.98	3.47	5.03	5.03	5.03	5.03	5.03
Efficiency	100%	90.1%	74.6%	69.5%	50.3%	25.2%	16.8%	12.6%	10.1%

Table 5.1: Theoretical speedup and parallel efficiency implied by the pipelining algorithm for different bandwidth-to-matrix-size ratios under increasing the number of pairs of PGCs used for the computation. Synchronization effects imposed by splitting the matrix in an upper and a lower matrix half are not considered.

computational time for applying the very last orthogonal transformation to clear the last block column and block row. However, since the computational cost of the inversion steps is rather small and the impact of the last orthogonal transformation as well as skipping $Q_P^{(1)}$, the model can be used to compute a good estimate of the speedup when adding pairs of PGCs to a matrix half.

Table 5.1 shows such computations for three different bandwidth-to-matrix-size ratios using only an upper or lower matrix half ($P = 0$ or $P = N$). Baseline is the number of timesteps necessary to perform all the orthogonal transformations with one pair of PGCs. The number of transformations and the number of timesteps is 4951 for $n_b/n = 1\%$. By adding another pair of PGC, the amount of timesteps decreases to 2526, which is 51% of the timesteps when using one pair. The loss of 1% is caused by the idle times that occur due to the pipeline.
The parallel efficiency gives a measure on how well the parallel resources are exploited. Due to the idling PGCs the efficiency decreases when using more and more PGCs. However, the critical point of 50% efficiency is only reached when using as many PGCs as block columns are available. For the cases $n_b/n = 2\%$ and $n_b/n = 5\%$ even more PGCs are added. These PGCs will never work and stay idle the whole time. Hence, the efficiency drops further.

The pipelining approach for the backtransformation follows the same principles and produces the same formulas and numbers as given in Equations (5.17) and (5.18) and Table 5.1. The approach of the backtransformation matrix was due to the rare cases where it is superior compared to the pipelining approach not followed and hence no runtime estimates have been

developed.

5.1.3 Modeling of the speedup by splitting the matrix in upper and lower half

The number of left and right-sided orthogonal transformations for each matrix half can be computed by Equations (5.17) and (5.18). A more detailed analysis has to take into account the synchronization between both matrix halves when applying their last inversion steps (see Section 4.2.2). The solution of this task cannot be easily given by a formula since the problem depends on many parameters that influence each other. However, a small simulation has been implemented to compute the number of necessary timesteps to finish the algorithm.
For a given bandwidth-to-matrix-size ratio and a fixed number of available pairs of PGCs, this simulation was run and some of the results are exemplarily shown in Figure 5.1. For every distribution of the available pairs of PGCs to the upper or lower matrix halves the number of computation steps was computed for different positions of the twist block P. It can be seen that choosing $P = N/2$ and splitting the available PGCs equally to the matrix halves gives the best result.

The comparison of the computation steps of $P = 0$ using all PGCs in the lower matrix half and $P = N/2$ with equally distributed PGCs gives the attainable speedup by using twisted factorization. For the inspected bandwidth-to-matrix-size ratio of 1%, the theoretical speedup was found to be 1.98 for 10 pairs and 20 pairs of PGCs. For a bandwidth-to-matrix-size ratio of 2%, the speedup was found to be 1.96 to 1.97, for 5% the speedup was between 1.93 and 1.94. Hence, only by using twisted factorization instead of Cholesky factorization and the regarding parallel execution of the upper and lower matrix half, the computation time can be reduced by approximately 50%.

The interference of the upper and lower matrix half in the backtransformation pipeline is limited. The orthogonal transformations can be applied to the eigenvector matrix without exchange of the upper and lower matrix half. Only the last inversion step in the lower matrix half requires exchange: To apply S_{P+1}, S_P must have been applied. Since the application of the inversion steps can be performed after applying all orthogonal transformations, no synchronization takes place during the orthogonal transformations. Hence, the exchange will affect only the less costly part of applying inversions steps and the computational time will reduce also here by a factor of approximately 50%.

5.2 Twisted SVD algorithm

5.2.1 Modeling of the speedup of the block and inter-block parallelization

Please refer to Section 5.1.1 for the description of the parallel model used in this work to model the runtime.

The QR algorithm and hence the estimates for speedup are the same in the GSVD algorithm and in the GEP algorithm. For the QL algorithm, only the direction of where to start in a

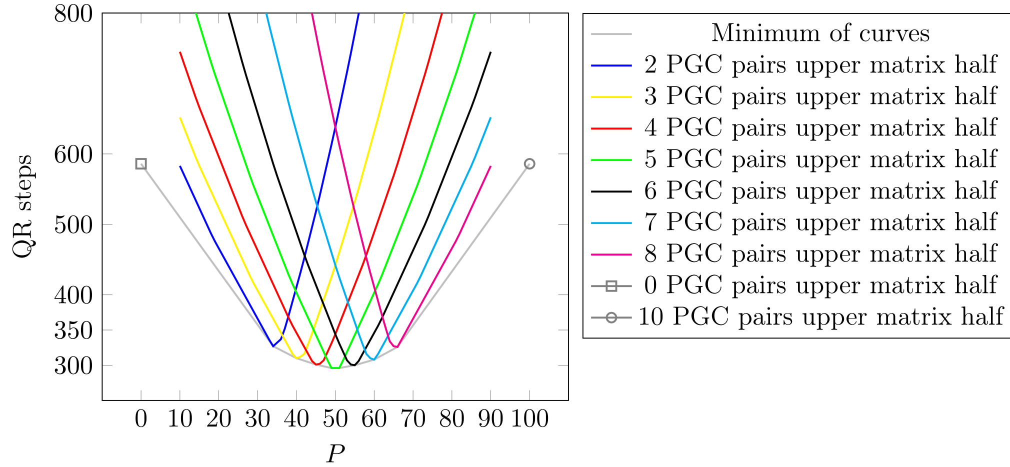


Figure 5.1: Number of resulting parallel steps over twist block P for different distributions of PGCs to the upper and lower matrix half. In total, 10 pairs of PGCs are available for computing a matrix with bandwidth-to-matrix-size ratio 1%. The minimum number of steps is obtained for 5 pairs of PGC in the upper and 5 pairs of PGC in the lower matrix half together with P in the range 49 to 51.

block changes and hence

$$t_{\text{QLblocks}} \approx t_{\text{QRblocks}} \tag{5.21}$$

can be assumed.

For the right-sided transformations the strided access to the matrix might impact the performance during the computation of the Householder vectors. For the blocked and unblocked application of Householder vectors, the access to the matrix columns and therefore the cache lines will be consecutive. Hence, no slowdown is to expect there. Considering the time spent to compute a Householder vector and the time to apply it in blocked or unblocked fashion, the generation of the vector is rather cheap. Hence, this slowdown is ignored and the time for computing a Householder vector can be estimated by

$$t_{\text{HH}}^{\text{col}} \approx 3\frac{n_b}{p_c}t_{\text{Flops}} + 2\frac{n_b}{p_c}t_{\text{Mem}} + 4\lceil\log(2p_c)\rceil t_{\text{Word}} + 2\lceil\log(2p_c)\rceil t_{\text{Msg}}. \tag{5.22}$$

When applying a Householder vector from right in an unblocked way to a matrix with maximal local size $\frac{m}{p_r} \times \frac{n_b}{p_c}$, the time can be estimated by

$$\begin{aligned} t_{\text{HHAppl}}^{\text{col}}(m) \approx\ & 4\frac{n_b m}{p_r p_c}t_{\text{Flops}} + (\frac{n_b m}{p_r p_c} + \frac{m}{p_r} + \frac{n_b}{p_c})t_{\text{Mem}} \\ & + 2\frac{m}{p_r}\lceil\log(2p_c)\rceil t_{\text{Word}} + 2\lceil\log(2p_c)\rceil t_{\text{Msg}}. \end{aligned} \tag{5.23}$$

These two formulas can be used to estimate the computation time of an unblocked LQ or RQ

decomposition of b columns:

$$\begin{aligned} t_{\text{RQ}} \approx t_{\text{LQ}} &= b t_{\text{HH}}^{\text{col}} + \sum_{k=1}^{b-1} t_{\text{HHAppl}}^{\text{col}}(b-k) \\ &\approx (2\frac{n_b(b^2-b)}{p_r p_c} + 3\frac{n_b b}{p_c}) t_{\text{Flops}} + (2\frac{n_b(b^2-b)}{p_r p_c} + 2\frac{n_b b}{p_c}) t_{\text{Mem}} \\ &+ (\ (\frac{b^2-b}{p_r} + 4b)\lceil\log(2p_c)\rceil + \frac{n_b b}{p_c}\lceil\log(p_r)\rceil) t_{\text{Word}} \\ &+ (4b\lceil\log(2p_c)\rceil + b\lceil\log(p_r)\rceil) t_{\text{Msg}} \end{aligned} \tag{5.24}$$

The computation time for T differs in the right-sided update since Y is not distributed over the process rows but over the process columns. As a result, the time for the generation of T, as described in Algorithm 38, can be estimated by

$$\begin{aligned} t_{\text{genT}}^{\text{col}} \approx \frac{1}{3}b^3 + \frac{n_b b^2}{p_c} - \frac{5}{4}b^2 - \frac{1}{12}b) t_{\text{Flops}} + (\frac{n_b b}{p_c} + 2b^2) t_{\text{Mem}} \\ + b^2\lceil\log(2p_c)\rceil t_{\text{Word}} + 2\lceil\log(2p_c)\rceil t_{\text{Msg}}. \end{aligned} \tag{5.25}$$

The blocked right-sided CWY application has already been described by Equation (5.8).

The time of performing the RQ decomposition of the k-th tile and applying the Householder vectors by *CWYright* to the trailing block (and mirror-inverted for the LQ) is estimated by

$$\begin{aligned} t_{\text{RQblocks}}(k) \approx t_{\text{LQblocks}}(k) \approx (&\frac{4n_b^2 b + n_b b^2(2-4k)}{p_r p_c} + \frac{b^2 k}{p_r} + \frac{2n_b b^2}{p_c} + \frac{b^3}{3}) t_{\text{Flops}} \\ + (&\frac{n_b^2 + n_b b^2 - n_b bk - n_b b}{p_r p_c} + \frac{n_b b - b^2 k}{p_r} + \frac{5n_b b}{p_c} + 3b^2) t_{\text{Mem}} \\ + (&\ (\frac{2n_b b + b^2(1-2k)}{p_r} + b^2)\lceil\log(2p_r)\rceil + \frac{n_b b}{p_c}\lceil\log(p_r)\rceil) t_{\text{Word}} \\ + (&\ (4b+4)\lceil\log(2p_c)\rceil + b\lceil\log(p_r)\rceil) t_{\text{Msg}}. \end{aligned} \tag{5.26}$$

The runtime for the upper blocks applying Q generated by RQ decomposition or for the lower block rows applying Q generated by LQ decomposition is estimated by

$$\begin{aligned} t_{\text{blocks}}^{\text{up}} \approx t_{\text{blocks}}^{\text{lo}} \approx (\frac{4n_b^2 b}{p_r p_c} + \frac{n_b b^2}{p_c} + \frac{b^3}{3}) t_{\text{Flops}} + (\frac{n_b^2}{p_r p_c} + \frac{n_b b}{p_r} + \frac{3n_b b}{p_c} + 3b^2) t_{\text{Mem}} \\ + (\ (\frac{2n_b b}{p_r} + b^2)\lceil\log(2p_c)\rceil + \frac{n_b b}{p_c}) t_{\text{Word}} + (2b\lceil\log(2p_c)\rceil) t_{\text{Msg}}. \end{aligned} \tag{5.27}$$

This estimate does not consider the time for communicating Y to the four blocks.

The derived estimates can be put together to obtain an overall estimate for performing a whole RQ or LQ step. Analogous to the QR decomposition in the twisted Crawford part, some considerations have to be made (here exemplarily for the RQ):

- The two bottom blocks perform the RQ, the four blocks above only apply the Householder transformations.

- The four top blocks have a constant number of rows in $CWYright$ whereas in RQ blocks have a decreasing number of rows to process by $CWYright$.
- In the last tile of the RQ, the RQ blocks do not have to compute T and to perform $CWYright$.
- The four blocks that only apply the transformations have to wait for receiving Y and τ from the RQ blocks.

Initially, the applying blocks have to wait for the RQ blocks before they can start to work. After some tiles the situation changes. Then the RQ blocks will be faster in computing the next RQ decomposition of a tile and applying the obtained Householder vectors to the trailing block as the four top blocks in applying a tile to their whole block. In the first phase, the RQ block determines the runtime, in the latter, the applying blocks.

An overall estimate for the RQ and the LQ decomposition can be derived by using Equations (5.26) and (5.27). The missing piece is the turning point when Equation (5.26) does not dominate the runtime anymore but Equation (5.27). This point is denoted by o. It gives the percentage of tiles of the block after which the dominance changes. Using this, the overall runtime of a RQ and a LQ step can be estimated:

$$t_{\text{Qright}} \approx \sum_{k=1}^{o\frac{n_b}{b}} t_{\text{RQblocks}}(k) + (1-o)\frac{n_b}{b} t^{\text{up}}_{\text{blocks}} + \frac{n_b b}{p_c} t_{\text{Word}} + \frac{n_b}{b} t_{\text{Msg}} \tag{5.28}$$

The backtransformation of the left-sided singular vectors is identical to the backtransformation of the eigenvectors. Hence, the time estimate given in Equation (5.13) can be used to estimate the computation time for applying the backtransformation of one $Q_k^{(i)}$ to the left singular vectors.

For the backtransformation of the right singular vector matrix a similar formula can be derived. Also here, T and Y have to be communicated and applied to the matrix containing w singular vectors. The time estimate is given by

$$\begin{aligned} t_{\text{backtrafo}} \approx{} & (\frac{4n_b^2 w}{p_r p_c} + \frac{n_b^2 b}{p_c}) t_{\text{Flops}} + (\frac{n_b^2 w/b}{p_r p_c} + \frac{n_b w}{p_r} + \frac{2n_b^2}{p_c} + n_b b) t_{\text{Mem}} \\ & + (\frac{2n_b w}{p_r} \lceil \log(2p_c) \rceil + \frac{n_b^2 + 0.5 n_b b^2 + n_b b}{p_c}) t_{\text{Word}} \\ & + (\frac{2n_b}{b} \lceil \log(2p_c) \rceil + \frac{n_b}{p_c}) t_{\text{Msg}}. \end{aligned} \tag{5.29}$$

Like the twisted Crawford algorithms, the twisted SVD algorithms will show the best scaling behavior when using both directions of the 2D process grid in the process groups. At some point, communication will become the bottleneck and the algorithms will stop scaling. The derived runtime estimations will usually give good estimates for the scaling behavior but cannot capture all the details. For example, some of the communication can be hidden by the use of non-blocking MPI calls. Therefore, the formulas help to estimate and understand the

scaling behavior but cannot deliver exact numbers for the runtime.

5.2.2 Modeling of the efficiency of the pipelining approach

Since the pipeline concept is the same for the twisted SVD algorithm as for the eigenvalue algorithm, the efficiency is rather similar. In Section 4.4.1 Equation (4.8) provides the length of the pipeline when computing and applying QR by

$$l_{\text{pipe}}^{\text{QR}}(k) = \lfloor \frac{N-k+1}{2} \rfloor,$$

Equation (4.9) by

$$l_{\text{pipe}}^{\text{LQ}}(k) = \lfloor \frac{N-k}{2} \rfloor$$

and Equation (4.12) provides the length of the pipeline in the regarding wind-down phase:

$$l_{\text{pipe}}^{\text{lo}}(i_{\text{min}}) = \lfloor \frac{N-P-\lfloor \frac{i_{\text{min}}+1}{2} \rfloor + 1}{2} \rfloor$$

The number of the overall orthogonal transformations can be computed by

$$c^{\text{QR}} = 1 + \sum_{k=P+1}^{N} l_{\text{pipe}}^{\text{QR}}(k) + \sum_{\substack{i_{\text{min}}=3 \\ i_{\text{min}}=i_{\text{min}}+2}}^{2N-2P-3} l_{\text{pipe}}^{\text{lo}}(i_{\text{min}}) \tag{5.30}$$

for the QR and by

$$c^{\text{LQ}} = 1 + \sum_{k=P+1}^{N} l_{\text{pipe}}^{\text{LQ}}(k) + \sum_{\substack{i_{\text{min}}=2 \\ i_{\text{min}}=i_{\text{min}}+2}}^{2N-2P-2} l_{\text{pipe}}^{\text{lo}}(i_{\text{min}}) \tag{5.31}$$

for the LQ transformations.

Similar for the upper matrix when having F in twisted shape and G in banded lower triangular shape. For this case, Equation (4.10) provides the length of the pipeline during the wind-up phase for QL by

$$l_{\text{pipe}}^{\text{QL}}(k) = \lfloor \frac{k+1}{2} \rfloor,$$

Equation (4.11) for RQ by

$$l_{\text{pipe}}^{\text{RQ}}(k) = \lfloor \frac{k-1}{2} \rfloor$$

and Equation (4.13) gives the regarding formula for the wind-down phase:

$$l_{\text{pipe}}^{\text{up}}(i_{\text{min}}) = \lfloor \frac{P-\lfloor \frac{i_{\text{min}}}{2} \rfloor + 1}{2} \rfloor$$

These formulas can be used to compute the overall number of QL and RQ transformations:

$$c^{\mathrm{QL}} = 1 + \sum_{k=P+1}^{N} l_{\mathrm{pipe}}^{\mathrm{QL}}(k) + \sum_{\substack{i_{\min}=3 \\ i_{\min}=i_{\min}+2}}^{2P-1} l_{\mathrm{pipe}}^{\mathrm{up}}(i_{\min}) \tag{5.32}$$

$$c^{\mathrm{RQ}} = 1 + \sum_{k=P+1}^{N} l_{\mathrm{pipe}}^{\mathrm{RQ}}(k) + \sum_{\substack{i_{\min}=2 \\ i_{\min}=i_{\min}+2}}^{2P-2} l_{\mathrm{pipe}}^{\mathrm{up}}(i_{\min}) \tag{5.33}$$

It has to be mentioned that the last orthogonal transformations are of reduced size but are counted as full size transformations (the one in the formula). Additionally, in the upper matrix half, one left and one right-sided transformation can be skipped directly after applying S_P. The latter is not skipped in the formula since it is complicated to exclude it and the impact on the overall result is limited. For the former, it can be argued that due to the block structure the computational cost per working process group is not that much less than in an ordinary step. Hence, the necessary computing time is comparable.

Analogous to Equations (5.17) and (5.18), formulas for the computation time relative to using one pair of PGCs can be derived for the twisted SVD algorithm. Contrary to the eigenvalue case, where only a QR decomposition is computed and applied from left and right, during the SVD a QR/LQ pair or a QL/RQ pair of orthogonal transformations is computed and applied from left or right. Due to the data layout (column mayor) and how the parallel algorithms are set up, different performance of left and right-sided transformations can occur. The time for the parallel computation of QR and LQ in the lower matrix half, when using n_{PGCpairs} pairs of process group columns, can be described by

$$\begin{aligned} t^{\mathrm{lo}}(n_{\mathrm{PGCpairs}}) \approx\; & t^{\mathrm{QR}} + t^{\mathrm{LQ}} + t^{\mathrm{QR}} \sum_{k=P+1}^{N} \lceil \frac{l_{\mathrm{pipe}}^{\mathrm{QR}}(k)}{n_{\mathrm{PGCpairs}}} \rceil + t^{\mathrm{LQ}} \sum_{k=P+1}^{N} \lceil \frac{l_{\mathrm{pipe}}^{\mathrm{LQ}}(k)}{n_{\mathrm{PGCpairs}}} \rceil \\ & + t^{\mathrm{QR}} \sum_{\substack{i_{\min}=3 \\ i_{\min}=i_{\min}+2}}^{2N-2P-3} \lceil \frac{l_{\mathrm{pipe}}^{\mathrm{lo}}(i_{\min})}{n_{\mathrm{PGCpairs}}} \rceil + t^{\mathrm{LQ}} \sum_{\substack{i_{\min}=2 \\ i_{\min}=i_{\min}+2}}^{2P-2} \lceil \frac{l_{\mathrm{pipe}}^{\mathrm{lo}}(i_{\min})}{n_{\mathrm{PGCpairs}}} \rceil. \end{aligned} \tag{5.34}$$

t^{QR} and t^{LQ} shall denote the time to perform one QR decomposition and its application from left or the time to perform one LQ decomposition and its application from right.

For the upper matrix half a similar formula can be given to estimate the computing time using t^{QL} and t^{RQ} as times for the regarding orthogonal transformations by

$$\begin{aligned} t^{\mathrm{up}}(n_{\mathrm{PGCpairs}}) \approx\; & t^{\mathrm{QL}} + t^{\mathrm{RQ}} + t^{\mathrm{QL}} \sum_{k=1}^{P} \lceil \frac{l_{\mathrm{pipe}}^{\mathrm{QL}}(k)}{n_{\mathrm{PGCpairs}}} \rceil + t^{\mathrm{RQ}} \sum_{k=1}^{P} \lceil \frac{l_{\mathrm{pipe}}^{\mathrm{RQ}}(k)}{n_{\mathrm{PGCpairs}}} \rceil \\ & + t^{\mathrm{QL}} \sum_{\substack{i_{\min}=3 \\ i_{\min}=i_{\min}+2}}^{2P-1} \lceil \frac{l_{\mathrm{pipe}}^{\mathrm{up}}(i_{\min})}{n_{\mathrm{PGCpairs}}} \rceil + t^{\mathrm{RQ}} \sum_{\substack{i_{\min}=2 \\ i_{\min}=i_{\min}+2}}^{2P-2} \lceil \frac{l_{\mathrm{pipe}}^{\mathrm{up}}(i_{\min})}{n_{\mathrm{PGCpairs}}} \rceil. \end{aligned} \tag{5.35}$$

Computing the numbers for Equations (5.34) and (5.35) gives, besides minor differences, the same numbers as listed in Table 5.1. The same numbers are found for QR, LQ and RQ. Only for QL differences occur since it has to perform more steps and hence obtains slightly different numbers. Actually, for QL, speedup and efficiency are slightly better (the speedup differs by up to 1%, 2% and 5% for the bandwidth-to-matrix-size ratios 1%, 2% and 5%).

5.2.3 Modeling of the speedup by having twisted matrices

The influence of having a twisted matrix or a banded lower triangular matrix in S can also be modeled for the SVD algorithm. In Section 4.4.2 it was found that the only influence of the two matrix halves on each other is the application of inversion step S_P to one block in the lower matrix half. Since this update can take place at any time, e.g. when the process group would be idle anyway, both matrix halves can operate fully independently. Consequently, the maximum of $t^{\mathrm{up}}(n_{\mathrm{PGCpairs}})$ and $t^{\mathrm{lo}}(n_{\mathrm{PGCpairs}})$ determines the runtime.

To compute an estimate for the computation time, guesses for t^{QR}, t^{QL}, t^{LQ}, and t^{RQ} are necessary. It is reasonable to assume $t^{\mathrm{QR}} \approx t^{\mathrm{QL}}$ and $t^{\mathrm{LQ}} \approx t^{\mathrm{RQ}}$. Measurements confirmed this guess. These measurements have also shown that the ratio between left and right-sided orthogonal transformations can be found between 0.75 and 1.0 for quadratic process grids in process groups, depending on the number of processes per process group. The use of more processes accelerates the right-sided transformations more than the left-sided and hence, the use of more processes drives $t^{\mathrm{QR}}/t^{\mathrm{LQ}}$ closer to one.

Computations using Equations (5.34) and (5.35) have been carried out for the bandwidth-to-matrix-size ratios 1%, 2% and 5% with ten and 20 pairs or PGCs. For these computations, $t^{\mathrm{QR}}/t^{\mathrm{LQ}}$ was fixed to 0.8, 0.9 and 1.0. The estimated runtime for matrix S in twisted shape and matrix G in twisted or in non-twisted shape was computed and compared with the case of non-twisted G and S. All possible setups regarding the choice of P and the distribution of PGCs have been tested.
It was found that the speedup is at least 2 in the inspected cases. The maximum speedup was found to be 2.2. In all cases, as expected, the case of two twisted matrices is slightly faster. Independent of the parameters, the choice $P = N/2$ with equal distribution of the pairs of PGCs to the matrix halves is the best choice. When using higher numbers of PGCs, other setups also reach the same theoretical speed, but no setup outperforms the choice $P = N/2$. The ratio $t^{\mathrm{QR}}/t^{\mathrm{LQ}}$ in the selected range did not have significant impact on the findings for optimal P.

6 Performance Analysis

After deriving theoretical estimates for the runtime in Chapter 5, the implementation is going to be analyzed in this chapter. The single parallelization layers of the implementation are tested in an isolated way and the results discussed regarding the theoretical estimates. Subsequently, weak and strong scaling are analyzed for the twisted Crawford and the twisted SVD algorithm. To demonstrate the potential of the implementation, a comparison of the twisted Crawford algorithm with ELPA is carried out. For the twisted SVD algorithm, a comparison of the runtime with the twisted Crawford algorithm closes the chapter.

6.1 Computational resources

For the test runs presented in the following, SuperMuc-NG [43] was used. SuperMuc-NG consists of 311040 cores and is equipped with a 719TB of main memory. As processors, Intel Skylake Xeon Platinum 8174 are used. Each node has 96GB of memory available for the 48 cores of the node. The nodes themselves are grouped together in islands and the islands are connected by a fast OmniPath network with 100GBit/s. Each islands consists of 792 compute nodes.

In the development phase the COBRA supercomputer [48] of the MPCDF was used. It consists of 3424 compute nodes, each hosting two Intel Xeon Gold 6148 processors with 20 compute cores. This sums up to 136960 compute cores and 529TB of memory. Additionally, 128 Tesla V100-32 and 240 Quadro RTX 5000 GPUs are available at COBRA. The network topology is similar to the one in SuperMuc-NG.

6.2 Twisted Crawford algorithm

In the following, the single parallelization layers of the twisted Crawford algorithm are analyzed. The theoretical assumptions made in the previous chapter are evaluated and thus the scaling behavior of the parallelization layers of the implementation inspected. Afterwards, the overall scaling of the implementation using all its parallelization layers is tested. Finally, a comparison with the ELPA two-step solver is carried out.

The described test cases have bandwidth-to-matrix-size ratios 1%, 2% and 5%. The practical relevance of such thin banded GEP can be seen in [17]. In the context of structural analysis of aerospace composites, strongly anisotropic elliptic partial differential equations have to be solved. One step to solution is to solve a generalized eigenproblem of large size. This part is currently done by using ARPACK [28]. Due to the structure of the matrices, the twisted

Crawford algorithm can also be used to compute such GEPs.

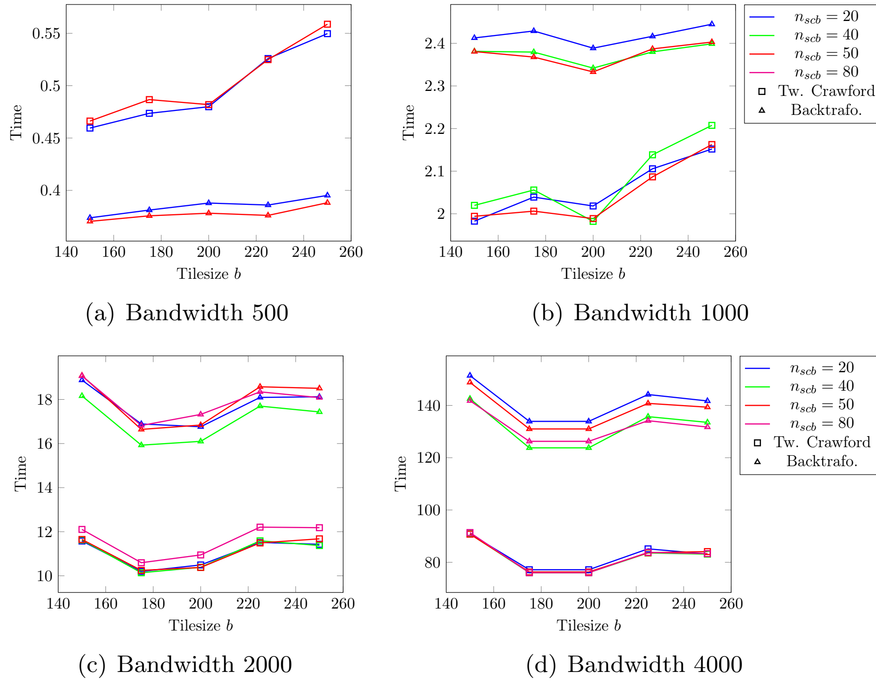
For the scaling tests only the computation time is of interest and therefore, only randomly generated matrices have been used that fulfill the criteria regarding symmetry, bandwidth and positive definiteness.

6.2.1 Block and inter-block parallelization

Every block uses a blockcyclic ScaLAPACK-like data distribution inside which distributes the block of size n_b to the available processes in a 2D blockcyclic way. This blockcyclic distribution uses an own blocksize n_{scb}. The formulas in Chapter 5 do not consider this value since n_b/p_r and n_b/p_c as estimates for the local size of a block are used. Hence, experiments have been carried out to find an optimal value for this parameter. The QR blocking factor b has minor influence in the formulas and thus a good choice for b is solely determined experimentally.
For this test, typical blocksizes ($n_b \in \{500, 1000, 2000, 4000\}$) have been used together with two process group setups (2×2 processes per process group and 4×4 processes per process group). For every process group and blocksize different combinations of b and n_{scb} have been used to run the algorithm on a matrix of size $n = 10n_b$ without upper matrix half ($P = 0$). The backtransformation of eigenvectors was carried out for all eigenvectors.
Figure 6.1 and Figure 6.2 provide the results of these computations. The differences in performance for various choices of n_{scb} are small for most of the runs. $n_{scb} = 50$ produces in many cases the best results or is almost as good as the best result obtained by another choice. Regarding the QR blocking factor b, some influence of n_b can be seen. For small blocks smaller blocking factors like $b = 150$ are favorable whereas for larger n_b, $b = 175$ and $b = 200$ give the best results. A well balanced choice is the use of $b = 175$ together with $n_{scb} = 50$. This combination will be used in most of the later experiments.

To test the scaling behavior on the block and inter-block level, the time for a left-sided and a right-sided update and the time for backtransformation have been measured in an isolated way. The blocksizes 500, 1000, 2000 and 4000 were used in this test. The number of eigenvectors have been varied for every blocksize: The number of eigenvectors to transform back w was computed by $w = k \cdot n_b$ with $k \in \{4, 8, 12, 16\}$. For these numbers of eigenvectors, one step of backtransformation was carried out and the time was measured. The ScaLAPACK blocksize was fixed to $n_{scb} = 50$, the QR blocking factor was set to $b = 175$. The number of PGCs was chosen such that each process group has to work only on one block. The number of processes and the setups of the processes within the process grid of a process group have been varied to obtain the scaling behavior.

Figure 6.3 shows the speedups when using the processes in a $1 \times *$ process grid where all processes are in one row. Hence, the data of a block is only distributed over the columns. The right-sided update scales until many processes, the left stops scaling at around four processes. This is expectable since all processes in a row replicate some of the computations. Hence, the Flops per process are decreasing mildly while adding processes. The right-sided update does not have to compute T which saves this replicated computation. This leads to a better scaling behavior. In general, bigger blocks can exploit a higher number of processes in a row.



(a) Bandwidth 500 (b) Bandwidth 1000

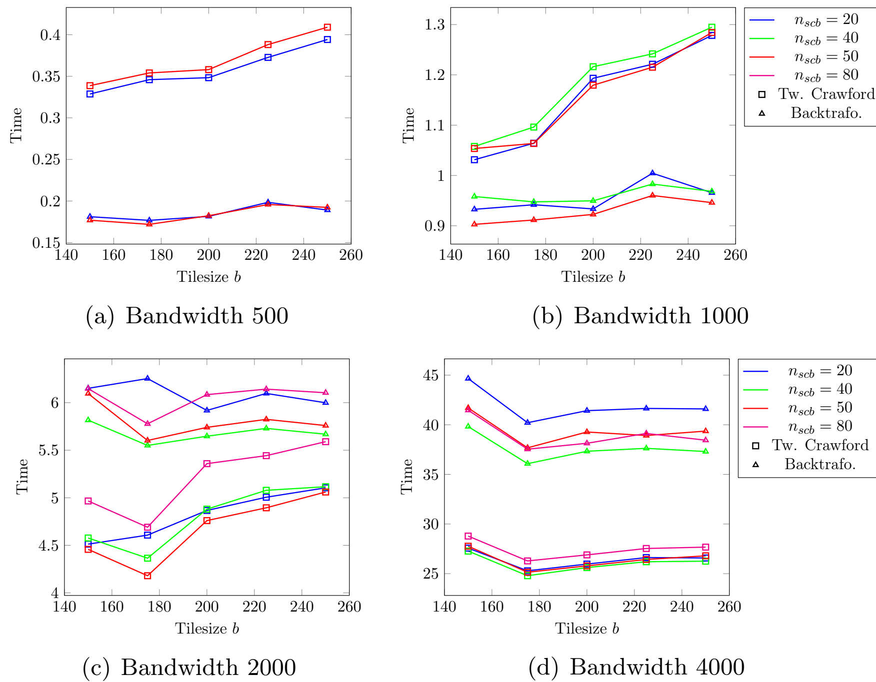
(c) Bandwidth 2000 (d) Bandwidth 4000

Figure 6.1: Time to solution for various bandwidth n_b: Different values of n_{scb} are plotted over the QR tilesize. A block itself is distributed on a 2×2 process grid. The legends apply to all subfigures.

In Figure 6.4 the speedup for the process setup $* \times 1$ is plotted. This process setup distributes solely the rows of a block, every process owns every column of the block. The left and right-sided update show similar scaling behavior for all blocksizes. Now, the computation of T is distributed and therefore, the left-sided update shows better scaling behavior. Again, the bigger the blocks, the higher the attainable speedup by using more processes.

Finally, a combination of both setups is tested. To do so, a quadratic setup of the processes $(* \times *)$ was used which distributed the data of a block in a 2D blockcyclic way. The results are shown in Figure 6.5. As expected by studying the formulas in the previous chapter, the quadratic setup allows to achieve high speedups per block. The scaling behavior can be seen as a combination of the two cases investigated before: The left-sided update will stop scaling after a certain number of processes whereas the right-sided update continues scaling. Bigger blocks shift this point further to the right.

The same study has been carried out for the backtransformation. As mentioned before, the number of eigenvectors was varied to test different setups. Figure 6.6 presents the results for using all processes in a row. Since the eigenvectors per block are distributed to the number of processes per block in this setup, the number of Flops to carry out for each process depends on the number of eigenvectors. Therefore, the curves for different numbers of eigenvectors show different scaling behavior: The higher the amount of eigenvectors, the better the scaling. Compared to the left-sided update, the expensive computation of T has not to be carried out. This is one reason for the backtransformation's scaling to be superior.



(a) Bandwidth 500

(b) Bandwidth 1000

(c) Bandwidth 2000

(d) Bandwidth 4000

Figure 6.2: Time to solution for various bandwidth n_b: Different values of n_{scb} are plotted over the QR tilesize. A block itself is distributed on a 4×4 process grid. The legends apply to all subfigures.

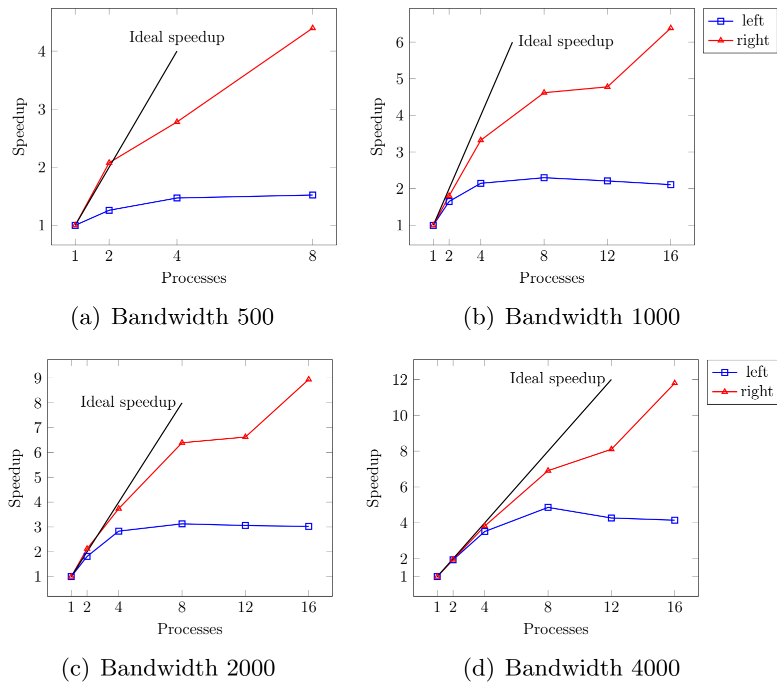
All processes in a column has been evaluated in Figure 6.7. The number of eigenvectors to transform does not matter in this case since every process has its share of every eigenvector. The case of a quadratic process grid was tested and the results are given in Figure 6.8. The quadratic setups show good scaling behavior and have again the impact of the number of eigenvectors.

Overall, the scaling of the left and right-sided update as well as the backtransformation on the block level depends highly on the blocksize. Bigger blocks support speedups for higher number of processes where for smaller blocks the performance stagnates. The formulas in Chapter 5 facilitate a better understanding of this behavior. For exact computations, however, they contain to many inaccuracies, e.g. where communication can be hidden by computation.

6.2.2 Pipelining approach

Equations (5.17) and (5.18) give formulas to compute estimates for the runtime and can hence be used to compute the speedup when increasing the number of PGCs. Such numbers are presented in Table 5.1 for some exemplary cases and can be used to evaluate the implementation.

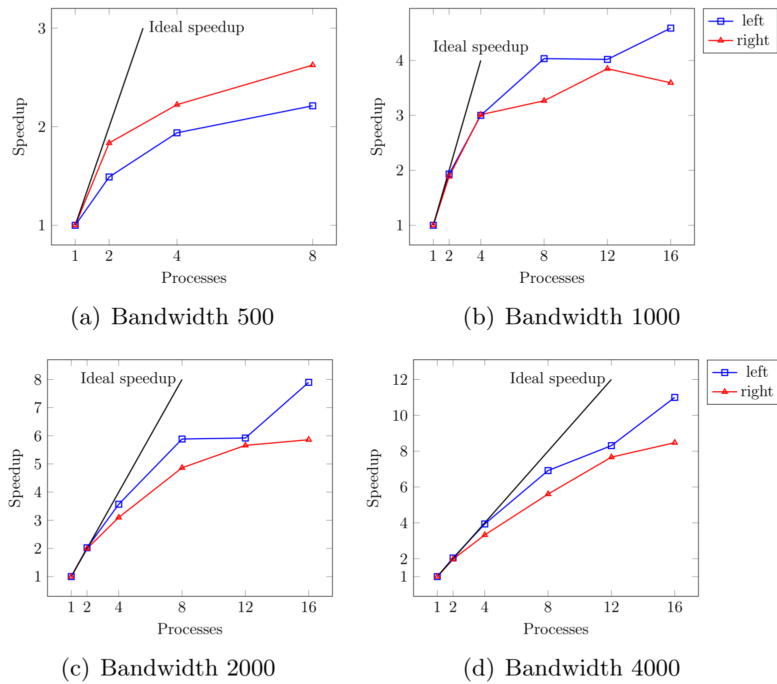
To do so, a numerical experiment was carried out. For bandwidth-to-matrix-size ratios of 1%, 2% and 5% the implementation was run without using twisted factorization. The number of pairs of PGCs was successively increased and the computation time was measured. n_b was



(a) Bandwidth 500 (b) Bandwidth 1000

(c) Bandwidth 2000 (d) Bandwidth 4000

Figure 6.3: Speedups of the left (ParallelBlockedQR, Algorithm 37) and right-sided steps (ParallelBlockedRightSidedApplQ, Algorithm 40) for various numbers of processes in a process group. The process grid is arranged as $1 \times *$, where $*$ is the number of processes and all processes are arranged in a row. The legends apply to all subfigures.



(a) Bandwidth 500 (b) Bandwidth 1000

(c) Bandwidth 2000 (d) Bandwidth 4000

Figure 6.4: Speedups of the left (ParallelBlockedQR, Algorithm 37) and right-sided steps (ParallelBlockedRightSidedApplQ, Algorithm 40) for various numbers of processes in a process group. The process grid is arranged as $* \times 1$, where $*$ is the number of processes and all processes are arranged in a column. The legends apply to all subfigures.

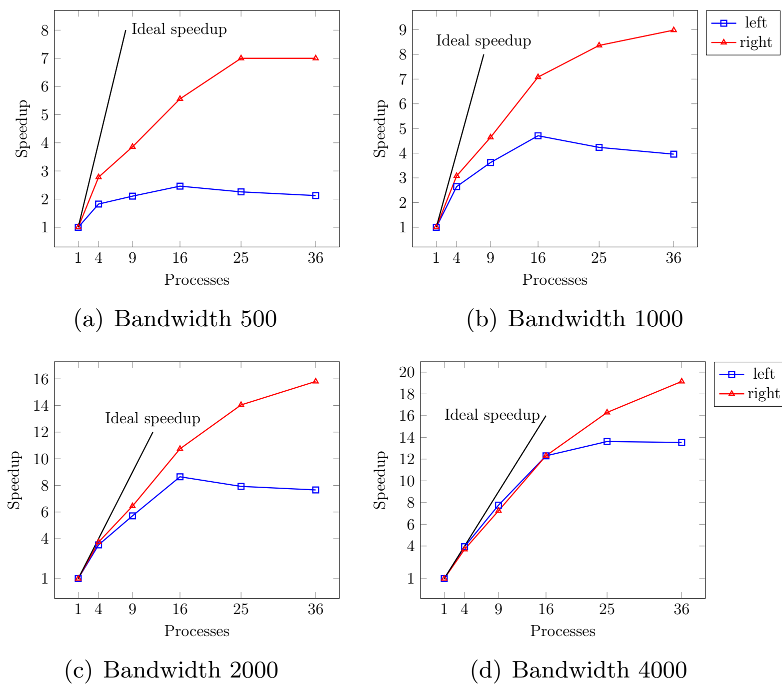
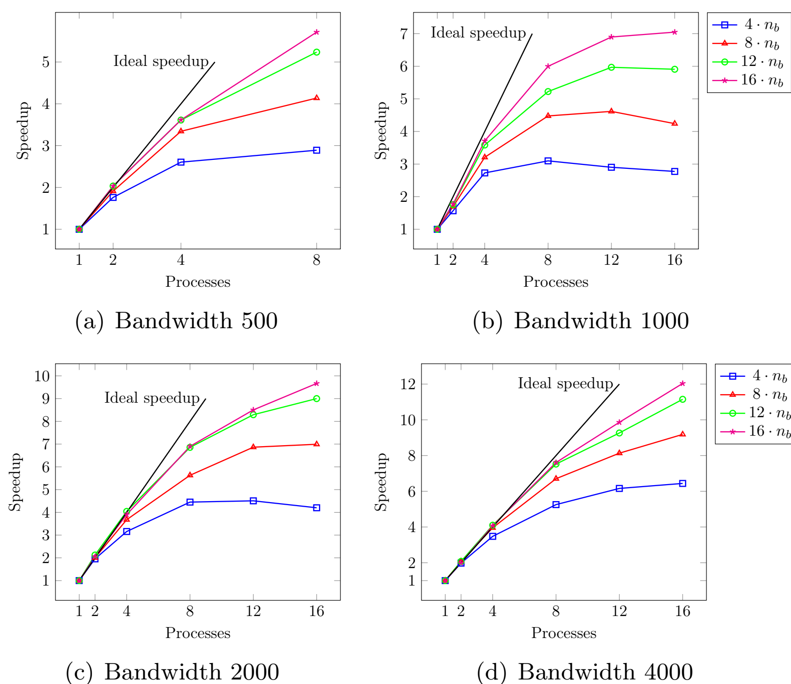


Figure 6.5: Speedups of the left (ParallelBlockedQR, Algorithm 37) and right-sided steps (ParallelBlockedRightSidedApplQ, Algorithm 40) for various numbers of processes in a process group. The process grid is arranged as $* \times *$ in a quadratic way, where $*$ is the number of processes in every direction of the process grid. The legends apply to all subfigures.



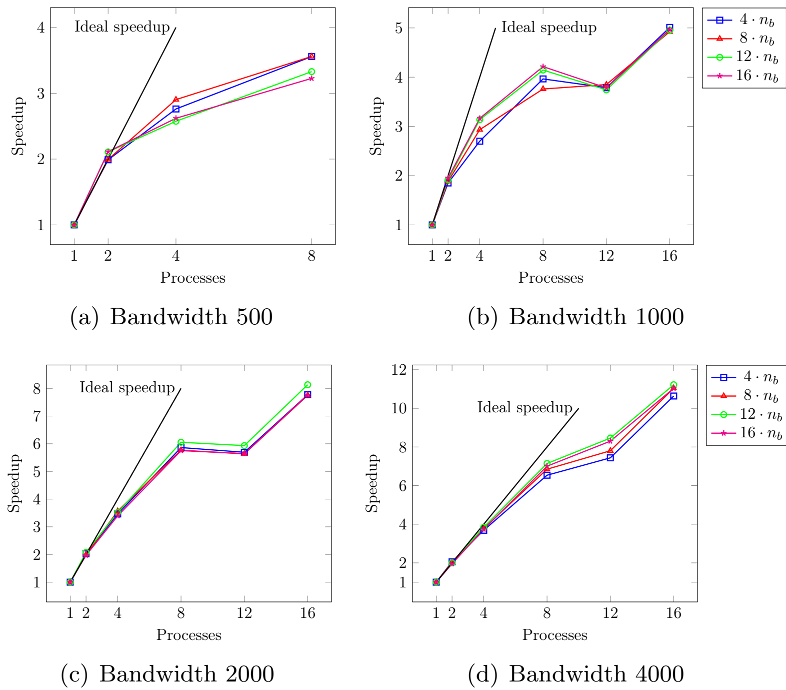
(a) Bandwidth 500

(b) Bandwidth 1000

(c) Bandwidth 2000

(d) Bandwidth 4000

Figure 6.6: Speedups of the backtransformation (ParallelCWYtransposedLeft, Algorithm 42) for various numbers of processes in a process group. The process grid is arranged as $1 \times *$, where $*$ is the number of processes and all processes are arranged in a row. The different lines give the results for different numbers of eigenvectors w. The legends apply to all subfigures.



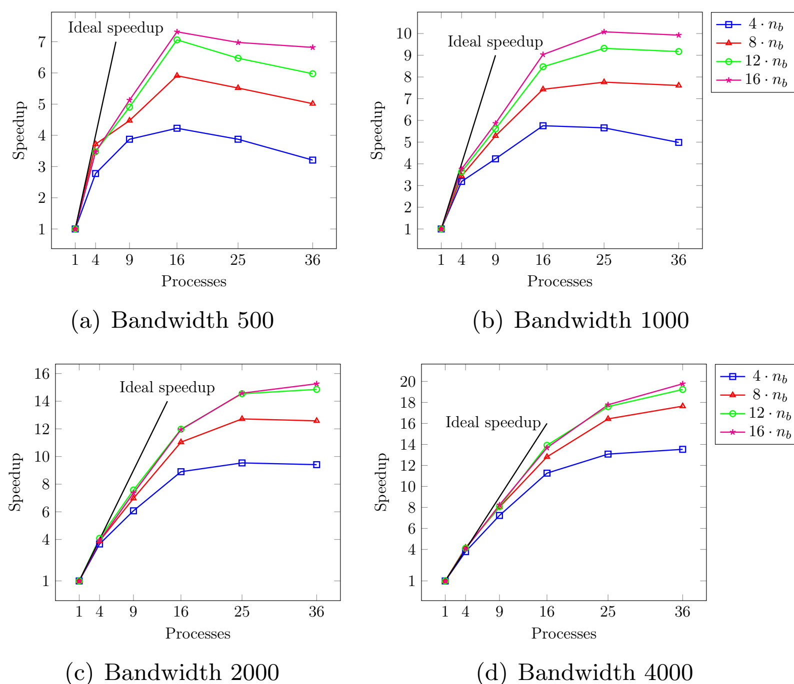
(a) Bandwidth 500

(b) Bandwidth 1000

(c) Bandwidth 2000

(d) Bandwidth 4000

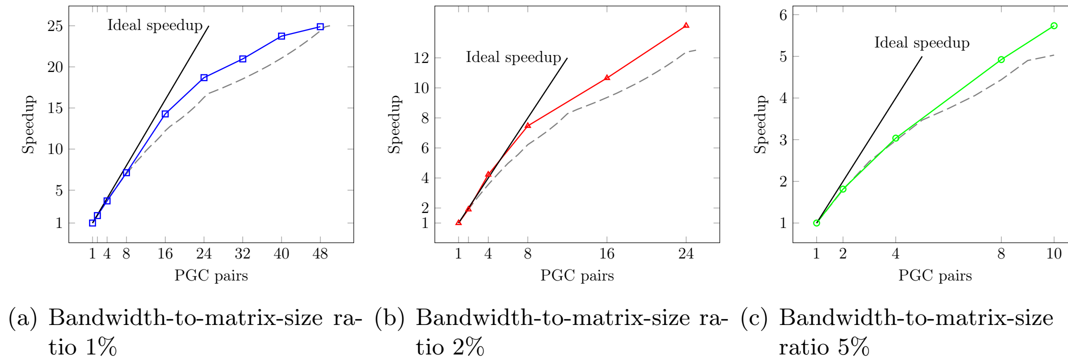
Figure 6.7: Speedups of the backtransformation (ParallelCWYtransposedLeft, Algorithm 42) for various numbers of processes in a process group. The process grid is arranged as $* \times 1$, where $*$ is the number of processes and all processes are arranged in a column. The different lines give the results for different numbers of eigenvectors w. The legends apply to all subfigures.



(a) Bandwidth 500 (b) Bandwidth 1000

(c) Bandwidth 2000 (d) Bandwidth 4000

Figure 6.8: Speedups of the backtransformation (ParallelCWYtransposedLeft, Algorithm 42) for various numbers of processes in a process group. The process grid is arranged as $* \times *$, where $*$ is the number of processes in every direction of the process grid. The different lines give the results for different numbers of eigenvectors w. The legends apply to all subfigures.



(a) Bandwidth-to-matrix-size ratio 1%
(b) Bandwidth-to-matrix-size ratio 2%
(c) Bandwidth-to-matrix-size ratio 5%

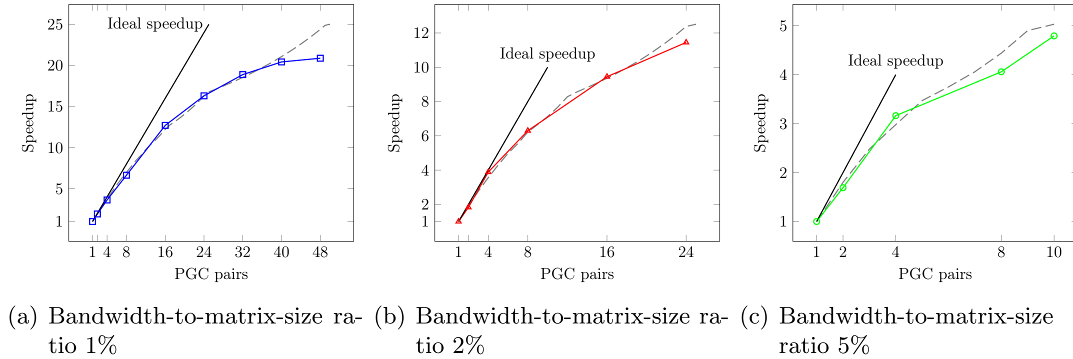
Figure 6.9: Speedup for different numbers of PGC pairs for various bandwidth-to-matrix-size ratio. The block size is fixed to 1000, thus the matrix size of the different cases is 100000 for 1%, 50000 for 2% and 20000 for a bandwidth-to-matrix-size ratio of 5%. The theoretical speedup by Equation (5.17) is plotted as gray dashed line without marks for comparison.

fixed to 1000 and thus the matrix sizes to 100000, 50000 and 20000. A tilesize of $b = 150$ was used together with a ScaLAPACK blocksize of $n_{scb} = 50$. Every process group consisted of one process to eliminate the influence of communication within the process groups. To omit effects of saturated memory bandwidth or communication bandwidth, every process group was assigned to an own node.
Figure 6.9 shows the speedup results of the different runs to the regarding base run with one pair of PGCs. The plots show good agreement in the speedup values as well as in the inclination. Surprisingly, the computational results outperform the theoretical estimations. A possible reason might be that, when using only a few PGC pairs, the computational load per process is very high and the clockspeed of the CPU is reduced. This leads to a slower baseline and that causes high speedups for less CPU intensive PGC pair numbers. However, this has not been investigated in this work.

So far, only the transformation of the banded GEP to the banded SEP has been inspected. For the backtransformation the same analysis can be carried out. Only the case of backtransformation for 12.5% of the eigenvectors is analyzed exemplarily. The computational setup was chosen as described for the transformation of the GEP to the SEP.
The obtained speedups are shown in Figure 6.10. Again, a good agreement between the theoretical values and the computational speedups is found.
Hence, Equations (5.17) and (5.18) are as well good estimators for the speedup when adding pairs of PGCs to the backtransformation step. Since the formulas work for the twisted Crawford part and the backtransformation part, they can serve as general estimator for the speedup when increasing the number of PGC pairs.

6.2.3 Parallel execution of the upper and a lower matrix half

Theoretical assumptions for speedups and the optimal choices for P are derived in Section 5.1.3. These theoretical numbers are now evaluated in numerical experiments.

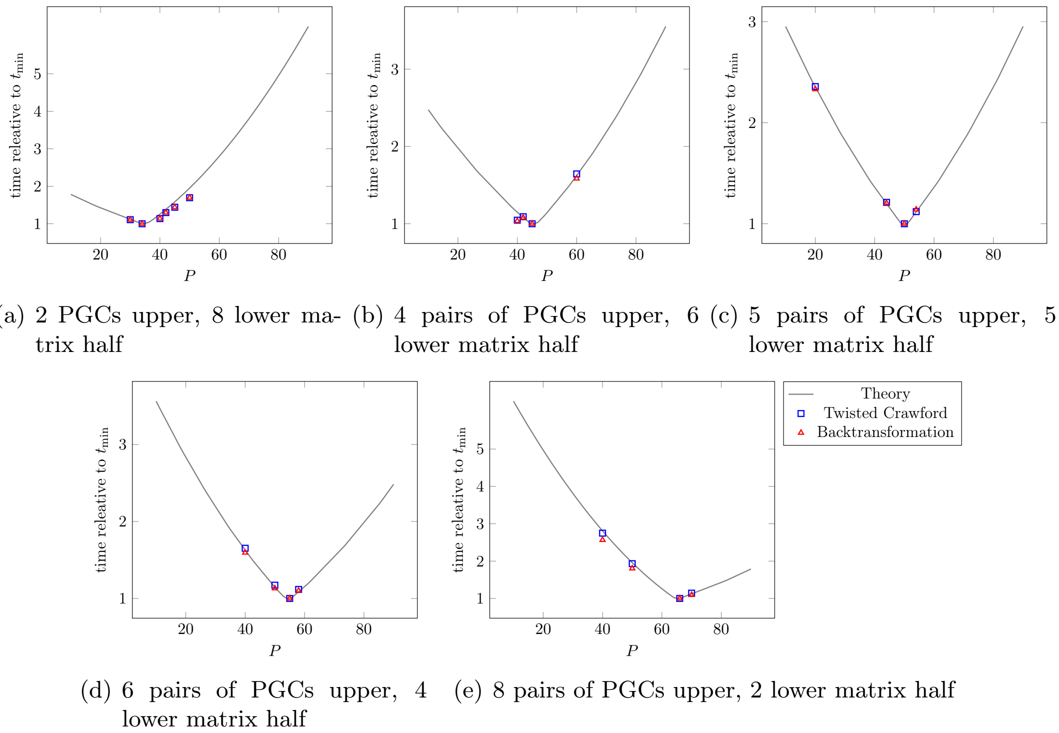


(a) Bandwidth-to-matrix-size ratio 1% (b) Bandwidth-to-matrix-size ratio 2% (c) Bandwidth-to-matrix-size ratio 5%

Figure 6.10: Speedup for different numbers of PGC pairs for various bandwidth-to-matrix-size ratio during backtransformation. The block size is fixed to 1000, thus the matrix size of the different cases is 100000 for 1%, 50000 for 2% and 20000 for a bandwidth-to-matrix-size ratio of 5%. In all cases, 12.5% of the eigenvectors are computed. The theoretical speedup by Equation (5.17) is plotted as gray dashed line without marks for comparison.

Figure 5.1 plots the number of time steps in parallel (relative to the duration of one QR decomposition and its left and right-sided application) over the choice of P for various PGC distributions. The curves obtained for each PGC distribution are evaluated at the beginning. To do so, a bandwidth-to-matrix-size ratio of 1% is chosen together with $n_b = 1000$, $ts = 150$ and $n_{scb} = 50$. In total, the number of pairs of PGCs is kept constant at 10. For the experiments, the PGC distributions 2/8, 4/6, 5/5, 6/4 and 8/2 have been selected and the computation time was measured for different P.
Figure 6.11 shows the regarding theoretical curves in gray, in color the measurements for the twisted Crawford algorithm (blue squares) and the backtransformation measurements (red triangles). The measurements agree well with the curves and hence it can be assumed that the prediction of the best P value for a given PGC setup is sufficiently precise.

After validating the prediction of the best P value for single PGC distributions, the speedup of the different PGC distributions against a pure lower matrix half run is plotted. This allows to evaluate the prediction of the attainable speedup for every PGC distribution as it is shown in Figure 5.1 by the gray curve. Figure 6.12 plots the measurements against the theoretical values. The measurements follow the curve but over and undershoot the prediction slightly. The best speedup values for the twisted Crawford algorithm were achieved for four pairs of PGCs in the upper and six pairs in the lower matrix half ($P = 45$) and five pairs of PGCs in each half ($P = 50$). For the backtransformation the best speedup was found for five pairs of PGCs in each matrix half. The attainable speedup was predicted to be 1.98 and was measured at 1.96 for the twisted Crawford algorithm and 2.03 for the backtransformation. Thus, the theoretical predictions have shown their capabilities and the supposed best setup with $P = N/2$ and equal splitting of the PGCs has shown to be the best choice. Due to the attained speedup, the use of a twisted factorization is highly recommended.



(a) 2 PGCs upper, 8 lower matrix half (b) 4 pairs of PGCs upper, 6 lower matrix half (c) 5 pairs of PGCs upper, 5 lower matrix half

(d) 6 pairs of PGCs upper, 4 lower matrix half (e) 8 pairs of PGCs upper, 2 lower matrix half

Figure 6.11: Effect of the choice of P on the computation time for 10 pairs of PGCs in various distributions to the upper and lower matrix half for a bandwidth-to-matrix-size ratio of 1%. The legend applies to all subfigures.

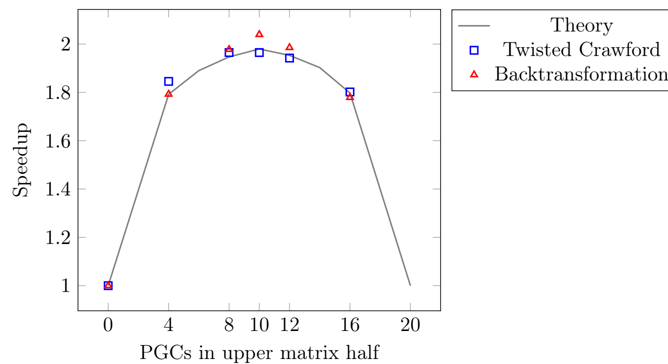


Figure 6.12: Speedup of using a twisted factorization instead of a standard Cholesky factorization for B at a bandwidth-to-matrix-size ratio of 1% for different distributions of PGCs to the upper and lower matrix half. For every PGC combination, the best P setup was chosen.

6.2.4 Scaling of the overall implementation

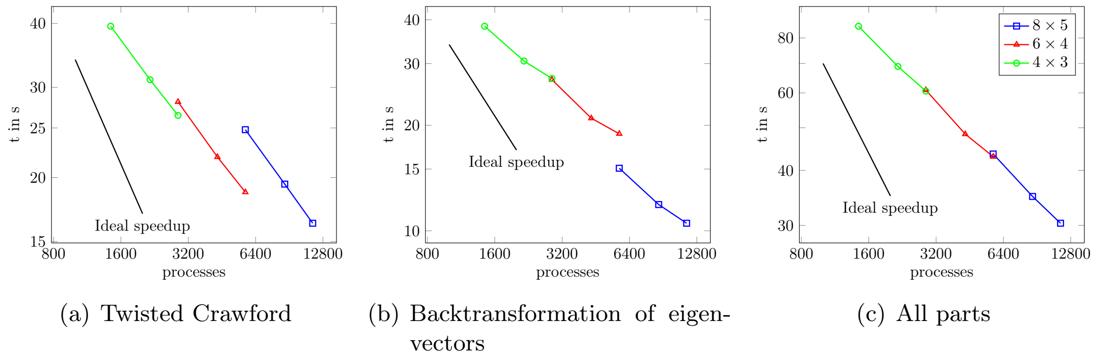
After evaluating the single parallelization layers, the pieces can be put together and the overall scaling is analyzed.

Strong scaling describes the speedup by increasing the number of processes for a fixed problem size. It is connected to Amdahl's law [2], which gives a hint on the serial part of a code and describes the limits of scaling.
Tests for strong scaling have been carried out for bandwidth-to-matrix-size ratios of 1%, 2% and 5%. As blocksize $n_b = 2000$ was used for 1% and 2% and $n_b = 5000$ for 5% bandwidth-to-matrix-size ratio. The former showcase the behavior for a small blocksize and long pipelines, the latter for large blocks and a short pipeline. The QR blocking factor was fixed to 150, n_{scb} to 50 and the number of eigenvectors to transform back, was selected to be 12500 for all runs.

Figure 6.13 shows the scaling results for a bandwidth-to-matrix-size ratios of 1%, Figure 6.14 for a bandwidth-to-matrix-size ratio of 2% and Figure 6.15 for a bandwidth-to-matrix-size ratio of 5%.
For a bandwidth-to-matrix-size ratios of 1%, both parts of the algorithm scale very well and good speedups can be attained. As expected after analyzing the scaling on the block and inter-block level, an increase of the number of processes per process group can lead to higher computation time in comparison with the same number of processes but less processes per process group when running the twisted Crawford part. The effect can especially be seen for the 8×6 process group setup in comparison with the setup 6×4. The latter uses a higher number of PGCs and hence is faster. In the backtransformation, this effect cannot be seen. The plot of all parts includes the factorization of B and the overhead from transforming matrices from and to ScaLAPACK format. Both scale by the number of processes per process group and are more or less constant when adding additional PGCs.
The same behavior can be observed for a bandwidth-to-matrix-size ratio of 2%. In this case, saturation effects using more and more PGCs can be seen. The curves for the single process group setups flatten for the last measurement and are outperformed by the next larger process group setup.
A bandwidth-to-matrix-size ratio of 5% draws a little different picture. Due to the bigger n_b, block scaling becomes more efficient and the plot for the twisted Crawford part shows almost a straight line. The backtransformation step has already shown in the other bandwidth-to-matrix-size ratios that shifting processes from process groups to PGCs does not improve the runtime. In this case, the situation becomes more extreme and the opposite is the case.

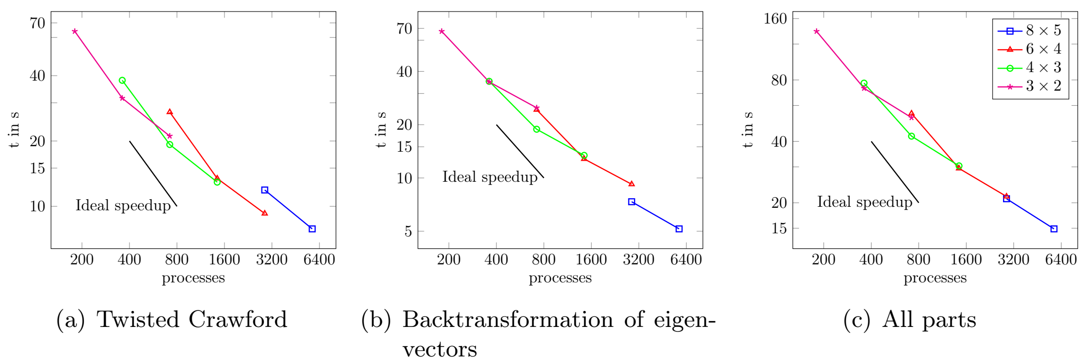
After inspecting the strong scaling behavior, the selection of the best process setup remains a challenging task. In general, it is recommendable to first increase the PGC number. By having a look on Figures 6.9 and 6.10 it can be seen when and how the attainable speedup decreases and in combination with Figure 6.5 and Figure 6.8, the decision when to increase the processes per process group can be made.

Finally, the weak scaling of the implementation shall be analyzed. Gustafson [31] pointed out that problem sizes will generally scale with the available computational resources. Weak scaling analyzes how implementations behave when keeping the computational load per process constant while increasing the problem size. Obviously, to keep the load per process



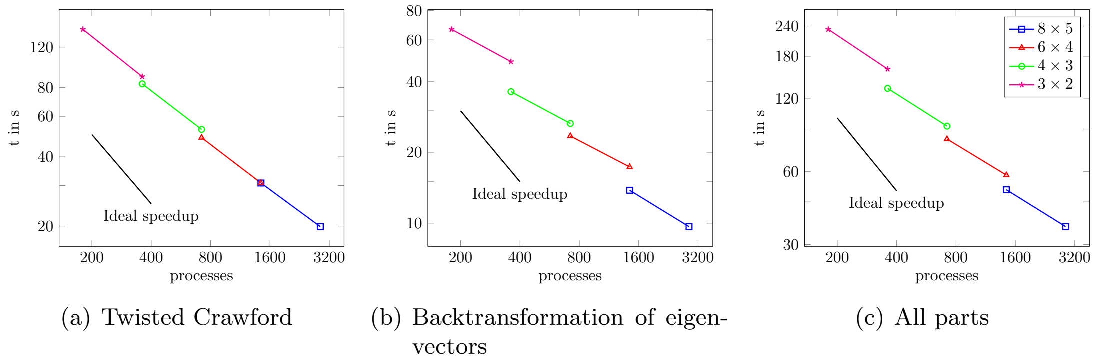
(a) Twisted Crawford (b) Backtransformation of eigenvectors (c) All parts

Figure 6.13: Scaling of twisted Crawford algorithm and backtransformation employing all parallelization levels for a matrix of size 200000, The blocksize is 2000 (bandwidth-to-matrix-size ratio 1%), the twist position at 100000. For backtransformation, 12500 eigenvectors have been transformed. The right plot shows the overall runtime, including factorization and matrix redistributions. The different lines show different setups of process groups. The legend applies to all subfigures.



(a) Twisted Crawford (b) Backtransformation of eigenvectors (c) All parts

Figure 6.14: Scaling of twisted Crawford algorithm and backtransformation employing all parallelization levels for a matrix of size 100000, The blocksize is 2000 (bandwidth-to-matrix-size ratio 2%), the twist position at 50000. For backtransformation, 12500 eigenvectors have been transformed. The right plot shows the overall runtime, including factorization and matrix redistributions. The different lines show different setups of process groups. The legend applies to all subfigures.



(a) Twisted Crawford (b) Backtransformation of eigenvectors (c) All parts

Figure 6.15: Scaling of twisted Crawford algorithm and backtransformation employing all parallelization levels for a matrix of size 100000, The blocksize is 5000 (bandwidth-to-matrix-size ratio 5%), the twist position at 50000. For backtransformation, 12500 eigenvectors have been transformed. The right plot shows the overall runtime, including factorization and matrix redistributions. The different lines show different setups of process groups. The legend applies to all subfigures.

constant, the number of processes has to be increased as the problem size grows.
Increasing the problem size can be interpreted as increasing the blocksize n_b while increasing the number of processes per process group. This goes along with keeping the number of blocks, N, constant, thus the bandwidth-to-matrix-size ratio stays constant while increasing the matrix size. Another interpretation could be to increase also the number of blocks and hence decrease the bandwidth-to-matrix-size ratio successively. Which interpretation fits better depends on the origin of the eigenvalue problem. Here, the former is chosen and n_b is increased while N stays constant.

For this test, N is kept constant at 20, $b = 150$ and $n_{scb} = 50$. The number of eigenvectors w is increased along with the matrix size, 10% of the eigenvectors are transformed back. The number of PGCs is kept constant at 8. This allows to only inspect the weak scaling of the block parallelization. The reason for this approach is that, as the strong scaling has shown, first, the number of PGCs is increased successively and after saturation starts to come into play, the number of processes per process group is increased. Hence, the starting point will usually be to have many PGCs but small process groups. Further increasing the number of PGCs will then not pay off and the processes are better invested in process groups. The growing blocksize also supports better scaling in the process groups.

The results of the experiment are shown in Figure 6.16. The overall number of processes is raised from 24 to 2400. This translates to an increase in the size of the process groups from 1 process to 100 processes. Since the number of PGCs is kept constant, the overall number of processes raises by this factor. While increasing n_b from 1000 to 10000, thus by a factor of 100, the runtime increases by a factor of 19.6 for the twisted Crawford part and by 20.8 for the backtransformation.

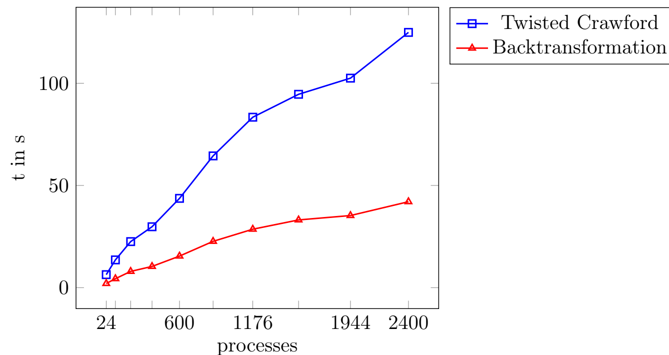


Figure 6.16: Weak scaling of the twisted Crawford algorithm and the backtransformation of eigenvectors.

6.2.5 Comparison to ELPA

In the following, a comparison study between the twisted Crawford algorithm and the ELPA two-step solver [37] (release 2018.11.001) is carried out. Matrices of size 10000, 50000 and 100000 are used as test matrices. For ELPA, the bandwidth of A and B does not matter since it assumes full matrices. In the twisted Crawford algorithm, bandwidth-to-matrix-size ratios of 1%, 2% and 5% are used. The intermediate bandwidth of ELPA is set to 32. The twisted Crawford algorithm uses $b = 150$ and $n_{scb} = 50$. The percentage of eigenvectors to transform back was chosen to be 12.5%. Hence, 1250 for matrix size 10000, 6250 for matrix size 50000 and 12500 for matrix size 100000.

Table 6.1 shows the results for matrix size 10000. In this case, the bandwidths of the matrices are 100, 200 and 500. Due to the small bandwidth, no bandreduction step was used after running the twisted Crawford algorithm. Thus, the tridiagonalization step of ELPA was run with bandwidths of 100, 200 and 500. Optimal values are usually in the range of 32 to 64 [4]. In ELPA, a novel matrix multiplication algorithm by Manin [45] following Cannon's ideas [18] can be used if slightly increasing the number of processes to a setup where in the process layout p_c is a multiple of p_r. Using Manin's Cannon multiplication, ELPA computes significantly faster.
Computing the eigenspektrum for a bandwidth-to-matrix-size ratio of 1%, the speedup of the twisted Crawford approach to ELPA is roughly 3, for a bandwidth-to-matrix-size ratio of 2%, the speedup is about 2. However, for a bandwidth-to-matrix-size ratio of 5%, ELPA is faster by a factor of more than 2. The reason is, that the tridiagonalization step becomes extremely slow for a bandwidth of 500. Hence, a bandreduction step might help to reduce the computation time in this case.

Such a bandreduction algorithm is currently under development by Manin. He follows the approach in [14]. Other ideas can be found in [9], where avoiding communication is a key fact of the algorithm and [52], where blocked Givens rotations are employed. Manin's implementation is still under development. Hence, the numbers are preliminary and should improve as the algorithm matures.
The right most column in Table 6.1 shows timings for combining the twisted Crawford algorithm with Manin's bandreduction algorithm. The band is reduced from 500 to 100 by the

	ELPA		Twisted Crawford			
	Cannon	no Cannon	$n_b/n = 1\%$	$n_b/n = 2\%$	$n_b/n = 5\%$	
Processes	121	120	120	120	120	120
Factorization	0.36	0.39	0.018	0.039	0.13	0.13
Transformation to banded form	1.83	2.61	0.15	0.20	0.37	0.37
Further bandreduction	0	0	0	0	0	1.13
Compute EV, EVec for banded matrix	0.47	0.45	0.62	1.12	6.62	0.62
Backtransformation from banded form	0.26	0.26	0.12	0.14	0.15	0.15
Further backtransformation	0	0	0	0	0	0.28
Matrix redistributions	0	0	0.019	0.028	0.037	0.061
Sum	2.91	3.72	0.93	1.53	7.31	2.74
Speedup to Cannon	1.00	0.78	3.13	1.90	0.40	1.07

Table 6.1: Comparison of timings obtained by ELPA and the twisted Crawford algorithm for a matrix of size 10000 on 120 processes without further bandreduction. 1250 eigenvectors have been transformed back. For a bandwidth-to-matrix-size ratio of 5%, additionally, the variant using a bandreduction step is listed.

bandreduction and afterwards, the tridiagonalization of ELPA is run. After ELPA delivered the eigenvalues and eigenvectors, the latter are transformed back by the bandreduction algorithm and finally, transformed back by the twisted Crawford algorithm.
Between the twisted Crawford algorithm and the bandreduction step the matrix is transformed to ScaLAPACK format. This intermediate format can be omitted when immediately transforming the twisted Crawford format to the bandreduction format. However, since the bandreduction algorithm is still under development, this implementation has not been conducted yet.
By transforming the banded matrix to a narrower banded matrix, the computation of the eigenvalues and vectors for the banded matrix is accelerated by a factor of 10. The bandreduction is rather expensive with 1.13 seconds and and the backtransformation takes 0.28 seconds. The overall runtime sums up to 2.74 and is hence slightly faster than the ELPA runs with Cannon's algorithm. The importance of the bandreduction step can be observed when comparing the numbers with and without bandreduction step. The runtime decreased by a factor of 2.67.

Table 6.2 gives the results for matrix size 50000 when using 720 processes or 729 processes running ELPA with Cannon's algorithm. As already seen for matrix size 10000, the time to solution in the twisted Crawford algorithm depends on the bandwidth-to-matrix-size ratio. The smaller the ratio, the faster the computation. Even if the block parallelization gets worse by having smaller blocks, the number of blocks grows and hence, the speedup by pipelining compensates.

	ELPA		Twisted Crawford		
	Cannon	no Cannon	$n_b/n = 1\%$	$n_b/n = 2\%$	$n_b/n = 5\%$
Processes	729	720	720	720	720
Factorization	6.52	7.14	0.40	0.58	1.52
Transformation to banded form	39.49	50.77	2.14	3.46	7.67
Further bandreduction	0	0	7.64	8.59	16.49
Compute EV, EVec for banded matrix	5.55	4.92	5.87	5.87	5.87
Backtransformation from banded form	6.73	7.16	3.33	2.84	3.95
Further backtransformation	0	0	8.89	8.49	7.41
Matrix redistributions	0	0	0.49	0.29	0.49
Sum	58.30	70.01	28.76	30.11	43.41
Speedup to Cannon	1.00	0.83	2.03	1.94	1.34

Table 6.2: Comparison of timings obtained by ELPA and the twisted Crawford algorithm for a matrix of size 50000 on 720 processes. 6250 eigenvectors have been transformed back.

The most expensive part in the ELPA runs is the transformation from the GEP to the banded SEP. Here, all three twisted Crawford runs outperform ELPA by far. When considering the bandreduction step this statement still holds. The backtransformation from banded form is also approximately twice as fast in the twisted Crawford algorithm. However, when considering the band-to-band backtransformation, things change. This part is very costly and when considering it, the backtransformation takes twice as long as the ELPA backtransformation. The transformations between matrix formats do not have a huge impact on the runtime. The overall runtime of the twisted Crawford algorithm is smaller in all three cases. For the case of 1% bandwidth-to-matrix-size ratio, it is half of the runtime of ELPA with Cannon's algorithm. Even if the bandreduction is still under developement and shorter runtimes are expectable for the final implementation, this test-series has shown that the twisted Crawford algorithm together with a bandreduction algorithm is superior for medium-sized matrices with thin bands.

Similar results can be seen in Table 6.3 for matrix size 100000, which was computed by 1440 processes using the twisted Crawford algorithm or standard ELPA, and by 1444 processes using ELPA's Cannon variant. The runtime of the twisted Crawford runs is now fully dominated by the bandreduction step and the regarding backtransformation. Further improvements on these substeps will hence accelerate the overall runtime drastically.
The obtained speedups 1.88 for a bandwidth-to-matrix-size ratio of 1%, 1.81 for 2% and 1.17 for 5% show also for larger matrices the advantage of the twisted Crawford approach.

To conclude, for thin banded matrices of size 10000, 50000 and 100000, the twisted Crawford algorithm outperforms classical two-step solvers like the one in ELPA. The bottleneck

	ELPA		Twisted Crawford		
	Cannon	no Cannon	$n_b/n = 1\%$	$n_b/n = 2\%$	$n_b/n = 5\%$
Processes	1444	1440	1440	1440	1440
Factorization	23.26	27.31	1.11	2.11	5.08
Transformation to banded form	132.47	173.82	7.06	13.44	30.78
Further bandreduction	0	0	29.74	33.55	72.68
Compute EV, EVec for banded matrix	18.11	17.73	17.64	17.64	17.64
Backtransformation from banded form	19.18	22.86	11.78	12.81	13.80
Further backtransformation	0	0	34.14	25.62	23.88
Matrix redistributions	0	0	1.21	1.40	1.22
Sum	193.04	241.72	102.68	106.56	165.09
Speedup to Cannon	1.00	0.80	1.88	1.81	1.17

Table 6.3: Comparison of timings obtained by ELPA and the twisted Crawford algorithm for a matrix of size 100000 on 1440 processes. 12500 eigenvectors have been transformed back.

is the bandreduction algorithm and the regarding backtransformation. Both are still under development and further improvements will boost the advantage of the twisted Crawford algorithm.
For all cases of the twisted Crawford, the factorization step is faster as the one in ELPA. This is mainly because of exploiting the banded structure (of different bandwidth; hence, the differences within the twisted Crawford numbers). Another reason is, that the factorization algorithm is split in two matrix halves that are processed in parallel.

The biggest matrix addressed by ELPA (which has been reported) so far had a matrix size of 1 million [41]. Similar numbers can be found for EigenExa [33]. For both solvers, sizes like this are challenging and require lots of resources. The ELPA run was carried out using 200000 cores of the supercomputer Theta [38] at Argonne National Laboratory and took around 10000 seconds to compute the eigenvalues of the SEP.
Table 6.4 gives results using the twisted Crawford algorithm together with Manin's bandreduction, ELPA's tridiagonalization procedure and ELPA's eigenvalue computation for tridiagonal matrices. The overall runtime for 15360 cores with 3282.51 seconds and 2777.42 seconds for 24000 cores is by far smaller than what is reported for ELPA, even when using by far less resources. Clearly, comparing the Skylake nodes of SuperMUC-NG to the Knight's Landing chips in Theta does not allow a fair comparison. However, the underlying computing power by the 200000 cores of Theta is by far bigger than 15360 or 24000 cores of SuperMUC-NG.

This comparison allows to conclude, that the twisted Crawford algorithm allows to address GEPs of large matrices with much less resources. Additionally to saving computational

Processes	15360	24000
Process setup	8×8, 80 PGC	10×10, 80 PGC
Factorization	117.42	85.80
Transformation to banded form	616.31	456.38
Further bandreduction	1756.83	1623.75
Tridiagonalization and solve	791.95	611.49
Sum	3282.51	2777.42

Table 6.4: Computing the eigenvalues of a GEP with matrix size 1 million and bandwidth 10000.

resources and thus energy, the time to solution reduces drastically. Due to the lower requirements on hardware resources and the shorter computation time, problem sizes which have been out of reach for direct solvers can now be tackled using the twisted Crawford algorithm.

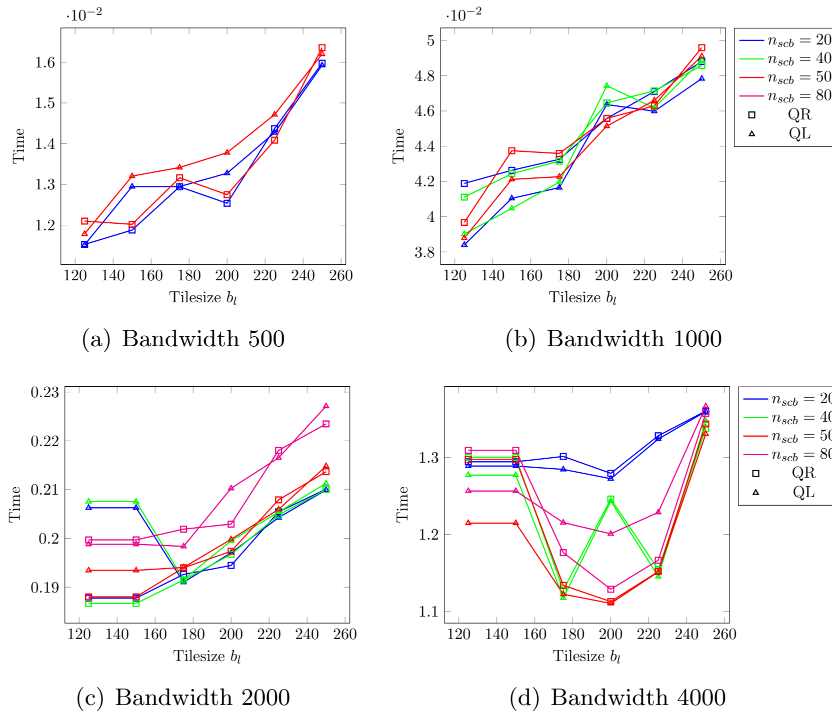
6.3 Crawford SVD algorithm

In the previous chapter, theoretical assumptions have been made on the implementation of the Crawford SVD algorithm. In the following, these assumptions are going to be evaluated experimentally. First, the single parallelization layers will be analyzed and afterwards the scaling of the overall algorithm is tested for different scenarios. The backtransformation of the singular vectors has not been implemented and is hence not going to be evaluated. However, the left singular vectors can be obtained by using a simplified version of the implementation of the eigenvector backtransformation without applying the inversion steps.

Since only the computation time is of interest for scaling tests, it is sufficient to use random matrices. Therefore, symmetric positive definite random matrices with the desired bandwidth have been created and by factorization (Cholesky or twisted) transformed to banded lower triangular matrices or banded twisted matrices.

6.3.1 Block and inter-block parallelization

The analysis of the scaling behavior of the block and inter-block parallelization targets the efficiency of the implementations of the QR, QL, RQ and LQ steps. Every block contains a 2D process grid to which the data is assigned to in a 2D blockcyclic way. A theoretical analysis of the algorithms regarding runtime has been conducted in Chapter 5. To assign the data to the process grid in a blockcyclic way, a certain blocksize n_{scb} has to be defined. This value is determined experimentally since it does not appear in the theoretical formulas. Additionally, the blocking factors for the left (b_l) and right (b_r) sided transformations are obtained experimentally.



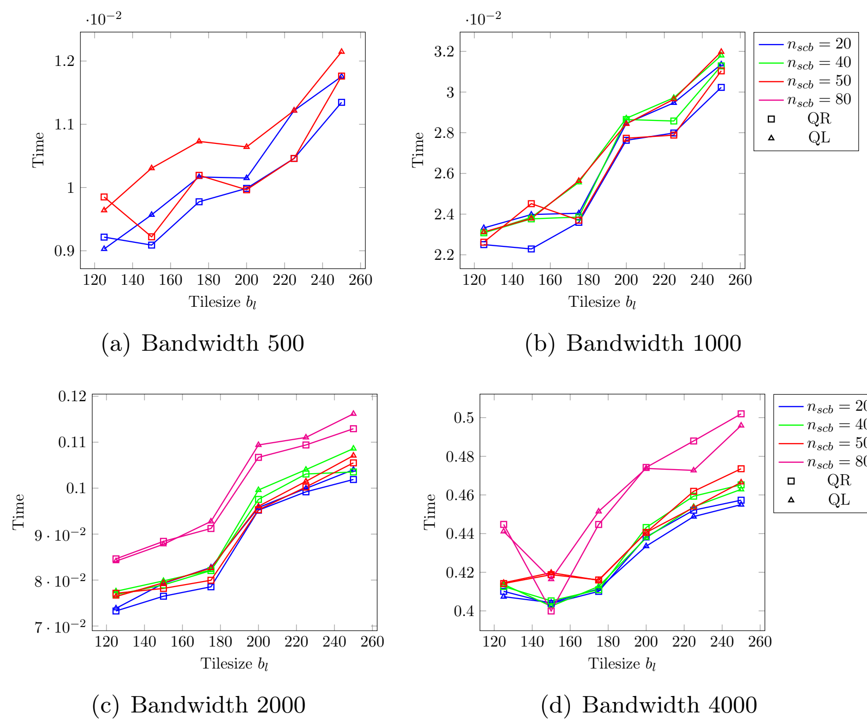
(a) Bandwidth 500 (b) Bandwidth 1000

(c) Bandwidth 2000 (d) Bandwidth 4000

Figure 6.17: Time to solution for various bandwidth n_b: Different values of n_{scb} are plotted over the QR and QL tilesize. A block itself is distributed on a 2×2 process grid. The legends apply to all subfigures.

Two process group setups (2×2 and 4×4 processes) have been used to compute the runtime of QR, QL, RQ and LQ for different numbers of n_{scb}, b_l and b_r. The runtimes have been measured in an isolated way where only the regarding transformation was carried out. For the test, blocksizes of $n_b \in \{500, 1000, 2000, 4000\}$ have been used.
Figures 6.17 and 6.18 show the results for the left-sided transformations of the two process group setups mentioned. Smaller blocksizes tend to give better results for smaller tilesizes. The bigger blocksizes provide the best results for intermediate numbers of b_l. Regarding n_{scb}, small values seem to be beneficial but for $n_b = 4000$, the results are by far worse than with the other n_{scb} values.
Figures 6.19 and 6.20 provide the view on the right-sided transformations. Besides a few cases, also for RQ and LQ $n_{scb} = 20$ is the best choice. However, for $n_b = 4000$ and a 2×2 process grid the performance is worse than for the other values of n_{scb}. The behavior regarding the tilesizes depends again highly on n_b. Smaller n_b prefer smaller tilesizes b_r whereas $n_b = 4000$ prefers high values for b_r.
For the further experimental setups a unique setup shall be selected. Hence, as a compromise, $n_{scb} = 50$ together with $b_l = 175$ and $b_r = 100$ will be used in the following computational runs.

The analysis of the scaling behavior of the block and inter-block parallelization targets the efficiency of the implementations of the QR, QL, RQ and LQ steps. A theoretical analysis of the algorithms regarding runtime has been conducted in Chapter 5. This is here extended by numerical experiments on the scaling behavior of the algorithms. Isolated runs of the kernels



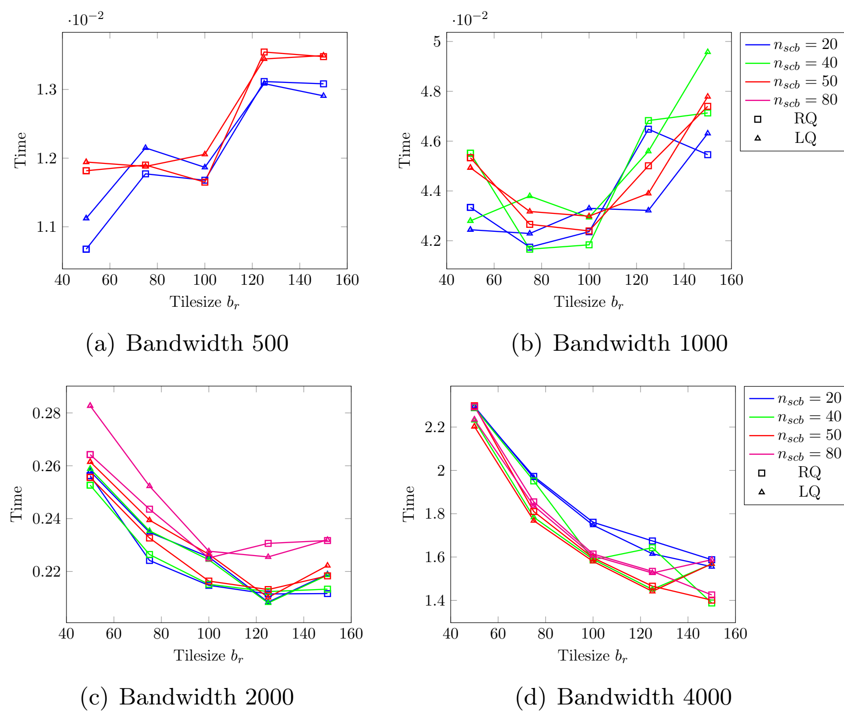
(a) Bandwidth 500

(b) Bandwidth 1000

(c) Bandwidth 2000

(d) Bandwidth 4000

Figure 6.18: Time to solution for various bandwidth n_b: Different values of n_{scb} are plotted over the QR and QL tilesize. A block itself is distributed on a 4×4 process grid. The legends apply to all subfigures.



(a) Bandwidth 500

(b) Bandwidth 1000

(c) Bandwidth 2000

(d) Bandwidth 4000

Figure 6.19: Time to solution for various bandwidth n_b: Different values of n_{scb} are plotted over the RQ and LQ tilesize. A block itself is distributed on a 2×2 process grid. The legends apply to all subfigures.

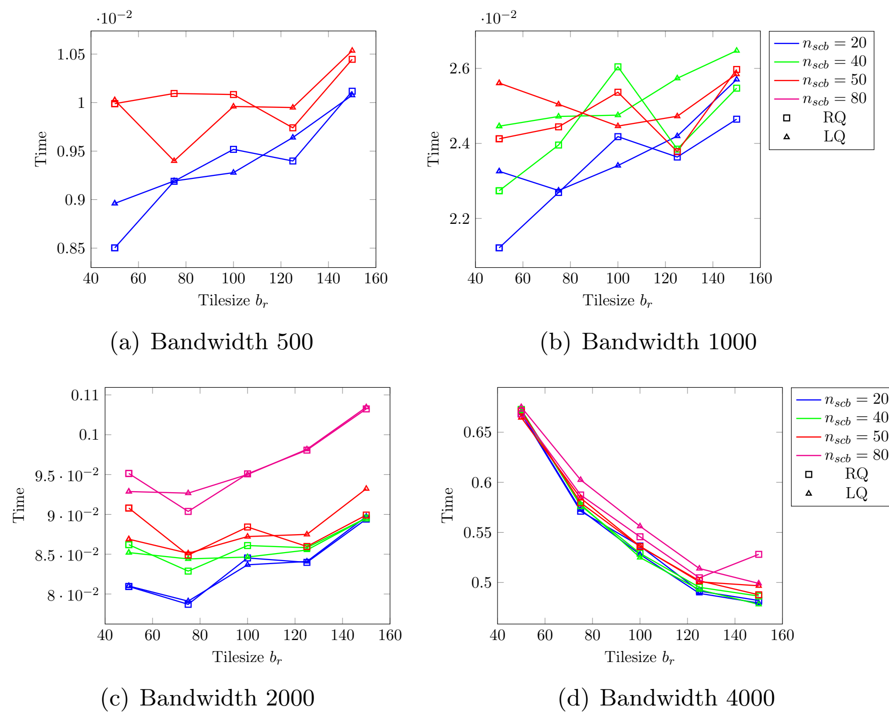


Figure 6.20: Time to solution for various bandwidth n_b: Different values of n_{scb} are plotted over the RQ and LQ tilesize. A block itself is distributed on a 4×4 process grid. The legends apply to all subfigures.

for QR, QL, RQ and LQ have been carried out for the blocksizes 500, 1000, 2000 and 4000. The number of processes per process group and their arrangement in the 2D process grid of the process group have been varied to study the scaling behavior. Every process group was assigned to a single node to exclude influence from other sources.

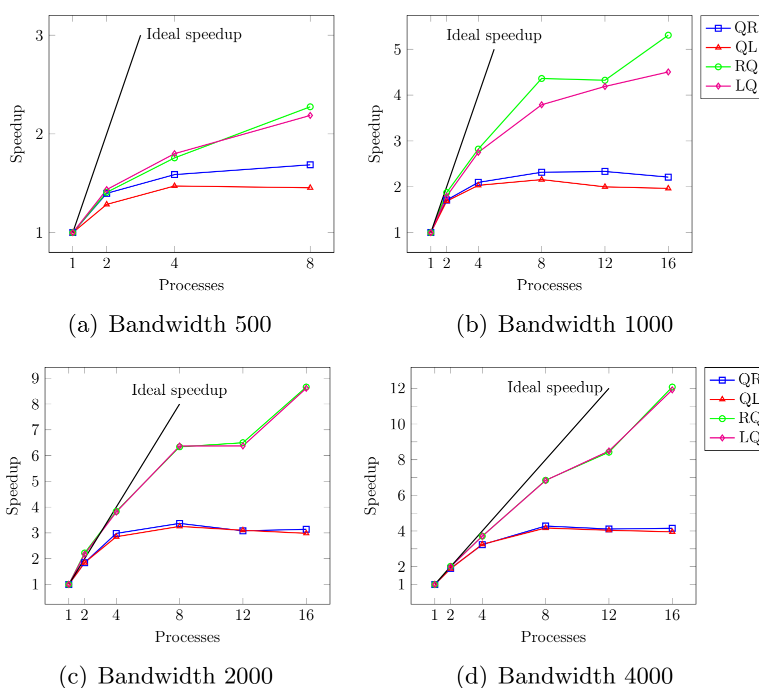
In Figure 6.21 the speedups compared to one process per PG are plotted for the case of arranging the processes in a grid of $1 \times *$ processes. Consequently, the columns of the block are distributed but not the rows. For the left-sided transformations, this leads to stagnation while the right-sided transformations still scale. The reason is the repetition of the T computation over the columns during the left-sided transformation. In the right-sided transformations, the T computation is distributed over the columns and repeated over the rows.

Figure 6.22 shows the opposite picture for process setups of the shape $* \times 1$: While the left-sided transformations scale well, the runtime of the right-sided transformations barely improves after a few processes. Besides the mentioned T computation, which is not distributed for right-sided transformations, Equations (5.26) and (5.27) can be used to analyze the scaling. The right-sided transformations benefit more from increasing p_c than from increasing p_r.

When inspecting Equations (5.9), (5.10), (5.26) and (5.27), it becomes obvious that quadratic or almost quadratic process setups will give the best computational results. To evaluate this, quadratic setups have been used and the result is plotted in Figure 6.23. As expected, the obtained speedups for the left and the right-sided transformation are similar. For the right-sided transformations they are slightly higher when having $n_b \in \{2000, 4000\}$, but in absolute values, left and right-sided transformations end up being almost equally fast when increasing the number of processes per process group. For the left as well as for the right-sided updates, scaling stops at a certain number of processes.

6.3.2 Pipelining approach

The development of runtime when adding additional pairs of PGCs is described in Equations (5.17) and (5.18). The regarding Table 5.1 presents the speedup numbers and parallel efficiency values for the bandwidth-to-matrix-size ratios of 1%, 2% and 5%. To evaluate the formula for SVD, a numerical experiment has been carried out. The blocksize n_b was fixed to 1000 and for the three bandwidth-to-matrix-size ratios the runtime was measured when increasing the number of PGCs. In this experiment, the tilesizes $b_l = 175$, $b_r = 100$ and a ScaLAPACK blocksize of $n_{scb} = 50$ was used. To ensure that other factors do not influence the runtime, a process group consisted of one process and every process group was assigned to a single node. Since in the SVD algorithm the code for the upper matrix half is not the same as for the lower matrix half (at least when having a twisted matrix F and a banded lower triangular matrix G), the experiment was run for $P = 0$ and $P = N$. This means that in the first case, only the lower matrix half algorithm is run, in the latter only the upper matrix half algorithm is run.
The results are given in Figure 6.24. The curves for all bandwidth-to-matrix-size ratios agree very well with the prediction by Equations (5.17) and (5.18). The inclination is well captured and also the absolute speedup values are almost exactly hit for $P = 0$. For $P = N$, the speedup values fall a little behind the theoretical prediction but the inclination of the curves



(a) Bandwidth 500 (b) Bandwidth 1000

(c) Bandwidth 2000 (d) Bandwidth 4000

Figure 6.21: Speedups of the left (ParallelBlockedQR Algorithm 37 and ParallelBlockedQL Algorithm 53) and the right-sided updates (ParallelBlockedLQ Algorithm 59 and ParallelBlockedRQ Algorithm 60) for various numbers of processes in a process group. The process grid is arranged as $1 \times *$, where $*$ is the number of processes. The legends apply to all subfigures.

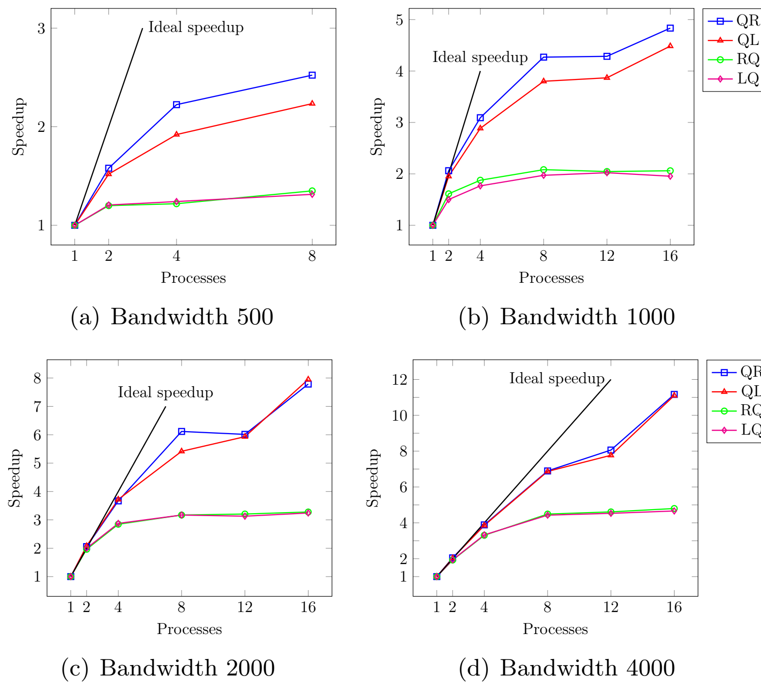
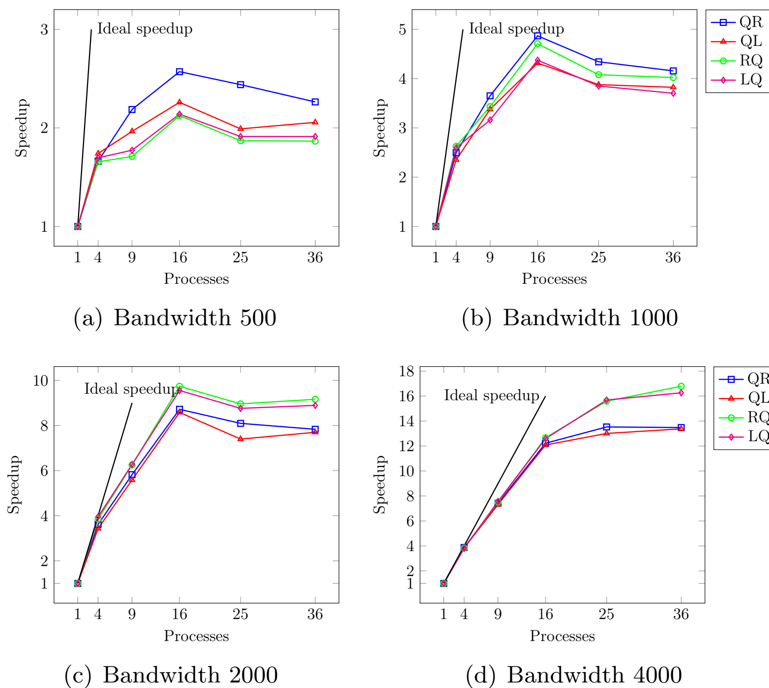
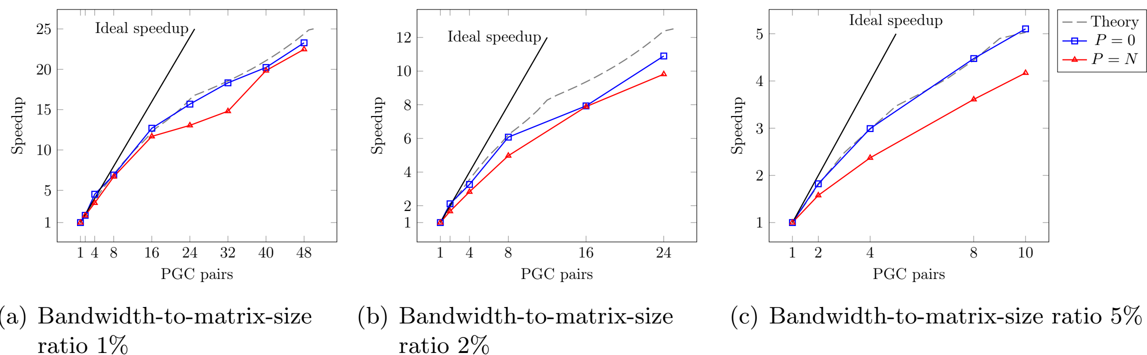


Figure 6.22: Speedups of the left (ParallelBlockedQR Algorithm 37 and ParallelBlockedQL Algorithm 53) and the right-sided updates (ParallelBlockedLQ Algorithm 59 and ParallelBlockedRQ Algorithm 60) for various numbers of processes in a process group. The process grid is arranged as $* \times 1$, where $*$ is the number of processes. The legends apply to all subfigures.



(a) Bandwidth 500
(b) Bandwidth 1000
(c) Bandwidth 2000
(d) Bandwidth 4000

Figure 6.23: Speedups of the left (ParallelBlockedQR Algorithm 37 and ParallelBlockedQL Algorithm 53) and the right-sided updates (ParallelBlockedLQ Algorithm 59 and ParallelBlockedRQ Algorithm 60) for various numbers of processes in a process group. The process grid is arranged as $* \times *$ in a quadratic way, where $*$ is the number of processes. The legends apply to all subfigures.



(a) Bandwidth-to-matrix-size ratio 1%

(b) Bandwidth-to-matrix-size ratio 2%

(c) Bandwidth-to-matrix-size ratio 5%

Figure 6.24: Speedup using only the upper or the lower matrix half algorithm for different numbers of PGC pairs for various bandwidth-to-matrix-size ratios. The block size is fixed to 1000, thus the matrix size of the different cases is 100000 for 1%, 50000 for 2% and 20000 for a bandwidth-to-matrix-size ratio of 5%. The theoretical speedup by Equation (5.17) is plotted as gray dashed line without marks for comparison.

agrees well. Hence, Equations (5.17) and (5.18) have shown to be good estimators for the speedup when increasing the number of PGC pairs.

6.3.3 Parallel execution of the upper and a lower matrix half

In the following, the speedup between twisted and non-twisted computations is inspected. This becomes of special interest if F is not given in banded lower triangular or twisted shape but is provided as $B = F^T F$. Then the question which method of factorization leads to the best results has to be raised and the theoretical predictions and the numerical experiments allow to answer it.
For this experiment, the setup of the pipelining approach with $n_b = 1000$, $b_l = 175$, $b_r = 100$ and $n_{scb} = 50$ was chosen for bandwidth-to-matrix-size ratios of 1%, 2% and 5%. The number of PGCs was fixed to 20 for the computations.
The timings of only the lower matrix half algorithm ($P = 0$) and only the upper matrix half algorithm ($P = N$) are compared with the variant of computing the upper and lower matrix half simultaneously ($P = N/2$). Table 6.5 lists the results of the computations and gives the speedup of $P = N/2$ over the pure upper or pure lower matrix half cases. As expected, the speedup is found to be slightly above 2. Hence, when F has to be generated by factorization before running the algorithm, then it should be computed as twisted matrix.
The case of having both matrices, G and F, as twisted matrices has not been implemented. Also not implemented is the possibility to use the same processes first on the lower matrix half and afterwards on the upper matrix half.

6.3.4 Scaling of the overall implementation

After evaluating the parallelization layers, the overall scaling is going to be analyzed. First, the strong scaling of the twisted SVD algorithm is investigated for bandwidth-to-matrix-size

	n_b/n	1%	2%	5%
$P = 0$	t	106.74	34.07	7.20
	t/t_0	2.03	2.02	2.04
$P = N$	t	114.98	35.39	7.94
	t/t_0	2.18	2.10	2.25
$P = N/2$	t_0	52.67	16.85	3.53

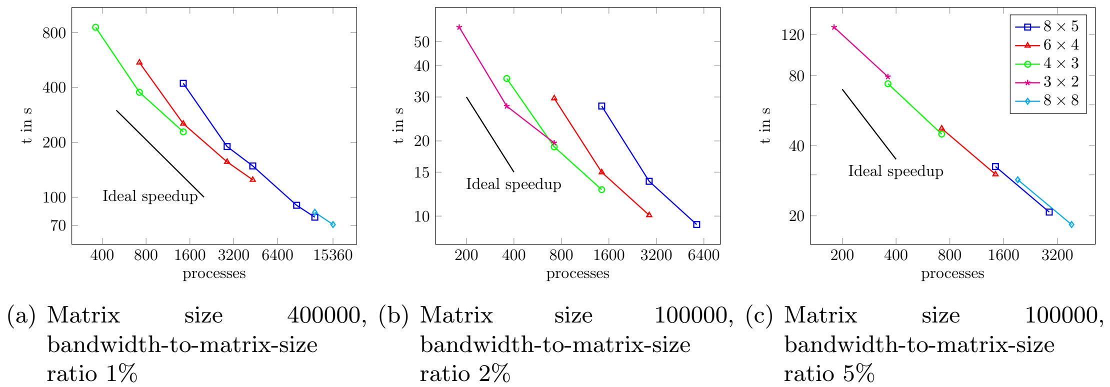
Table 6.5: Speedup by using a twisted factorization compared to a computation solely with the lower matrix half algorithm ($P = 0$) or solely with the upper matrix half algorithm ($P = N$). The test was carried out using 10 pairs of PGCs.

ratios of 1%, 2% and 5%. For the test case $n_b/n = 1\%$, a bandwidth of $n_b = 4000$ was chosen. Thus, this matrix has a size of 400000. A matrix of size 200000 was chosen for a bandwidth-to-matrix-size ratio 2% and 100000 for a bandwidth-to-matrix-size ratio 5%. Hence $n_b = 2000$ and $n_b = 5000$. As tilesizes $b_l = 175$ for QR and QL and $b_r = 150$ for RQ and LQ have been used together with $n_{scb} = 50$.

Figure 6.25 shows the result for the test cases. For a bandwidth-to-matrix-size ratio of 1% good scaling in both parallelization layers can be seen. The single process group setups scale well when increasing the number of PGCs. Also when increasing the number of processes per process group, a good speedup can be observed. The first points of each line correspond to runs with the same number of PGCs. The good scaling between the first points of each line reflects the block scaling and is possible because of the large blocksize.
The middle plot shows the scaling for a bandwidth-to-matrix-size ratio of 2%. Due to the blocksize of $n_b = 2000$, the efficiency of the block parallelization is limited beyond the process group setup 4×3. Because of the larger bandwidth-to-matrix-size ratio, also the scaling by increasing the number of PGCs is limited. The first point of each line corresponds to runs with 10 PGCs, the last point to 40 PGCs. Even if the parallel efficiency is limited, it was still possible to further decrease the runtime by using more processes per process group.
For a bandwidth-to-matrix-size ratio of 5%, the blocksize is 5000. Hence, scaling on block level shows very good results. Due to the bandwidth-to-matrix-size ratio, the maximum number of PGCs is 20. Hence, the algorithm cannot play to its strengths in the PGC scaling.

Additionally to the strong scaling, also the weak scaling of the algorithm is investigated. As explained in the respective part of the twisted Crawford section, the bandwidth-to-matrix-size ratio is kept constant while the blocksize size and the number of processes per process group is increased.
The bandwidth-to-matrix-size ratio was chosen to be 5%, thus $N = 20$. The number of PGCs was fixed at 8. The left-sided tilesize b_l was kept at 175, the right-sided tilesize b_r was chosen to be 150.

Figure 6.26 presents the results of the runs. In the initial run with 24 processes, a single process of a process group has 1000×1000 matrix entries of each block it owns. This share of the block is kept constant throughout the tests. In the last run with 2400 processes, the



(a) Matrix size 400000, bandwidth-to-matrix-size ratio 1%

(b) Matrix size 100000, bandwidth-to-matrix-size ratio 2%

(c) Matrix size 100000, bandwidth-to-matrix-size ratio 5%

Figure 6.25: Scaling of the twisted SVD algorithm employing all parallelization levels for different matrix sizes and bandwidth-to-matrix-size ratios. The different lines show different setups of process groups. The legend applies to all subfigures.

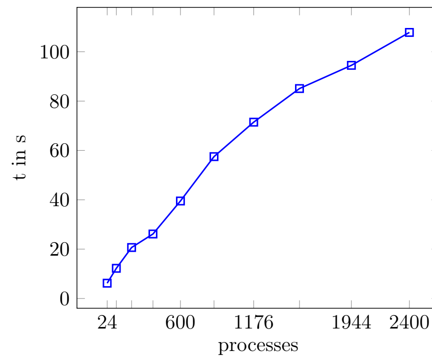


Figure 6.26: Weak scaling of the twisted SVD algorithm.

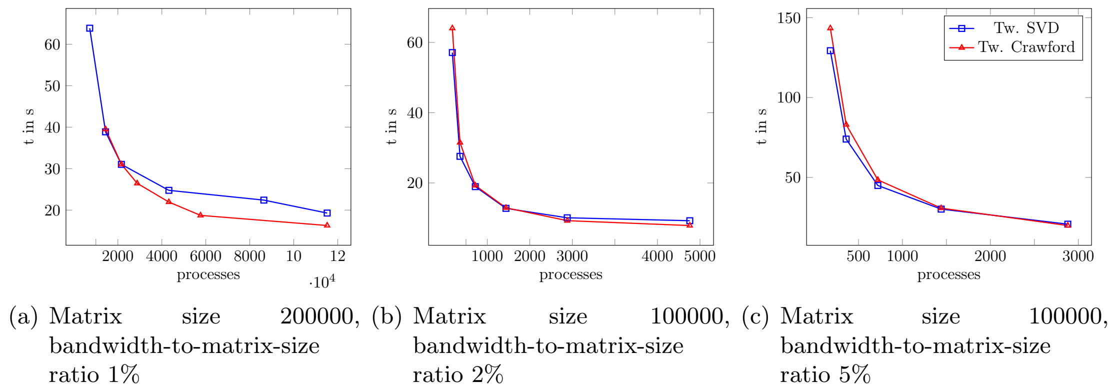
blocksize was at $n_b = 10000$. From the first to the last run, the number of processes grew by a factor of 100 while the runtime of the algorithm grew by a factor of 17.4.

The costs for transforming matrices from the ScaLAPACK format to the algorithm specific format are for the twisted SVD algorithm comparable to those which are listed for the twisted Crawford algorithm in Tables 6.1 to 6.3. Hence, they impact runtime and performance only marginally and thus, a presentation of the numbers is skipped.

6.3.5 Comparison to twisted Crawford algorithm

A short comparison of the runtime of the twisted Crawford and the twisted SVD algorithm shall close this section. This is of interest since the latter can be used to compute the former, especially for ill-conditioned systems. Test cases are a matrix of size 200000 with $n_b/n = 1\%$, and matrices of size 100000 for the cases $n_b/n = 2\%$ and $n_b/n = 5\%$. The SVD algorithm used $b_l = 175$ and $b_r = 150$ together with $n_{scb} = 50$, the eigenvalue algorithm $b = 150$ and n_{scb}.

The results are plotted in Figure 6.27. For the cases $n_b/n = 2\%$ and $n_b/n = 5\%$, both



(a) Matrix size 200000, bandwidth-to-matrix-size ratio 1% (b) Matrix size 100000, bandwidth-to-matrix-size ratio 2% (c) Matrix size 100000, bandwidth-to-matrix-size ratio 5%

Figure 6.27: Comparison between twisted SVD and twisted Crawford algorithm. The best process group setup for a given process number has been selecteds. The legend applies to all subfigures.

curves are almost identical. For a bandwidth-to-matrix-size ratio of 1% the SVD algorithm is up to 18% slower. A general expectation would be that the SVD is slower in any case since it has also to compute a RQ or LQ decomposition instead of just applying an existing Q from right.
These results make the twisted SVD algorithm a strong extension for the twisted Crawford algorithm, especially for ill-conditioned systems since scaling and runtimes are comparable.

For the twisted Crawford algorithm a bandreduction routine is under development by Manin. It will fill the gap between the output of the twisted Crawford algorithm and the bandwidth that are necessary for efficiently running ELPA's two-step solver. Such a bandreduction is also needed to bridge the gap of bandwidth in the singular value decomposition. In [52] such an algorithm is proposed. Another possibility is to extend Manin's approach to banded lower or banded upper triangular matrices.

7 Conclusion

In this thesis, algorithms for banded generalized eigenvalue problems and banded generalized singular value problems have been described and developed. The developed serial algorithm for the generalized singular value problem builds on Lang's extensions to Crawford's ideas for the banded generalized eigenvalue problem. For the eigenvalue as well as for the singular value problem, parallel algorithms aiming for high-performance architectures have been developed. An extensive numerical analysis was carried out for these parallel algorithms. The parallel algorithms have been implemented and the performance of the implementations has been measured. The obtained results agree well with the theoretical predictions of the numerical analysis. The parallel algorithms scale well for thin banded matrices of medium to large size and reduce the time to solution compared to the classical way of solving them as dense problems. This has been demonstrated by comparing the developed twisted Crawford algorithm to the ELPA two-step solver. For this comparison, after running the twisted Crawford algorithm, a bandreduction algorithm was used to further reduce the bandwidth and the second step of ELPA's two-step solver. The obtained speedups depend on the bandwidth-to-matrix-size ratio and have been found between 1.1 and 3.1 for medium size matrices. For large matrices at the limits of ELPA, the speedup is higher, even when using by far less computational resources. The limiting factor, especially for larger matrices, is the bandreduction step. Since it is still under development, higher speedups can be expected for the future.

The developed parallel algorithm for eigenvalue computation allows to tackle larger matrices as it is possible with the state-of-the art eigensolvers, or, for smaller matrix sizes, to solve them faster or to compute the solution by using less computational resources. Hence, this algorithm helps researchers in the field of computational science and engineering to get faster solutions for their problems and to enable them to run more detailed simulations.
The parallel algorithm for the generalized singular value problem allows the same for generalized eigenvalue problems with ill-conditioned positive definite matrices. There it helps to maintain accuracy in comparison to solving it as eigenvalue problem and is a step towards reducing the computational time compared to the classical methods for generalized singular value problems.

The output of the two algorithms are banded matrices of the same bandwidth as the initial matrices have been. This output can be further processed by the second step of a two-step eigenvalue or singular value solver. For larger initial bandwidth, however, an additional bandreduction step is necessary. This has not been developed or implemented in this work. [13, 14] describe such algorithms for the eigenvalue case and they can be adapted to the non-symmetric case of the singular value decomposition. Manin is currently implementing a bandreduction algorithm for the eigenvalue case. Another open point is the implementation of the backtransformation of singular vectors, for which the parallel implementation has been described in this work.

The serial version of the twisted Crawford algorithm has been developed by Lang. Contributions of the author are the serial version of the twisted SVD algorithm and the parallel algorithms of the twisted Crawford as well as the twisted SVD algorithm. Both parallel algorithms have been implemented and optimized by the author. The numerical analysis as well as the performance analysis are also part of the work of the author.

As the development of supercomputing goes on and will probably soon enter the exascale era, the development of algorithms must never stop. The developed eigenvalue algorithm is a small piece in proposing new, problem tailored, algorithms to better exploit the matrix properties. In this sense the twisted Crawford algorithm will be an add-on for a specific subclass of matrices and extend the existing algorithmic machinery for them. The same holds for the twisted SVD algorithm whose application field is even more specialized. However, the available computational routines are by far less developed and therefore the parallel algorithm can be a great asset. In both setups, eigenvalue and singular value computation, exploiting the matrix structure pays off and is a promising way for thin banded matrices.

Bibliography

[1] Alok Aggarwal and S Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 3rd edition, 1999.

[4] Thomas Auckenthaler. *Highly scalable eigensolvers for petaflop applications.* PhD thesis, Technische Universität München, 2013.

[5] Thomas Auckenthaler, Volker Blum, H-J Bungartz, Thomas Huckle, Rainer Johanni, Lukas Krämer, Bruno Lang, Hermann Lederer, and Paul R Willems. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Computing*, 37(12):783–794, 2011.

[6] Thomas Auckenthaler, H-J Bungartz, Thomas Huckle, Lukas Krämer, Bruno Lang, , and Paul R Willems. Developing algorithms and software for the parallel solution of the symmetric eigenvalue problem. *Journal of Computational Science*, 2(3):272–278, 2011.

[7] Z. Bai. The CSD, GSVD, their Applications and Computations. *Preprint Series 958, University of Minnesota*, apr 1992.

[8] Z. Bai and J. W. Demmel. Computing the generalized singular value decomposition. *SIAM J. Sci. Comp.*, 14:1464–1486, 1993.

[9] Grey Ballard, James Demmel, and Nicholas Knight. Avoiding communication in successive band reduction. *ACM Transactions on Parallel Computing (TOPC)*, 1(2):1–37, 2015.

[10] Michael W Berry. Large-scale sparse singular value computations. *The International Journal of Supercomputing Applications*, 6(1):13–49, 1992.

[11] Christian Bischof, Bruno Lang, and Xiaobai Sun. Parallel Tridiagonalization through Two-Step Band Reduction. *Proceedings of the Scalable High-Performance Computing Conference*, 10 1997.

[12] Christian Bischof and Charles Van Loan. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):s2–s13, 1987.

[13] Christian H Bischof, Bruno Lang, and Xiaobai Sun. Algorithm 807: The SBR Toolbox-software for successive band reduction. *ACM Transactions on Mathematical Software (TOMS)*, 26(4):602–616, 2000.

[14] Christian H Bischof, Bruno Lang, and Xiaobai Sun. A framework for symmetric band reduction. *ACM Transactions on Mathematical Software (TOMS)*, 26(4):581–601, 2000.

[15] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[16] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1432–1441. IEEE, 2011.

[17] Richard Butler, Tim Dodwell, Anne Reinarz, Anhad Sandhu, Robert Scheichl, and Linus Seelinger. High-performance dune modules for solving large-scale, strongly anisotropic elliptic problems with applications to aerospace composites. *Computer Physics Communications*, 249:106997, 2020.

[18] Lynn Elliot Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University-Bozeman, College of Engineering, 1969.

[19] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[20] Chin-Chen Chang, Piyu Tsai, and Chia-Chen Lin. SVD-based digital image watermarking scheme. *Pattern Recognition Letters*, 26(10):1577–1586, 2005.

[21] C. R. Crawford. Reduction of a Band-Symmetric Generalized Eigenvalue Problem. *Comm. ACM*, 16(1):41–44, January 1973.

[22] Jan JM Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik*, 36(2):177–195, 1980.

[23] J. W. Demmel, M. Gu, S. Eisenstat, I. Slapničar, K. Veselić, and Z. Drmač. Computing the singular value decomposition with high relative accuracy. *Linear Algebra Appl.*, 299(1-3):21–80, 1999.

[24] Inderjit Dhillon. *A new O (n2) algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem*. PhD thesis, University of California at Berkeley, 1997.

[25] M. Gates, S. Tomov, and J. Dongarra. Accelerating the SVD Two Stage Bidiagonal Reduction and Divide and Conquer Using GPUs. *Parallel Computing*, 71, nov 2017.

[26] Wallace Givens. Computation of plain unitary rotations transforming a general matrix to triangular form. *Journal of the Society for Industrial and Applied Mathematics*, 6(1):26–50, 1958.

[27] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU press, 2012.

[28] Francisco M Gomes and Danny C Sorensen. ARPACK++: A C++ implementation of ARPACK eigenvalue package. *CRPC, Rice University, Houston, TX, Tech. Rep. TR97729*, 1997.

[29] Benedikt Großer and Bruno Lang. Efficient parallel reduction to bidiagonal form. *Parallel Computing*, 25(8):969–986, 1999.

[30] Benedikt Großer and Bruno Lang. An O (n2) algorithm for the bidiagonal SVD. *Linear Algebra and its Applications*, 358(1-3):45–70, 2003.

[31] John L Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[32] Alston S Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, 5(4):339–342, 1958.

[33] Toshiyuki Imamura. Development of a Dense Eigenvalue solver for Exa-scale Systems, August 2018. Talk at MolSSI Workshop / ELSI Conference.

[34] Toshiyuki Imamura, Susumu Yamada, and Masahiko Machida. Development of a high performance eigensolver on the petascale next generation supercomputer system. *Progress in Nuclear Science and Technology*, 2:643–650, 2011.

[35] Andrew V Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing*, 23(2):517–541, 2001.

[36] Walter Kohn and Lu Jeu Sham. Self-consistent equations including exchange and correlation effects. *Physical review*, 140(4A):A1133, 1965.

[37] P Kůs, Andreas Marek, Simone S Köcher, H-H Kowalski, Christian Carbogno, Ch Scheurer, Karsten Reuter, Matthias Scheffler, and Hermann Lederer. Optimizations of the eigensolvers in the ELPA library. *Parallel Computing*, 85:167–177, 2019.

[38] Argonne National laboratory. Theta Supercomputer. https://www.alcf.anl.gov/support-center/theta/theta-machine-overview, July 2020.

[39] Bruno Lang. *Effiziente Orthogonaltransformationen bei der Eigen-Singulärwertberechnung.* Fachbereich der Bergischen Universität-Gesamthochschule Wuppertal, 1997.

[40] Bruno Lang. Efficient reduction of banded hermitian positive definite generalized eigenvalue problems to banded standard eigenvalue problems. *SIAM Journal on Scientific Computing*, 41(1):C52–C72, 2019.

[41] Hermann Lederer. Eigenwert-Löser für Petaflop-Anwendungen - Algorithmische Erweiterungen und Optimierungen. Talk at 7. HPC-Statustagung, HLRS, Stuttgart, December 2017.

[42] Richard B Lehoucq and Danny C Sorensen. Deflation techniques for an implicitly restarted Arnoldi iteration. *SIAM Journal on Matrix Analysis and Applications*, 17(4):789–821, 1996.

[43] LRZ. SuperMUC-NG. https://doku.lrz.de/display/PUBLIC/SuperMUC-NG, June 2020.

[44] Hatem Ltaief, Piotr Luszczek, and Jack Dongarra. High-performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures. *ACM Trans. Math. Software*, 39(3):Art. 16, 22, 2013.

[45] Valeriy Manin and Bruno Lang. Cannon-type triangular matrix multiplication for the reduction of generalized HPD eigenproblems to standard form. *Parallel Computing*, 91:102597, 2020.

[46] Andreas Marek, Volker Blum, Rainer Johanni, Ville Havu, Bruno Lang, Thomas Auckenthaler, Alexander Heinecke, Hans-Joachim Bungartz, and Hermann Lederer. The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter*, 26(21):213201, 2014.

[47] Gordon E Moore et al. Cramming more components onto integrated circuits. *Electronics*, 38, 1965.

[48] MPCDF. COBRA supercomputer. https://www.mpcdf.mpg.de/services/computing/cobra/about-the-system, June 2020.

[49] Christopher C Paige and Michael A Saunders. Towards a generalized singular value decomposition. *SIAM Journal on Numerical Analysis*, 18(3):398–405, 1981.

[50] Roger Penrose. A generalized inverse for matrices. In *Mathematical proceedings of the Cambridge philosophical society*, volume 51, pages 406–413. Cambridge University Press, 1955.

[51] Chiara Puglisi. Modification of the Householder method based on the compact WY representation. *SIAM Journal on Scientific and Statistical Computing*, 13(3):723–726, 1992.

[52] Sivasankaran Rajamanickam. *Efficient algorithms for sparse singular value decomposition.* PhD thesis, University of Florida, 2009.

[53] C.J. Reddy, M.D. Deshpande, C.R. Cockrell, and F.B. Beck. Finite Element Method for Eigenvalue Problems in Electromagnetics. *NASA Technical Paper 3485*, 12 1994.

[54] J.N. Reddy. *An Introduction to the Finite Element Method.* McGraw-Hill, 1993.

[55] Michael Rippl, Daniel Kressner, and Thomas Huckle. A Parallel Algorithm for Banded Generalized Singular Value Decomposition. Unpublished Manuscript, 2020.

[56] Yousef Saad. *Numerical methods for large eigenvalue problems: revised edition.* SIAM, 2011.

[57] PK Sadasivan and D Narayana Dutt. SVD based technique for noise reduction in electroencephalographic signals. *Signal Processing*, 55(2):179–189, 1996.

[58] Robert Schreiber and Charles Van Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10(1):53–57, 1989.

[59] Gerard LG Sleijpen and Henk A Van der Vorst. A Jacobi–Davidson iteration method for linear eigenvalue problems. *SIAM review*, 42(2):267–293, 2000.

[60] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

[61] Bharath K Sriperumbudur, David A Torres, and Gert RG Lanckriet. A majorization-minimization approach to the sparse generalized eigenvalue problem. *Machine learning*, 85(1-2):3–39, 2011.

[62] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. Top 500 supercomputers, June 2020.

[63] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[64] Charles F Van Loan. Generalizing the singular value decomposition. *SIAM Journal on numerical Analysis*, 13(1):76–83, 1976.

[65] Christof Vömel and Jason Slemons. Twisted factorization of a banded matrix. *BIT Numerical Mathematics*, 49(2):433–447, 2009.

[66] JH Wilkinson. The calculation of the eigenvectors of codiagonal matrices. *The Computer Journal*, 1(2):90–96, 1958.