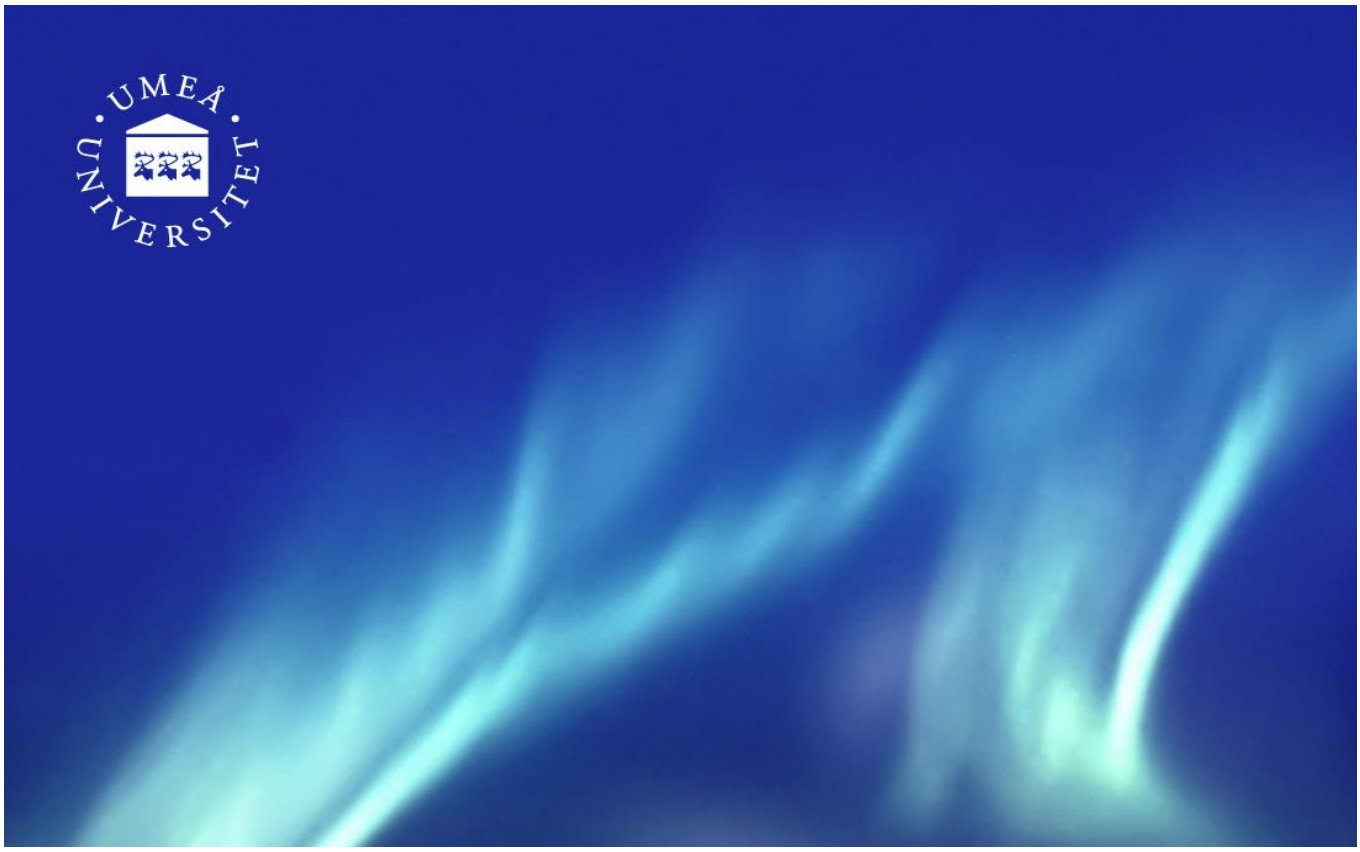

A parallel QZ algorithm for distributed memory HPC systems

by

Björn Adlerborn, Bo Kågström, and Daniel Kressner

UMINF-14/03

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN

A parallel QZ algorithm for distributed memory HPC systems

Björn Adlerborn[†] Bo Kågström[†] Daniel Kressner[‡]

January 27, 2014

Abstract

Appearing frequently in applications, generalized eigenvalue problems represent one of the core problems in numerical linear algebra. The QZ algorithm by Moler and Stewart is the most widely used algorithm for addressing such problems. Despite its importance, little attention has been paid to the parallelization of the QZ algorithm. The purpose of this work is to fill this gap. We propose a parallelization of the QZ algorithm that incorporates all modern ingredients of dense eigensolvers, such as multishift and aggressive early deflation techniques. To deal with (possibly many) infinite eigenvalues, a new parallel deflation strategy is developed. Numerical experiments for several random and application examples demonstrate the effectiveness of our algorithm on two different distributed memory HPC systems.

Key words. generalized eigenvalue problem, nonsymmetric QZ algorithm, multishift, bulge chasing, infinite eigenvalues, parallel algorithms, level 3 performance, aggressive early deflation.

AMS subject classifications. 65F15, 15A18.

1 Introduction

This paper is concerned with the numerical solution of *generalized eigenvalue problems*, which consist of computing the eigenvalues and associated quantities of a matrix pair (A, B) for general complex or real $n \times n$ matrices A, B.

The QZ algorithm proposed in 1973 by Moler and Stewart [33] is the most widely used algorithm for addressing generalized eigenvalue problems with dense matrices. Since then, it has undergone several modifications [15, 36]. In particular, significant speedups on serial machines have been obtained in [24] by extending multishift and aggressive early deflation techniques [9, 10]. In this work, we propose a parallelization of the QZ algorithm that incorporates these techniques as well. Our developments build on preliminary work presented in [1, 2] and extend recent work [20, 21] on parallelizing the QR algorithm for standard eigenvalue problems. In our parallelization, we also cover aspects that are unique to the QZ algorithm, such as the occurrence of possibly many infinite eigenvalues.

[†]Department of Computing Science and HPC2N, Umeå University, SE-90187 Umeå, Sweden(adler@cs.umu.se, bokg@cs.umu.se)

[‡]SB–MATHICSE–ANCHP, EPF Lausanne, Station 8, CH-1015 Lausanne, Switzerland (daniel.kressner@epfl.ch)

Apart from the QZ algorithm, other approaches for solving generalized eigenvalue problems have been considered for parallelization. This includes usage of a synchronous linear processor array [8], nonsymmetric Jacobi algorithms [11, 14] as well as spectral divide-and-conquer algorithms [5, 22, 32, 34].

1.1 Generalized Schur decomposition

Throughout this work, we assume that the pair (A, B) is regular, that is, $\det(A - \lambda B)$ is not zero for all λ. Otherwise, (A, B) needs to be preprocessed to deflate the corresponding singular part in its Kronecker canonical form, either by exploiting underlying structure of applying the GUPTRI algorithm [16].

In the following, we restrict our discussion to matrices with real entries: $A, B \in \mathbb{R}^{n\times n}$; the complex case is treated in an analogous way. The goal of the QZ algorithm consists of computing a *generalized Schur decomposition*

$$Q^T AZ = S, \qquad Q^T BZ = T, \tag{1}$$

where $Q, Z \in \mathbb{R}^{n\times n}$ are orthogonal and the pair (S, T) is in (real) generalized Schur form. This means that T is upper triangular and S is quasi-upper triangular with diagonal blocks of size 1×1 or 2×2. A 1×1 block s_{jj} corresponds to the real eigenvalue $\lambda = s_{jj}/t_{jj}$ of (A, B). In fact, the LAPACK [3] implementation `DHGEQZ` of the QZ algorithm does not even form this ratio but directly returns the pair $(\alpha, \beta) = (s_{jj}, t_{jj})$. This convention has the advantage that it covers infinite eigenvalues in a seamless manner, by letting $\beta = 0$, and it is used in our parallel implementation as well. A 2×2 diagonal block in S corresponds to a complex conjugate pair of eigenvalues, which can be computed from the eigenvalues of

$$\left(\begin{bmatrix} s_{jj} & s_{j,j+1} \\ s_{j+1,j} & s_{j+1,j+1} \end{bmatrix}, \begin{bmatrix} t_{jj} & t_{j,j+1} \\ 0 & t_{j+1,j+1} \end{bmatrix} \right).$$

This is performed by the LAPACK routine `DLAGV2`, which also normalizes T such that $t_{j,j+1} = 0$ and $t_{jj} \geq t_{j+1,j+1} > 0$, using a procedure described in [35].

1.2 Structure of the QZ algorithm

The QZ algorithm proceeds by first computing a decomposition of the form

$$Q^T AZ = H, \qquad Q^T AZ = T, \tag{2}$$

where $Q, Z \in \mathbb{R}^{n\times n}$ are orthogonal and T is again upper triangular, but H is only in upper Hessenberg form, that is, $h_{ij} = 0$ for $i \geq j + 2$. Two different types of algorithms have been proposed to reduce (A, B) to such a *Hessenberg-triangular form*. After an initial reduction of B to triangular form, the original algorithm by Moler and Stewart [33] uses Givens rotations to zero out each entry below the subdiagonal of A. Its memory access pattern lets this algorithm perform rather poorly on standard computing architectures. To address this, a blocked two-stage approach has been proposed by Dackland and Kågström [15]. The first stage reduces (A, B) to *block* Hessenberg-triangular form only, which can be achieved by means of blocked Householder reflectors. The second stage reduces (A, B) further to Hessenberg-triangular form by applying sweeps of Givens rotations. While the first stage can be parallelized quite well, the complex data dependencies make the parallelization of the second stage a more difficult task.

Recently, significant progress has been made in this direction on shared-memory architectures, in the context of reducing a single matrix to Hessenberg form [27]. In the numerical experiments of this paper, we make use of a preliminary parallel implementation of the two-stage algorithm described in [1]. However, it has been demonstrated in [25] that a serial blocked implementation of the rotation-based algorithm by Moler and Stewart outperforms the two-stage approach. We therefore plan to incorporate a parallel implementation of this algorithm as well.

Prior to any reduction, an optional preprocessing step called *balancing* can be used. The balancing described in [37] and implemented in the LAPACK routine `DGGBAL` consists of permuting (A, B) to detect isolated eigenvalues and applying a diagonal scaling transformation to remedy bad scaling. The scaling part is by default turned off in most implementations, such as the LAPACK driver routine `DGGEV` and the Matlab command `eig(A,B)`. We will therefore not consider it any further.

The computation of eigenvectors or, more generally, deflating subspaces of (A, B) requires to postprocess the matrix pair (S, T) in the generalized Schur decomposition (1). More specifically, if the (right) deflating subspace associated with a set of eigenvalues is desired, then these eigenvalues need to be reordered to the top left corners of (S, T). Such a reordering algorithm has been proposed in [26] and its parallelization is discussed in [19].

In this paper, we focus on parallelizing the iterative part of the QZ algorithm which reduces a pair (H, T) in Hessenberg-triangular form to generalized Schur form (S, T). This iterative part consists of three ingredients: bulge chasing, (aggressive early) deflation, and deflation of infinite eigenvalues. In the following we will refer to three different algorithms, with the abbreviations below, all of which use the above three ingredients (see also Appendix A):

- `PDHGEQZ`: This contribution.
- `PDHGEQZ1` : Modified version of parallel QZ algorithm described in Adlerborn et al. [2].
- `KKQZ`: Serial QZ algorithm proposed by Kågstrom and Kressner [24].

2 Parallel algorithms

In what follows, we assume that the reader is familiar with the basics of the implicit shifted QZ algorithm, see [31, 39] for introductions.

2.1 Data layout

We follow the convention of ScaLAPACK [7] for the distributed data storage of matrices. Suppose that $P = P_{\mathrm{r}} \cdot P_{\mathrm{c}}$ parallel processors are arranged in a $P_{\mathrm{r}} \times P_{\mathrm{c}}$ rectangular grid. The entries of a matrix are then distributed over the grid using a 2-dimensional block-cyclic mapping with block size n_{b} in both row and column dimensions. In principle, ScaLAPACK allows for different block sizes for the row and column dimensions but for simplicity we assume that identical block sizes are used.

2.2 Multishift QZ iterations

2.2.1 Chasing one bulge

Consider a Hessenberg-triangular pair (H, T) and two shifts σ_1, σ_2, such that either $\sigma_1, \sigma_2 \in \mathbb{R}$ or $\overline{\sigma_1} = \sigma_2$. For the moment, we will assume that T is invertible. Then the first step of the

classic implicit double shift QZ iteration consists of computing the first column of the shift polynomial:

$$v = (HT^{-1} - \sigma_1 I)(HT^{-1} - \sigma_2 I)e_1 = \begin{bmatrix} \mathsf{X} \\ \mathsf{X} \\ \mathsf{X} \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

where e_1 denotes the first unit vector and the symbol X denotes arbitrary, typically nonzero, entries. Now an orthogonal transformation Q_0 is constructed such that $Q_0^T v$ is a multiple of e_1. This can be easily achieved by a 3×3 Householder reflector. Applying Q_0 from the left to H and T affects the first three rows and creates fill-in below the (sub-)diagonal:

$$H \leftarrow Q_0^T H = \begin{bmatrix} \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}, \quad T \leftarrow Q_0^T T = \begin{bmatrix} \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathsf{X} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}.$$

Here, $\hat{\mathsf{X}}$ is used to denote entries that are affected by the current transformation. To annihilate the two new entries in the first column of T, we use a trick introduced by Watkins and Elsner [40] and shown to be numerically backward stable in [24]. Let Z_0 be a Householder reflector that maps $T^{-1}e_1$ to a multiple of the first unit vector. Then applying Z_0 from the right affects the first three columns and results in the nonzero pattern

$$H \leftarrow HZ_0 = \begin{bmatrix} \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}, \quad T \leftarrow TZ_0 = \begin{bmatrix} \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ \hat{0} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ \hat{0} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathsf{X} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}.$$

The region $\big(H(2:4,1:3), T(2:4,1:3)\big)$ is called the *bulge pair*. This encodes the information contained in the shifts σ_1, σ_2, in a way made concrete in [38]. By an analogous procedure, the trailing two entries in the first column of H are annihilated by a Householder transformation from the left and, subsequently, the subdiagonal entries in the second column of T are annihilated by a Householder transformation from the right. In effect, the bulge pair is moved one step towards the bottom right corner:

$$H \leftarrow Q_1^T H Z_1 = \begin{bmatrix} \mathsf{X} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ \hat{0} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ \hat{0} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ 0 & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}, \quad T \leftarrow Q_1^T T Z_1 = \begin{bmatrix} \mathsf{X} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ 0 & \hat{0} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ 0 & \hat{0} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \cdots \\ 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\ 0 & 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}. \tag{3}$$

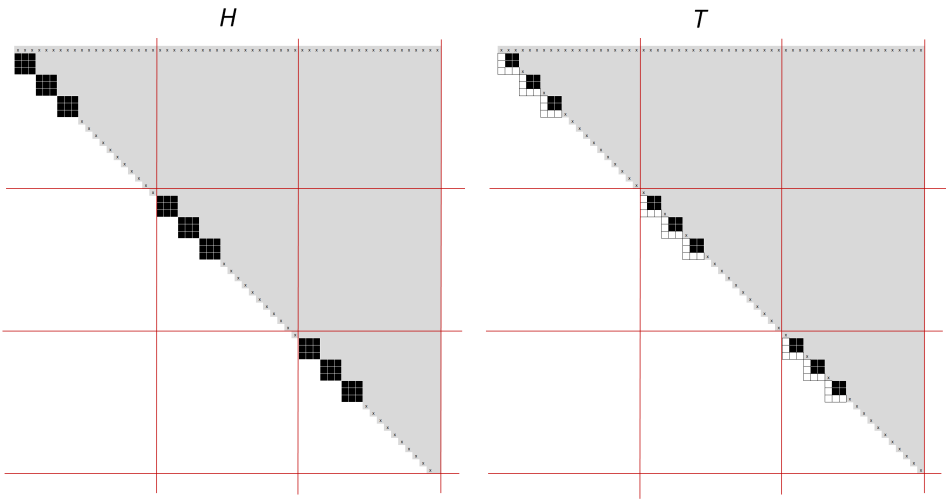

Figure 1: Three bulge chains, where each chain consists of three bulges (black boxes). The black 3×3 blocks in the H-part are dense, while the associated blocks in the T-part have zero elements in the last row and first column, respectively. Only parts of the matrices are displayed. The solid red lines represent block/process borders.

2.2.2 Chasing several bulges in parallel

In principle, the described process of bulge chasing can be continued until the bulge pair disappears at the bottom right corner, which would complete one double shift QZ iteration. However, the key to efficient serial and parallel implementations of the QZ algorithm is to chase several bulges at the same time. For example, after the bulge pair in (3) has been chased two steps further, we can immediately introduce another bulge pair belonging to another two shifts, without disturbing the existing bulge pair. We then have a chain of two tightly[1] packed bulges, which can be chased simultaneously. In practice, we will work with chains containing significantly more than two bulges to attain good node performance.

Another key to create potential for good parallel performance is to delay updates as much as possible. We chase bulges within a window (i.e., a principal submatrix), similar to the technique described in [20]. Only after the bulges have been chased to the bottom of the window we update the remaining parts of the matrix pair (H, T) outside the window, to the right and above, using level-3 BLAS. After the off-diagonal update, the window is placed on a new position so that the bulges can be moved further down the diagonal of (H, T). As in [20] we use several windows, up to $\min(P_\mathrm{r}, P_\mathrm{c})$, each containing a chain of bulges, and deal with them in parallel.

Figure 1 illustrates the described technique for three active windows, each containing a bulge chain consisting of three bulges. Each window is owned by a single process, leading to *intra-block* chases. Windows that overlap process borders lead to *inter-block* chases. In the algorithm, we alternate between intra-block and inter-block chases, see [20] for more details.

Generally, we use the undeflatable eigenvalues returned by aggressive early deflation described in § 2.3 as shifts for the QZ iteration. Exceptionally, it may happen that there are not sufficiently many undeflatable eigenvalues available. In such cases, we apply the QZ algorithm

[1] In fact, as recently shown in [29] for the QR algorithm, it is possible and beneficial to pack the bulges even closer.

(PDHGEQZ or PDHGEQZ1) to a sub-problem for obtaining additional shifts.

Each bulge consumes two shifts, and occupies a 3×3 principal submatrix. Assuming that each window is of size $n_{\mathsf{b}} \times n_{\mathsf{b}}$, we will pack at most $n_{\mathsf{b}}/6$ bulges in a window to be able to move all bulges across the process border during a single *inter-block* chase. The number of active windows is then given by

$$\min(\lceil 3n_{\text{shifts}}/n_{\mathsf{b}} \rceil, P_{\text{r}}, P_{\text{c}}),$$

where n_{shifts} is the total number of shifts to be used.

2.3 Aggressive early deflation (AED)

Classically, the convergence of the QZ algorithm is monitored by inspecting the subdiagonal entries of H. A subdiagonal entry $h_{k+1,k}$ is declared negligible if it satisfies

$$|h_{k+1,k}| \leq \mathbf{u} \times (|h_{k,k}| + |h_{k+1,k+1}|), \tag{4}$$

where $\mathbf{u}$ denotes the unit roundoff ($\approx 10^{-16}$ in double precision arithmetic). Negligible subdiagonal entries can be safely set to zero, deflating the generalized eigenvalue problem into smaller subproblems. *Aggressive early deflation (AED)* is a technique proposed by Brahman, Byers and Mathias [10] for the QR algorithm, that goes beyond (4) and significantly speeds up convergence.

2.3.1 Basic algorithm

In the following, we give a brief overview of AED for the QZ algorithm as proposed in [24]. First, the Hessenberg-triangular pair is partitioned as

$$(H,T) = \left(\begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ 0 & H_{32} & H_{33} \end{bmatrix}, \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ 0 & T_{22} & T_{23} \\ 0 & 0 & T_{33} \end{bmatrix} \right),$$

such that $H_{32} \in \mathbb{R}^{n_{\mathsf{AED}} \times 1}$ and $H_{33}, T_{33} \in \mathbb{R}^{n_{\mathsf{AED}} \times n_{\mathsf{AED}}}$, where $n_{\mathsf{AED}} \ll n$ denotes the size of the so called AED window (H_{33}, T_{33}).

By applying the QZ algorithm, the AED window is reduced to (real) generalized Schur form:

$$Q_{33}^T H_{33} Z_{33} = \hat{H}_{33}, \qquad Q_{33}^T T_{33} Z_{33} = \hat{T}_{33}.$$

Performing the corresponding orthogonal transformation of (H,T) yields

$$(H,T) \leftarrow \left(\begin{bmatrix} H_{11} & H_{12} & \hat{H}_{13} \\ H_{21} & H_{22} & \hat{H}_{23} \\ 0 & s & \hat{H}_{33} \end{bmatrix}, \begin{bmatrix} T_{11} & T_{12} & \hat{T}_{13} \\ 0 & T_{22} & \hat{T}_{23} \\ 0 & 0 & \hat{T}_{33} \end{bmatrix} \right), \tag{5}$$

where $s = Q_{33}^T H_{32}$ is the so called *spike* that contains the newly introduced nonzero entries below the subdiagonal of H. If the last entry of the spike satisfies

$$|s_{n_{\mathsf{AED}}}| \leq \mathbf{u} \times \|H\|_F, \tag{6}$$

it can be safely set to zero. This deflates a real eigenvalue of the matrix pair (5), provided that the diagonal block at the bottom right corner of $\hat{H}_{33}$ is 1×1. If this diagonal block is 2×2, the

corresponding complex conjugate pair of eigenvalues can only be deflated if the last two entries of the spike both satisfy (6).

If deflation was successful, the described process is repeated for the remaining nonzero entries of the spike. Otherwise, the generalized Schur form (H_{33}, T_{33}) is reordered to move the undeflatable eigenvalue to the top left corner. In turn, an untested eigenvalue is moved to the bottom right corner. After all eigenvalues of (H_{33}, T_{33}) have been checked for convergence by this procedure, the last step of AED consists of turning the remaining undeflated part of the pair (H, T) back to Hessenberg-triangular form.

2.3.2 Parallel implementation

As already demonstrated in [21], a careful implementation of AED is vital to achieving good overall parallel performance. From the discussion above, we identify three computational tasks:

1. Reducing an $n_{\mathsf{AED}} \times n_{\mathsf{AED}}$ Hessenberg-triangular matrix pair to generalized Schur form.
2. Reordering the eigenvalues of an $n_{\mathsf{AED}} \times n_{\mathsf{AED}}$ matrix pair in generalized Schur form.
3. Reducing a general matrix pair of size at most $n_{\mathsf{AED}} \times n_{\mathsf{AED}}$ to Hessenberg-Triangular form.

Only for very small values of n_{AED}, say $n_{\mathsf{AED}} \leq 201$, it makes sense to perform these tasks serially and call the corresponding LAPACK routines. For larger values of n_{AED}, parallel algorithms are used for all three tasks.

To perform Task 1, we call our, now modified, parallel implementation `PDHGEQZ1` [2] of the QZ algorithm with (serially performed) AED for moderately sized problems, say $n_{\mathsf{AED}} \leq 6000$, and recursively call the parallel implementation `PDHGEQZ` described in this paper for larger problems.

To perform Task 2, we make use of ideas described in [19] for reordering eigenvalues in parallel. To create potential for parallelism and good node performance, we work with groups of undeflatable eigenvalues instead of moving them individually. Such a group is reordered to the top left corner by an algorithm quite similar to the parallel bulge chasing discussed in § 2.2.2. We refer to [20, Sec. 2.3] for more details.

To perform Task 3, we call the parallel implementation of Hessenberg-triangular reduction described in [1].

After all three tasks have been performed, it remains to apply the corresponding orthogonal transformations to the off-diagonal parts of (H, T), above and to the right of the AED window. These updates are preformed by calls to the PBLAS routine `PDGEMM`.

2.3.3 Avoiding communication via data redistribution

Since $n_{\mathsf{AED}} \ll n$, the computational intensity of AED is comparably small and the parallel distribution of the matrices on a grid of $P_{\mathrm{r}} \times P_{\mathrm{c}}$ of processors may lead to significant communication overhead. This has been observed for the parallel QR algorithm [21] and holds for the parallel QZ algorithm as well.

A simple cure to this phenomenon is to redistribute data before performing AED, to limit the amount of participating processors and to keep the communication overhead under control. More specifically, the $n_{\mathsf{AED}} \times n_{\mathsf{AED}}$ AED window is first redistributed across a smaller $P_{\mathsf{AED}} \times$

P_{AED} grid, then Tasks 1 to 3 discussed above are performed, and finally the AED window is distributed back to the $P_\mathrm{r} \times P_\mathrm{c}$ grid. The choice of P_{AED} depends on n_{AED}, see § 3.2.

To get an impression of the benefits from this technique: Without redistribution, the total time spent on AED is 2095 seconds when solving a $32\,000 \times 32\,000$ random generalized eigenvalue problems on a 8×8 grid of Akka (see § 3.1). When using $P_{\mathsf{AED}} = 4$, which means redistributing the AED window to a 4×4 grid, the total time for AED reduces to 1283 seconds.

2.4 Deflation of infinite eigenvalues

If B is (nearly) singular, it can be expected that one or several diagonal entries of the triangular matrix T in the Hessenberg-triangular form are (nearly) zero. Following the LAPACK implementation of the QZ algorithm, we consider a diagonal entry t_{ii} negligible if

$$|t_{ii}| \leq \mathbf{u} \cdot ||T||_F \tag{7}$$

holds. Setting such an entry to zero and reordering it to the bottom right or top left corner allows to deflate an infinite eigenvalue. In the absence of roundoff error, this reordering is automatically effected by QZ iterations [38]. However, to avoid unnecessary iterations and the loss of infinite eigenvalues due to roundoff error, it is important to take care of infinite eigenvalues separately.

2.4.1 Basic algorithm

In the following, we briefly sketch the mechanism for deflating infinite eigenvalues proposed in [33]. Suppose that H is unreduced, i.e. $h_{k+1,k} \neq 0$ for all subdiagonal elements, and that the third diagonal entry of T is zero:

$$H = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \cdots \\ \times & \times & \times & \times & \times & \times & \cdots \\ 0 & \times & \times & \times & \times & \times & \cdots \\ 0 & 0 & \times & \times & \times & \times & \cdots \\ 0 & 0 & 0 & \times & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & \times & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}, \qquad T = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \cdots \\ 0 & \times & \times & \times & \times & \times & \cdots \\ 0 & 0 & 0 & \times & \times & \times & \cdots \\ 0 & 0 & 0 & \times & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & 0 & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}.$$

Applying a Givens rotations to columns 2 and 3 makes the second diagonal entry zero as well, but introduces an additional nonzero in H:

$$H \leftarrow \begin{bmatrix} \times & \hat{\times} & \hat{\times} & \times & \times & \times & \cdots \\ \times & \hat{\times} & \hat{\times} & \times & \times & \times & \cdots \\ 0 & \hat{\times} & \hat{\times} & \times & \times & \times & \cdots \\ 0 & \hat{\times} & \hat{\times} & \times & \times & \times & \cdots \\ 0 & 0 & 0 & \times & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & \times & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}, \qquad T \leftarrow \begin{bmatrix} \times & \hat{\times} & \hat{\times} & \times & \times & \times & \cdots \\ 0 & 0 & \hat{\times} & \times & \times & \times & \cdots \\ 0 & 0 & 0 & \times & \times & \times & \cdots \\ 0 & 0 & 0 & \times & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & 0 & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}.$$

This nonzero entry can be annihilated by applying a Givens rotations to rows 3 and 4:

$$H \leftarrow \begin{bmatrix} \times & \times & \times & \times & \times & \times & \cdots \\ \times & \times & \times & \times & \times & \times & \cdots \\ 0 & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ 0 & 0 & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ 0 & 0 & 0 & \times & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & \times & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}, \qquad T \leftarrow \begin{bmatrix} \times & \times & \times & \times & \times & \times & \cdots \\ 0 & 0 & \times & \times & \times & \times & \cdots \\ 0 & 0 & 0 & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ 0 & 0 & 0 & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ 0 & 0 & 0 & 0 & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & 0 & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}.$$

By an analogous procedure, the zero diagonal entries of T can be moved one position further upwards. A Givens rotation can then be applied to rows 1 and 2 in order to annihilate the first subdiagonal entry of H, finally yielding

$$H = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \cdots \\ 0 & \times & \times & \times & \times & \times & \cdots \\ 0 & \times & \times & \times & \times & \times & \cdots \\ 0 & 0 & \times & \times & \times & \times & \cdots \\ 0 & 0 & 0 & \times & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & \times & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}, \qquad T = \begin{bmatrix} 0 & \times & \times & \times & \times & \times & \cdots \\ 0 & \times & \times & \times & \times & \times & \cdots \\ 0 & 0 & \times & \times & \times & \times & \cdots \\ 0 & 0 & 0 & \times & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & 0 & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}.$$

Thus, an infinite eigenvalue can be deflated at the top left corner. An analogous procedure can be used to deflate infinite eigenvalues at the bottom right corner. This reduces the required operations if the zero diagonal entry of T is closer to that corner.

2.4.2 Parallel implementation

A number of applications lead to matrix pencils with a substantial fraction of infinite eigenvalues, see § 3 for examples. In this case, applying the above procedure to deflate each infinite eigenvalue individually is clearly not very efficient in a parallel environment.

Instead, we aim at deflating several infinite eigenvalues simultaneously. To do this in a systematic manner and attain good node performance, we proceed similarly as in the parallel multishift QZ iterations and parallel eigenvalue reordering algorithm discussed in § 2.2.2 and § 2.3.2, respectively. Up to $\min(P_\mathrm{r}, P_\mathrm{c})$ computational windows are placed on the diagonal of (H, T), such that each window is owned by a single diagonal process. Within each window, the negligible diagonal entries of T are identified according to (7), zeroed, and they are all moved either to the top left corner or to the bottom right corner of T, depending which corner is nearest. Only then we perform the corresponding updates outside the windows, by accumulating the rotations into orthogonal matrices and performing matrix-matrix multiplications.

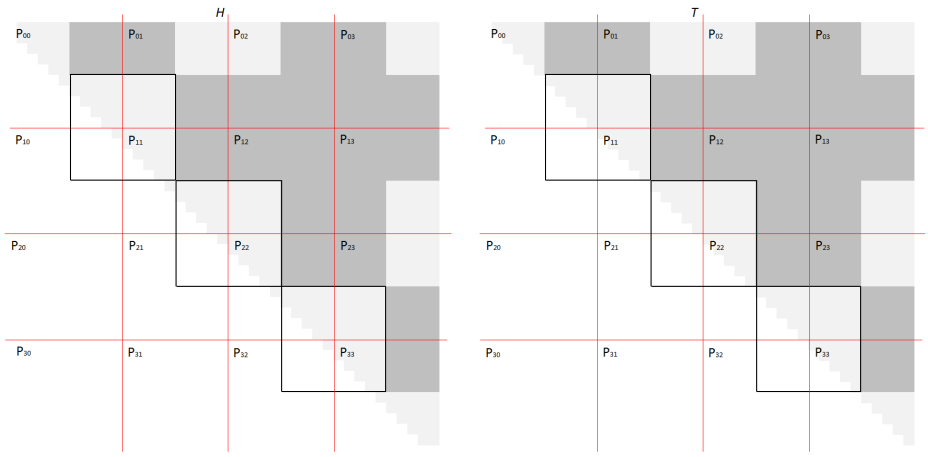

Figure 2: Crossborder layout of three windows during the parallel deflation of infinite eigenvalues. Computational windows and process borders indicated by black and red solid lines, respectively. Darker-gray areas indicate regions that need to be updated after the windows have been processed.

In the next step, all computational windows are shifted upwards or downwards. In effect, the zero diagonal entries of T are located in the bottom or top parts of the windows. Moreover, as illustrated in Figure 2, the windows now overlap process borders. To move the zero diagonal entries in the first computational window across the process border, we proceed as follows:

1. The two processes on the diagonal (P_{00} and P_{11}) exchange their parts of the window and receive the missing off-diagonal parts from the other two processes (P_{01} and P_{10}). In effect, two identical copies of the computational window are created.

2. Each of the two diagonal processes (P_{00} and P_{11}) identifies and zeroes negligible diagonal entries of T within the window, and moves all of them to the top left corner (or the bottom right corner, whichever is nearest the top left or bottom right corner of T.).

3. The two off-diagonal processes (P_{01} and P_{10}) receive the updated off-diagonal parts of the window from one of the on-diagonal processes (P_{00} or P_{11}). The accumulated orthogonal transformation matrices generated in Step 2 are broadcasted to the blocks on both sides of the process borders (to P_{00}, P_{01} and to $P_{02}, P_{03}, P_{11}, P_{12}, P_{13}$).

   For updating the parts of the matrix outside the window, neighboring processes holding cross-border regions exchange their data in parallel and compute the updates in parallel.

To achieve good parallel performance, the above procedure needs to be applied to several computational windows simultaneously. However, some care needs to be applied in order to avoid intersecting scopes of the diagonal processes in Steps 1 and 2. Following an idea proposed in [20], this can be achieved as follows. We number the windows from bottom to top, starting with index 0. First all even-numbered windows are treated and only then all odd-numbered windows are treated in parallel.

When the zero diagonal entries of T have been moved across the process borders, the next step again consists of shifting all computational windows upwards or downwards such that they are owned by diagonal processes. This allows to repeat the whole procedure, until all zero diagonal entries of T arrive at one of the corners and admit deflation. The parallel procedure of chasing zeros along the diagonal of T and deflating infinite eigenvalues is illustrated and described in some more detail in Figures 3–7.

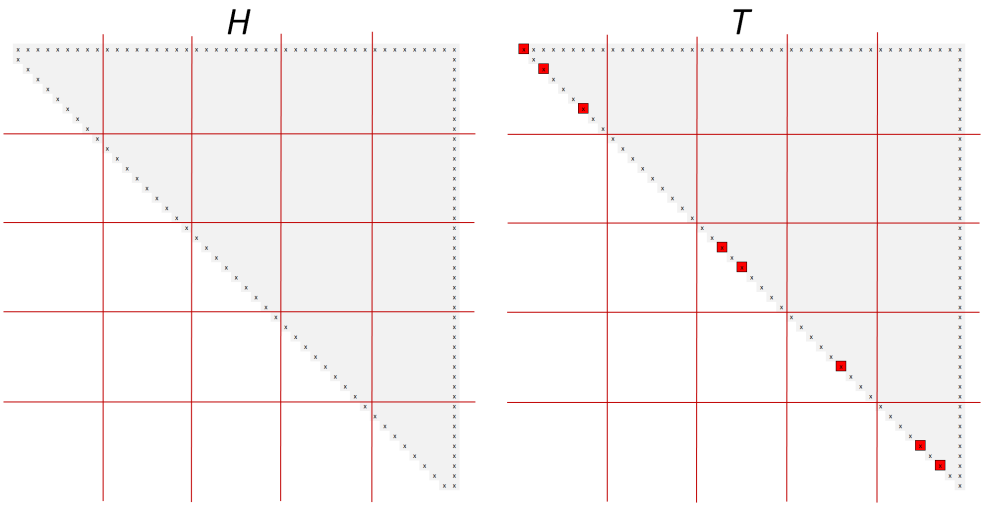

Figure 3: Example where 8 zeros have been identified on the diagonal of T, indicated by filled red squares. The grid is of size 5×5 and process borders are indicated by red solid lines.

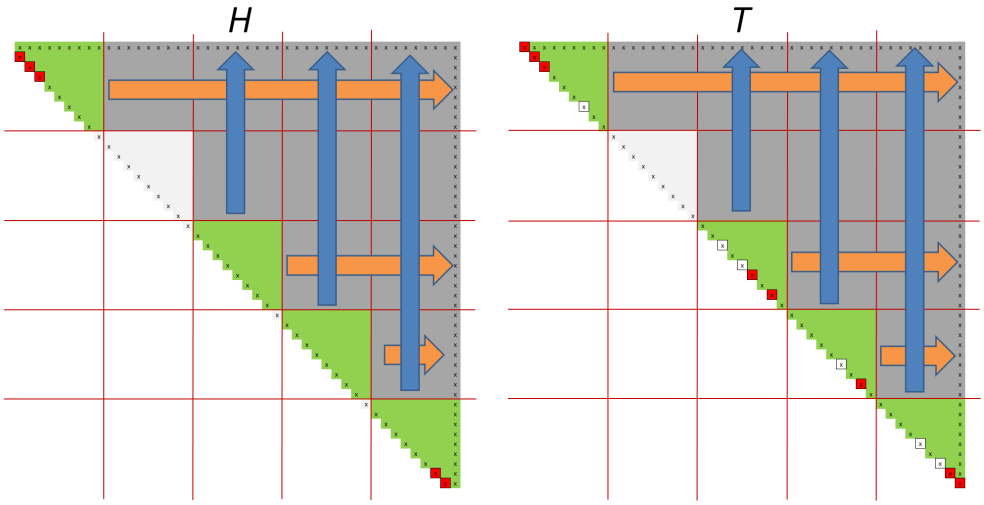

Figure 4: *Intra-block* chase where all diagonal blocks are processed in parallel. The 4 active diagonal blocks, within which the zero diagonal entries of T are moved up or down, are marked in green. The old positions of the zero diagonal entries are marked by white squares, while the new positions or (so far) not moved zeros are marked by red squares. Three and two deflations are performed at the top left and bottom right diagonal blocks, respectively, leading to the zero subdiagonal entries of H marked by red squares on the subdiagonal of H. The chase is followed by horizontal and vertical broadcasts of accumulated transformations, so that off-diagonal processors can update their parts of H and T in parallel. The area affected by the off-diagonal updates is marked in dark grey.

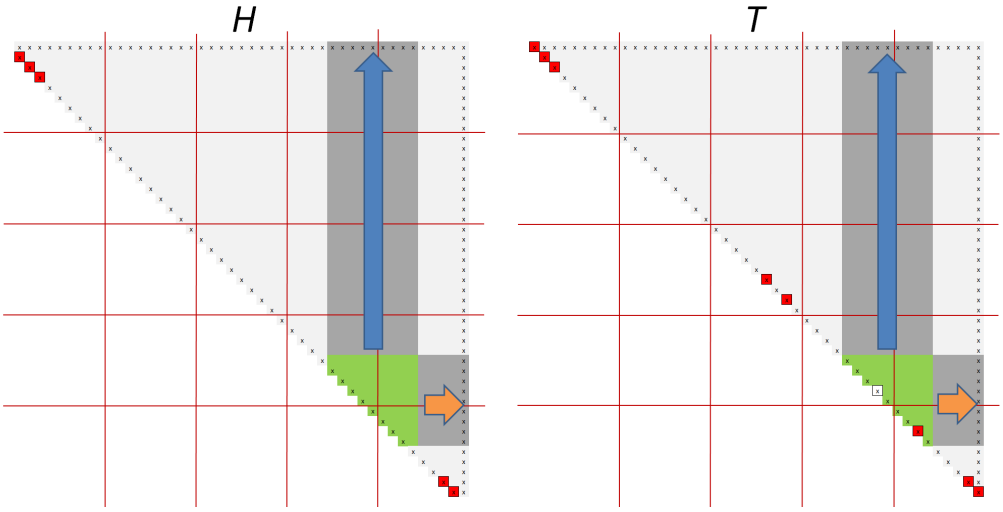

Figure 5: *Inter-block* chase, phase one, where all even-numbered diagonal blocks, indexed by $0 \ldots n_{\text{block}} - 1$ from bottom to top, are processed in parallel. In this case there is only one active diagonal block. See Figure 4 for an explanation of the color marking.

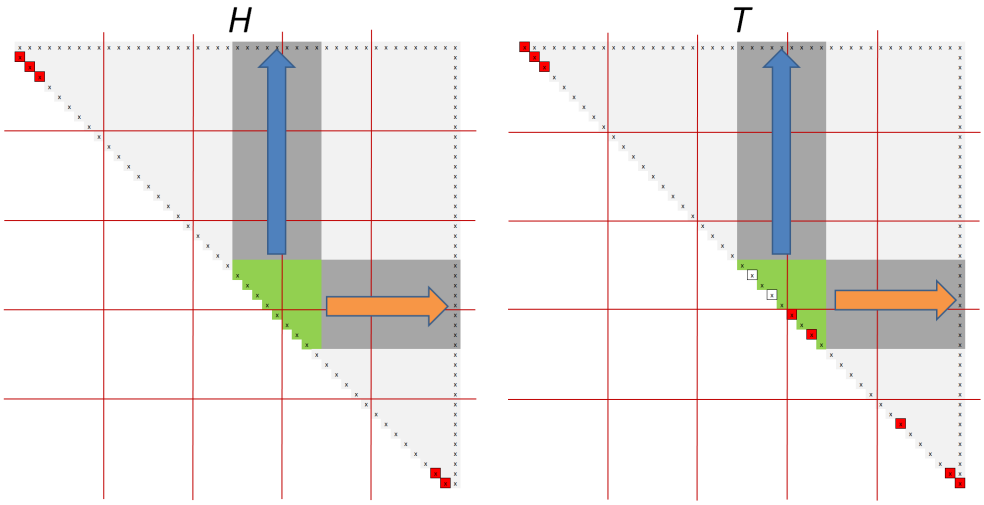

Figure 6: *Inter-block* chase, phase two, where all odd-numbered diagonal blocks are processed in parallel. See Figure 4 for an explanation of the color marking.

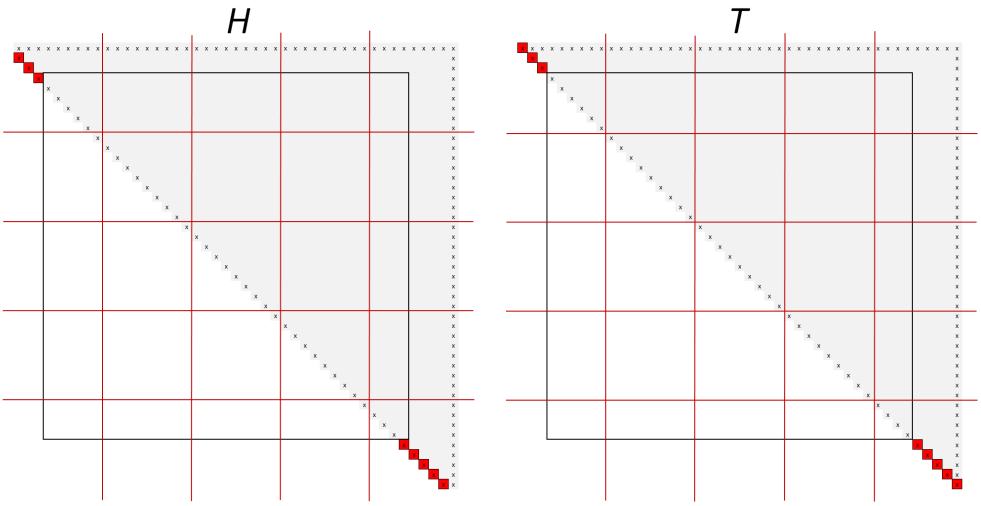

Figure 7: *Intra-block* and *inter-block* chases are repeated until all zeros on the diagonal of T have been moved to the top left or bottom right corners. A total of 8 deflations can be made. The remaining problem size is decreased accordingly, indicated by the black square on H and T.

3 Computational Experiments

3.1 Computing environment

The algorithms described in § 2 have been implemented in a Fortran 90 routine `PDHGEQZ`. Following the ScaLAPACK style, we use BLACS [7] for communication and call ScaLAPACK / PBLAS or LAPACK / BLAS routines whenever appropriate.

We have used the two HPC2N computer systems Akka and Abisko, see Table 1, for our computational experiments. For all our experiments on Akka, we used the Fortran 90 compiler mpif90 version 4.0.13 from the PathScale EKOPath(tm) Compiler Suite with the optimization flag `-O3`. For all our experiments on Abisko, we used the Intel compiler mpiifort version 13.1.2 with the optimization flag `-O3`.

On Abisko, each floating point unit (FPU) is shared between two cores. To attain near to optimal performance, we only utilize 50% of the cores within a compute node. For example, a grid $P_r \times P_c = 10 \times 10$ is allocated on 5 compute nodes, using up to 24 cores on each node. Although each core on Akka has its own FPU, we also do not utilize more than 50% of the cores on Akka either, for memory capacity reasons. Each compute node on Akka has 8 cores, so a 10×10 grid is allocated on 25 compute nodes, using 4 cores on each node.

3.2 Selection of n_b and n_{AED}

Our implementation `PDHGEQZ` of the parallel QZ algorithm is controlled by a number of parameters, in particular the AED window size n_{AED} and the number of shifts n_{shifts}. For choosing these parameters, we follow the strategy proposed in [20, 21] for the parallel QR implementation. The block size n_b, which determines the data distribution block size, is set to $n_b = 130$ when $n > 2000$. If $n \leq 2000$, an n_b value of 60 is near optimal. This is the same for both Akka

Table 1: Akka and Abisko at the High Performance Computing Center North (HPC2N)

Akka	64-bit low power Intel Xeon Linux cluster 672 dual socket quadcore L5420 2.5GHz nodes 256KB dedicated L1 cache, 12MB shared L2 cache, 16GB RAM per node Cisco Infiniband and Gigabit Ethernet, 10 GB/sec bandwidth OpenMPI 1.4.4, GOTO BLAS 1.13 LAPACK 3.4.0, ScaLAPACK/BLACS/PBLAS 2.0.1
Abisko	64-bit AMD Opteron Linux Cluster 322 nodes with a total of 15456 CPU cores Each node is equipped with 4 AMD Opteron 6238 (Interlagos) 12 core 2.6 GHz processors 10 'fat' nodes with 512 GB RAM each, as well as 312 'thin' nodes with 128 GB RAM each 40 Gb/s Mellanox Infiniband Intel-MPI 13.1.2, ACML package 5.3.1 (includes LAPACK 3.4.0) ScaLAPACK/BLACS/PBLAS 2.0.2

and Abisko.

As explained in § 2.3.3, the $n_{\mathsf{AED}} \times n_{\mathsf{AED}}$ AED window is redistributed to a $P_{\mathsf{AED}} \times P_{\mathsf{AED}}$ grid before performing AED. The redistribution itself requires some computation and communication, but the cost is small compared to the whole AED process, see [21]. The local size of the AED window is set to a new value n_{local} during the data redistribution. On Akka and Abisko, we choose $n_{\text{local}} = n_{\mathsf{b}}\lceil 384/n_{\mathsf{b}} \rceil$, implying that each process involved in the AED should own at least a 384×384 block of the whole AED window. The value 384 is taken from the QR implementation, see [21]. Then P_{AED} is chosen as the smallest value that satisfies

$$n_{\mathsf{AED}} \leq (1 + P_{\mathsf{AED}}) \cdot n_{\text{local}}. \tag{8}$$

Before the redistribution, we also adjust n_{b} to a value better suited for the problem size, i.e. 60 or 130 depending on n_{AED}. When $n_{\mathsf{AED}} \leq 6\,000$, we use `PDHGEQZ1` to compute the generalized Schur decomposition of the adjusted AED window, otherwise `PDHGEQZ` is called recursively.

3.3 Accuracy

All experiments have been performed in double precision arithmetic ($\epsilon_{\text{mach}} \approx 2.2 \times 10^{-16}$). For each run of the parallel QZ algorithm, we have verified its backward stability by measuring

$$R_{\text{r}} = \max\left\{ \frac{\|Q^T A Z - S\|_F}{\|A\|_F}, \frac{\|Q^T B Z - T\|_F}{\|B\|_F} \right\},$$

where (A, B) is the original matrix pair before reduction to generalized Schur form (S, T). The numerical orthogonality of the transformations has been tested by measuring

$$R_{\text{o}} = \max\left\{ \frac{\|Q^T Q - I_n\|_F}{\epsilon_{\text{mach}} n}, \frac{\|Z^T Z - I_n\|_F}{\epsilon_{\text{mach}} n} \right\}.$$

For all experiments reported in this paper, we have observed $R_{\text{r}} \in [10^{-14}, 10^{-15}]$ and $R_{\text{o}} \in [0.5, 2.5]$.

To verify that the matrix pair (S, T) returned by `PDHGEQZ` is indeed in real generalized Schur form, we have checked that the subdiagonal of S does not have two subsequent nonzero entries and that eigenvalues of two-by-two blocks correspond to a complex conjugate pair of eigenvalues. All considered matrix pairs passed this test.

3.4 Performance for random problems

We consider four different models of $n \times n$ pseudo-random matrix pairs (H, T) in Hessenberg-triangular form.

Hessrand1 For $j = 1, \ldots, n-2$, the subdiagonal entry $h_{j+1,j}$ is the square root of a chi-squared distributed random variable with $n - j$ degrees of freedom ($\sim \chi(n-j)$). For the diagonal entries of T, we choose $t_{11} \sim \chi(n)$ and $t_{jj} \sim \chi(j-1)$ for $j = 2, \ldots, n$. All other nonzero entries of H and T are normally distributed with mean zero and variance one ($\sim N(0,1)$). This corresponds to the distribution obtained when reducing a full matrix pair (A, B) with entries $\sim N(0,1)$ to Hessenberg-Triangular form; see [9] for a similar model. Such a matrix pair (H, T) has reasonably well-conditioned eigenvalues and, in turn, the QZ algorithm obeys a fairly predictable convergence behaviour.

Hessrand2 In this model, the nonzero entries of the Hessenberg matrix H and the triangular matrix T are all chosen from a uniform distribution in $[0, 1]$. The eigenvalues of such matrix pairs are notoriously ill-conditioned and lead to a more erratic convergence behaviour of the QZ algorithm.

Hessrand3 The Hessenberg matrix H is chosen as in the *Hessrand2* model, but the upper triangular matrix T is chosen as in the *Hessrand1* model. The eigenvalues tend to be less ill-conditioned compared to *Hessrand2*, but the potential ill-conditioning of T causes some eigenvalues to be identified as infinite eigenvalues by the QZ algorithm.

Infrand This model is identical to *Hessrand1*, with the notable exception that we set each diagonal element of T with probability 0.5 to zero. This yields a substantial number of infinite eigenvalues; around 1/3 of the eigenvalues are infinite.

3.4.1 Serial execution times

To verify that our parallel implementation `PDHGEQZ` also performs well on a single core, we have compared it with the serial implementations `DHGEQZ` (LAPACK version 3.4.0) and `KKQZ` (prototype implementation of serial multishift QZ algorithm with AED [24]). Figure 8 shows the timings obtained on a single core for *Hessrand1* matrix pairs. The serial performance of `PDHGEQZ` turns out to be quite good. It is even faster than `KKQZ`, although this is mainly due to the fact that `KKQZ` represents a rather preliminary implementation with little performance tuning. We expect the final version of `KKQZ` to be at least en par with `PDHGEQZ`. Similar results have been obtained for *Hessrand2*, *Hessrand3*, and *Infrand*.

3.4.2 Parallel execution time

Tables 2–4 show the parallel execution times obtained on Akka and Abisko for random matrix pairs of size $n = 4\,000$–$32\,000$.

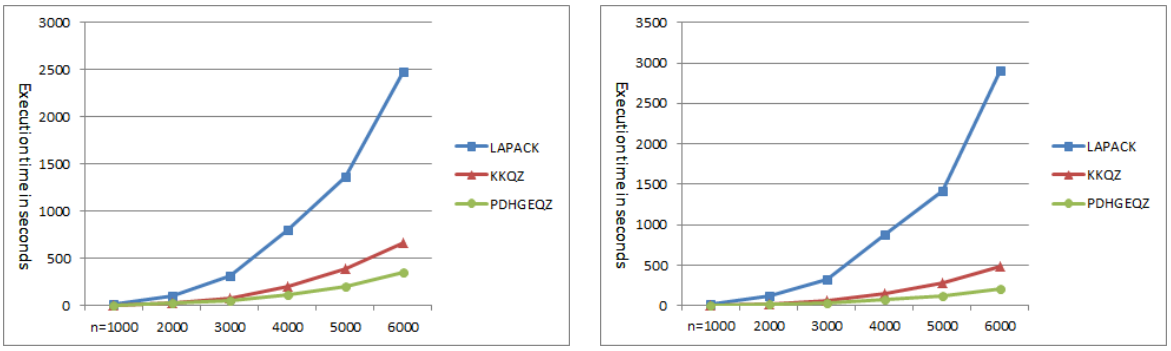

Figure 8: Serial execution times of `PDHGEQZ`, `DHGEQZ`, and `KKQZ` for *Hessrand1* matrix pairs on Akka (left figure) and Abisko (right).

The figures for the *Hessrand1* model in Table 2 indicate a parallel performance comparable to the one obtained for the parallel QR algorithm [21] on the same architecture. Analogous to the parallel QR algorithm, the execution time $T(\cdot,\cdot)$ of the parallel QZ algorithm, as a function of n and p, satisfies

$$2T(n,p) \leq T(2n,4p) < 4T(n,p), \tag{9}$$

provided that the local load per core is fixed to $n/\sqrt{p} = 4000$. This indicates that `PDHGEQZ` scales rather well, see also § 3.4.3.

As expected, the results for the *Hessrand2* model in Table 3 are somewhat erratic. Compared to the *Hessrand1* model, significantly smaller execution times are obtained for larger n, due to the greater effectiveness of AED.

Table 2: Parallel execution time (in seconds) of `PDHGEQZ` for *Hessrand1* model. Numbers within parentheses are the execution time ratios of `PDHGEQZ` over `PDHSEQR` [21] for similar random problems.

	Akka				Abisko			
$P_r \times P_c$	$n = 4k$	$n = 8k$	$n = 16k$	$n = 32k$	$n = 4k$	$n = 8k$	$n = 16k$	$n = 32k$
1×1	114(1.0)				73(1.5)			
2×2	76(1.4)	403(1.4)			37(1.8)	226(2.5)		
4×4	35(1.0)	181(1.1)	598(0.8)		23(1.8)	134(2.1)	418(1.5)	
6×6	28(0.7)	127(1.1)	432(1.0)	2188(1.0)	20(1.8)	85(2.2)	316(1.9)	1218(1.4)
8×8	25(0.7)	91(0.9)	367(1.1)	1754(1.2)	19(1.6)	72(1.9)	251(1.9)	1051(1.5)
10×10	24(0.7)	88(0.7)	298(0.9)	1486(0.9)	18(2.0)	66(1.7)	208(1.9)	919(1.8)

Finally, Table 4 reveals good parallel performance also for matrix pairs with a substantial fraction (roughly 1/3) of infinite eigenvalues. Interestingly, for $n \geq 8\,000$, the execution times for the *Infrand* model are often larger compared to the *Hessrand1* model, especially for a smaller number of processes. This effect is due to the success of AED already in the very beginning of the QZ algorithm: Deflating a converged finite eigenvalue in the bottom right corner with AED requires significantly less operations than deflating an infinite eigenvalue located far away from the top left and bottom right corners.

Table 3: Parallel execution time (in seconds) of `PDHGEQZ` for *Hessrand2* model.

	Akka				Abisko			
$P_r \times P_c$	$n = 4k$	$n = 8k$	$n = 16k$	$n = 32k$	$n = 4k$	$n = 8k$	$n = 16k$	$n = 32k$
1×1	143				46			
2×2	79	82			48	55		
4×4	45	29	102		33	22	71	
6×6	43	31	90	335	21	21	66	199
8×8	28	26	87	303	22	21	58	193
10×10	37	26	83	291	29	22	63	187

Table 4: Parallel execution time (in seconds) of `PDHGEQZ` for *Infrand* model.

	Akka				Abisko			
$P_r \times P_c$	$n = 4k$	$n = 8k$	$n = 16k$	$n = 32k$	$n = 4k$	$n = 8k$	$n = 16k$	$n = 32k$
1×1	117				67			
2×2	64	534			34	259		
4×4	30	194	1153		18	110	691	
6×6	21	126	627	3748	16	73	380	2024
8×8	19	81	441	3235	15	58	310	1636
10×10	18	74	342	2322	16	55	238	1272

3.4.3 Parallel execution time: $n = 100\,000$ benchmark

To test the performance for larger matrix pairs, we performed a few runs for $n = 100\,000$ for the *Hessrand1* and *Hessrand3* models. The obtained execution times (in seconds) on Akka and Abisko are shown in the first row of Table 5 and Table 6, respectively. Moreover, the following runtime statistics for `PDHGEQZ` are shown:

#AED	number of times AED has been performed
#sweeps	number of multishift QZ sweeps
#shifts/n	average number of shifts needed to deflate a finite eigenvalue
%AED	percentage of execution time spent on AED
#redist	number of times a redistribution of data has been performed

On Akka and for *Hessrand1*, see Table 5, we have included a comparison with the parallel QR algorithm applied to the `fullrand` model [21]. For the timings the ratios `PDHGEQZ/PDHSEQR` are listed, while absolute values are reported for the other statistics. The number for redistributions is available only for the QZ algorithm. Although the QZ algorithm is expected to performs about 3.5 more flops than the QR algorithm [18], the execution time ratios vary between 0.8 and 2.1. This is explained by a more effective AED wihtin `PDHGEQZ` for this problem type. The execution times on Abisko are substantially lower than on Akka, for both *Hessrand1* and *Hessrand3*. Even though the per core computing power is roughly the same on both machines, the network used on Abisko is faster than on Akka, and this in conjunction with using Intel-MPI instead of OpenMPI give faster execution times on Abisko for these random problems.

It turns out that only a few multishift QZ sweeps are performed for *Hessrand1*. For *Hessrand3*, the ill-conditioning makes AED even more effective and no multishift QZ sweep needs to performed.

Table 5: Execution time and statistics for `PDHGEQZ` on Akka for $n = 100\,000$. Numbers within parentheses provide a comparison to `PDHSEQR` for a similar random problem.

	$P_r \times P_c = 16 \times 16$		$P_r \times P_c = 24 \times 24$		$P_r \times P_c = 32 \times 32$		$P_r \times P_c = 40 \times 40$	
	Hessrand1	*Hessrand3*	*Hessrand1*	*Hessrand3*	*Hessrand1*	*Hessrand3*	*Hessrand1*	*Hessrand3*
Time	22093(1.6)	2725	13165(1.6)	1614	9326(1.4)	1450	7028(0.8)	2539
#AED	26(35)	17	27(31)	17	26(27)	17	25(23)	17
#sweeps	2(5)	0	2(6)	0	4(13)	0	9(12)	0
#shifts/n	0.08(0.20)	0	0.07(0.23)	0	0.11(0.35)	0	0.15(0.49)	0
%AED	83%(48%)	100%	81%(43%)	100%	78%(39%)	100%	73%(54%)	100%
#redist	1(-)	1	26(-)	17	25(-)	17	24(-)	17

Table 6: Execution time and statistics for `PDHGEQZ` on Abisko for $n = 100\,000$.

	$P_r \times P_c = 16 \times 16$		$P_r \times P_c = 24 \times 24$		$P_r \times P_c = 32 \times 32$		$P_r \times P_c = 40 \times 40$	
	Hessrand1	*Hessrand3*	*Hessrand1*	*Hessrand3*	*Hessrand1*	*Hessrand3*	*Hessrand1*	*Hessrand3*
Time	10259	1053	8031	1463	6092	1367	5680	2163
#AED	32	17	26	17	24	17	28	17
#sweeps	1	0	2	0	4	0	9	0
#shifts/n	0.04	0	0.07	0	0.11	0	0.16	0
%AED	79%	100%	74%	100%	70%	100%	66%	100%
#redist	2	1	25	17	23	17	25	17

3.5 Impact of infinite eigenvalues on performance and scalability

The purpose of this experiment is to investigate the impact of a varying fraction of infinite eigenvalues on the performance, additional to the observations already made in Table 4.

We consider $n \times n$ matrix pairs (A, B) of the form

$$A = Q \begin{bmatrix} A_{11} & 0 \\ 0 & A_{22} \end{bmatrix} Z^T, \qquad B = Q \begin{bmatrix} B_{11} & 0 \\ 0 & 0 \end{bmatrix} Z^T,$$

where $A_{11}, B_{11} \in \mathbb{R}^{(n-m)\times(n-m)}$ and $H_{22} \in \mathbb{R}^{m\times m}$ are full matrices with entries randomly chosen from a uniform distribution in $[0, 1]$. The orthogonal transformation matrices Q and Z are randomly chosen as well. Such a matrix pair will have m infinite eigenvalues of index 1 (corresponding to Jordan blocks of size 1×1).

Table 7 shows the execution time on Akka of `PDHGEQZ` applied to (A, B), after reduction to Hessenberg-triangular form. In all tests, the induced m infinite eigenvalues have been correctly identified by the parallel QZ algorithm. The fact that the execution times decrease as the number of infinite eigenvalues increases confirms the good performance of our parallel algorithm for deflating infinite eigenvalues. Regardless of how many infinite eigenvalues we have, equation 9 still holds, except when considering 40% infinite eigenvalues and comparing execution times on a 2×2 grid for $n = 4000$ with a grid of size 4×4 for $n = 8000$.

Similar observations have been made on Abisko.

3.6 Performance for benchmark examples

In the following, we present performance results for `PDHGEQZ` run on a number of different benchmarks, see Tables 9 – 16. The benchmarks are described in Table 8; most of them come from real world problems, while a few are constructed examples. The benchmarks are stored

Table 7: Execution time for PDHGEQZ on Akka for a varying fraction of infinite eigenvalues.

$P_r \times P_c$	n	10% inf	20% inf	30% inf	40% inf
1×1	4 000	88	69	56	50
2×2	4 000	38	33	26	20
4×4	4 000	31	28	21	16
6×6	4 000	24	19	16	13
8×8	4 000	20	17	14	10
2×2	8 000	235	185	153	149
4×4	8 000	140	116	100	81
6×6	8 000	101	85	67	58
8×8	8 000	82	70	59	46

in files using the Matrix Market format, see [4], and to be able to process these benchmark files effectively, specialized routines were developed that parse the files so that every process in the grid store their specific part of the globally distributed matrix pair (A, B). A and B are treated as dense in all benchmarks, although some of the samples are very sparse, and reduced to Hessenberg-Triangular form, except for the BBMSN benchmark which is stored in HT-form, before PDHGEQZ is called.

Table 8: Description of benchmark problems

Name	Size n	# ∞	Description	Ref
BBMSN	scalable	0	academic example	[24]
BCSST25	15 439	0	Columbia Center skyscraper	[4]
MHD4800	4 800	0	Alfven spectra in Magnetohydrodynamics	[4]
xingo6u, Bz1	20 738	17769	Brazilian Interconnect Power System	[23]
xingo3012, Bz2	20 944	17910	Brazilian Interconnect Power System	[23]
bips07_1998, Bz3	15 066	13046	Brazilian Interconnect Power System	[17]
bips07_2476, Bz4	16 861	14336	Brazilian Interconnect Power System	[17]
bips07_3078, Bz5	21 128	18017	Brazilian Interconnect Power System	[17]
railtrack2, RTR[1..5]	scalable	-	Palindromic quadratic eigenvalue problem	[6]
convective	11 730	0	2D convective thermal flow problems	[30]
gyro	17 361	0	butterfly gyroscope	[30]
steel1	5 177	0	heat transfer in steel profile	[30]
steel2	20 209	0	heat transfer in steel profile	[30]
supersonice	11 730	407	supersonic engine inlet	[30]
t2dal	4 257	0	2D micropyros thruster, linear FE	[30]
t2dah	11 445	0	2D micropyros thruster, quadratic FE	[30]
t3dl	20 360	0	3D micropyros thruster, linear FE	[30]
mna2	9 223	1146	modified nodal analysis	[13]
mna3	4 863	416	modified nodal analysis	[13]
mna5	10 913	123	modified nodal analysis	[13]

The first column in Table 8 shows the benchmark names and, in some cases, acronyms used in the result tables. Column two holds the problem sizes n for the benchmarks. For the benchmarks marked scalable, the problem sizes considered are marked in the result tables. Column three shows the average of the number of infinite eigenvalues identified. Columns 4 and 5 give a brief description of and reference to each benchmark.

The BBMSN benchmark, see Table 9 for execution times on Akka and Abisko, is constructed

Table 9: Execution time, in seconds, on Akka and Abisko for BBMSN.

n	$P_r \times P_c$	Akka	Abisko
5000	1×1	6	7
10000	2×2	23	18
20000	4×4	53	32

in such a way that the AED should be very successful. It turns out that the scaling is not optimal, but equation (9) still holds. Compared with the *Hessrand1* model in Table 2 we observe drastically reduced run-times, explained by the fact that no QZ sweeps are needed to compute the generalized Schur-form.

Table 10: Execution time, in seconds, on Akka and Abisko for Matrix market examples.

	Akka		Abisko	
$P_r \times P_c$	MHD4800	BCSST25	MHD4800	BCSST25
1×1	304		121	
2×2	82		35	
4×4	26	1195	19	840
6×6	23	701	16	550
8×8	24	602	16	536
10×10	23	492	16	518

BCSST25 and MDH4800 are two benchmarks with nonsingular B, i.e. only finite eigenvalues; see Table 10. For MDH4800 we observe good scaling, using a grid size up to 4×4, but the problem is too small to be solved effectively on larger grid sizes. BCSST25 is a somewhat larger problem, and `PDHGEQZ` therefore show better scaling.

Table 11: Execution time, in seconds, on Akka and Abisko for Brazilian interconnect benchmarks.

	Akka					Abisko				
$P_r \times P_c$	Bz1	Bz2	Bz3	Bz4	Bz5	Bz1	Bz2	Bz3	Bz4	Bz5
4×4	228	333				164	232			
6×6	117	165	306	339	559	96	126	220	226	234
8×8	123	125	186	192	199	64	99	143	146	160
10×10	78	102	155	157	152	61	74	113	123	120

The Brazillian interconnect power system benchmarks have a large ratio of infinite eigenvalues; see Table 11 for execution times. We observe good overall scaling properties. The initial Hessenberg-Triangular reduction moves tiny or zero elements in B up to the upper left corner, making the deflation of infinite eigenvalues particularly cheap. Table 12 reports the number of identified infinite eigenvalues for two different benchmarks and five different grid sizes. These problems have quite unbalanced A and B, leading to greater impact from rounding errors, and hence, the number of identified infinite eigenvalues will vary, even for the same grid and problem sizes if the same problem is executed more than once.

Table 12: Number of deflated infinte eigenvalues for Bz3 and Bz4 on Akka.

$P_r \times P_c$	#∞ Bz3	#∞ Bz4
4×4	13004	14305
6×6	13028	14263
8×8	13041	14333
10×10	13046	14336

Results for the scalable railtrack benchmark are reported in Table 13 for five different problem sizes and six different grid sizes, both for Akka and Abisko. These problems have a high fraction of infinite eigenvalues, which is reflected in the measured execution times. On average, `PDHGEQZ` identified $1409, 2813, 4212, 6963$, and 8320 infinite eigenvalues for RT1, RT2, RT3, RT4, and RT5, respectively.

Table 13: Execution time, in seconds, on Akka and Abisko for the railtrack benchmark. n is $5640, 8640, 11280, 16920, 19740$ for RT1, RT2, RT3, RT4, and RT5 respectively.

	Akka					Abisko				
$P_r \times P_c$	RT1	RT2	RT3	RT4	RT5	RT1	RT2	RT3	RT4	RT5
1×1	98					69				
2×2	43	70				22	36			
4×4	19	29	30	60		12	19	24	48	
6×6	15	20	25	45	58	10	15	17	38	58
8×8	13	18	23	44	67	10	12	16	33	47
10×10	12	18	20	43	63	9	12	15	30	45

All benchmarks related to the Oberwolfach test suite, except one (supersonic), have a non-singular B part. Execution times on Akka are displayed in Table 14; corresponding results on Abisko are found in Table 15. Most of these problems are rather small, and therefore `PDHGEQZ` only reveals good scaling properties for smaller grid sizes.

Table 14: Execution time, in seconds, on Akka for Oberwolfach benchmarks.

$P_r \times P_c$	t2dal	steel1	connective	t2dah	supersonic	gyro	steel2	t3dl
1×1	243	606						
2×2	108	297						
4×4	76	197	883	735	661			
6×6	73	190	616	343	328	1052	2164	2066
8×8	68	182	523	304	309	854	1847	1921
10×10	71	175	437	357	300	732	1509	1450

In Table 16, execution times are reported for the modified nodal analysis benchmark group, both for Akka and Abisko. These problems have a moderate fraction of infinite eigenvalues. Some of the problems are too small to be solved effectively on the larger grid sizes.

Table 15: Execution time, in seconds, on Abisko for Oberwolfach benchmarks.

$P_r \times P_c$	t2dal	steel1	convective	t2dah	supersonic	gyro	steel2	t3dl
1×1	117	249						
2×2	42	135						
4×4	24	70	353	315	305			
6×6	20	56	193	179	163	737	1567	1348
8×8	18	50	175	152	147	621	1293	1140
10×10	17	47	176	158	151	345	1161	1013

Table 16: Execution time, in seconds, on Akka and Abisko for modified nodal analysis benchmarks.

	Akka			Abisko		
$P_r \times P_c$	mna3	mna2	mna5	mna3	mna2	mna5
1×1	113			67		
2×2	61			30		
4×4	25	203	984	16	143	629
6×6	21	130	614	14	91	360
8×8	17	109	560	13	78	402
10×10	16	106	351	13	101	363

4 Conclusions

We have presented a new parallel algorithm and implementation `PDHGEQZ` of the multishift QZ algorithm with aggressive early deflation for reducing a matrix pair to generalized Schur form. To the best of our knowledge, this is the only parallel implementation capable of handling infinite eigenvalues. Our extensive computational experiments demonstrate robust numerical results and scalable parallel performance, comparably to what has been achieved for a recent parallel implementation of the QR algorithm [21].

There is certainly room to improve the tuning of the parameters used in `PDHGEQZ` and the interplay of the different components, including trade-offs between algorithm variants used in the implementation that targets distributed memory architectures. However, from the experience gained in the experiments, we do not expect this to lead to any further dramatic performance improvements. To boost the performance much more, the coarse-grain parallelism has to be combined with a fine-grained and strongly scalable parallel QZ algorithm that effectively manages the shared memory hierarchies of multicore nodes. Such initiatives for the parallel QR algorithm has recently started, see [28]. A critical part not addressed in this paper is the initial reduction to Hessenberg-triangular form. If this part is not handled properly, it will dominate the overall execution time. A parallel implementation of the Hessenberg-triangular reduction, based on ideas from [25], is currently in preparation.

Acknowledgments

The authors are grateful to Robert Granat, Lars Karlsson, and Meiyue Shao for helpful discussions on parallel QR and QZ algorithms. The work was supported by the Swedish Research

Council(VR) under grant A0581501, and by eSSENCE, a strategic collaborative e-Science programme funded by the Swedish Government via VR. We thank the High Performance Computing Center North (HPC2N) for providing computational resources and valuable support during test and performance runs.

A Implementation aspects

Our software is written in Fortran 90. It is a ScaLAPACK-style implementation, using BLACS for communication and ScaLAPACK/LAPACK/PBLAS/BLAS routines where appropriate.

Figure 9 shows an overview of the main routines related to our parallel QZ software. One-directed arrows indicate that one routine calls other routines. For example, `PDHGEQZ1` is called by `PDHGEQZ` and `PDHGEQZ3` and calls four routines: `PDHGEQZ2`, `PDHGEQZ4`, `PDHGEQZ5`, and `PDHGEQZ7`. Calls to LAPACK, ScaLAPACK, BLACS etc. are not show in Figure 9. Main entry routine is `PDHGEQZ`, see Figure 10 for an interface description, but the call might be directly passed on to `PDHGEQZ1` if the problem is small enough, i.e. $n \leq 6000$. `PDHGEQZ3` is responsible for doing parallel AED.

For the QZ sweeps, both `PDHGEQZ` and `PDHGEQZ1` call `PDHGEQZ5` and provide shifts, number of computational windows to chase and the number of shifts to use within each window as parameters. `PDHGEQZ5` then sets up and moves the windows until they are chased off the diagonal of (H, T).

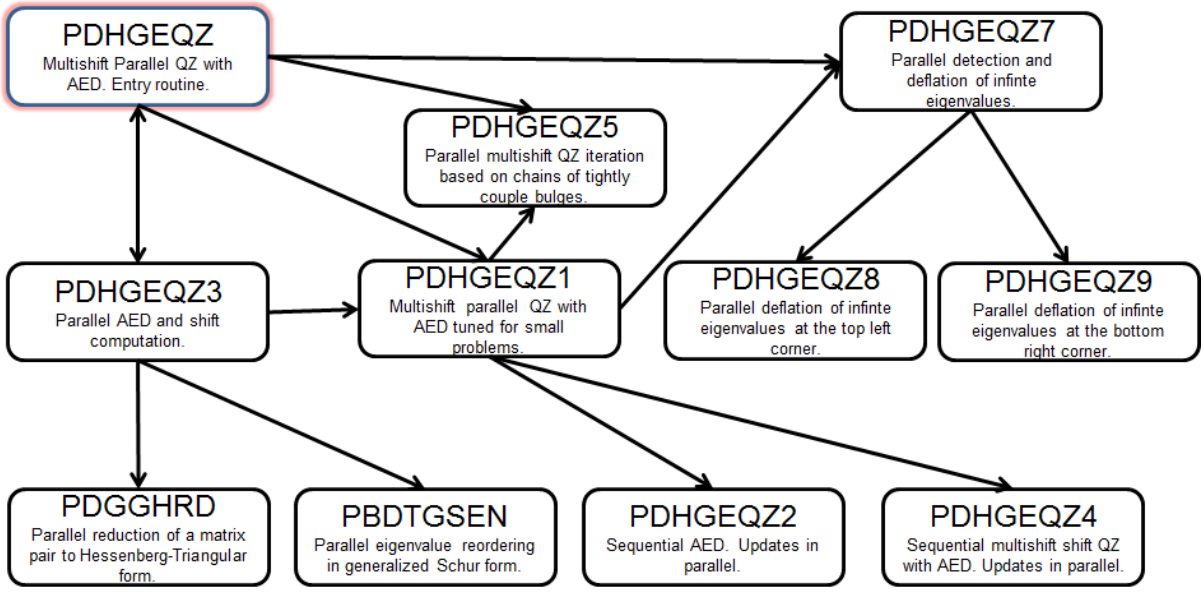

Figure 9: Software hierarchy for `PDGEHQZ`.

The interface for `PDHGEQZ` is similar to the existing (serial) LAPACK routine `DHGEQZ`, see Figure 10. The main difference between `PDHGEQZ` and `DHGEQZ` is the use of descriptors to define partitioning and globally distributed matrices across the $P_r \times P_c$ process grid, instead of leading dimensions, and that `PDHGEQZ` requires an integer workspace. `RLVL` indicates what level of recursion current execution is running in, and should initially be set to 0. We do not allow for more than two levels, that is, `PDHGEQZ` can call itself, but not more than one time. We follow the

```
      RECURSIVE SUBROUTINE PDHGEQZ(JOB, COMPQ, COMPZ,
     $       N, ILO, IHI,  A, DESCA, B, DESCB,
     $       ALPHAR, ALPHAI, BETA, Q, DESCQ, Z, DESCZ,
     $       WORK, LWORK, IWORK, LIWORK, INFO, RLVL)

*      ..
*      .. Scalar Arguments ..
*      ..
       CHARACTER          COMPQ, COMPZ, JOB
       INTEGER            IHI, ILO, INFO, LWORK, N
       INTEGER            LIWORK
       INTEGER            RLVL
*      ..
*      .. Array Arguments ..
*      ..
       DOUBLE PRECISION   A(*), B(*), Q(*), Z(*)
       DOUBLE PRECISION   ALPHAI(*), ALPHAR(*), BETA(*)
       DOUBLE PRECISION   WORK(*)
       INTEGER            IWORK(*)
       INTEGER            DESCA(9), DESCB(9)
       INTEGER            DESCQ(9), DESCZ(9)
```

Figure 10: Interface for `PDGEHQZ`

convention in LAPACK/ScaLAPACK for workspace queries, allowing -1 in `LWORK` or `LIWORK`. Optimal workspace is then returned in `WORK(1)` and `IWORK(1)`.

References

[1] B. Adlerborn, K. Dackland, and B. Kågström. Parallel and blocked algorithms for reduction of a regular matrix pair to Hessenberg-triangular and generalized Schur forms. In J. Fagerholm et al., editor, *PARA 2002*, LNCS 2367, pages 319–328. Springer-Verlag, 2002.

[2] B. Adlerborn, D. Kressner, and B. Kågström. Parallel variants of the multishift QZ algorithm with advanced deflation techniques. PARA 2006, LNCS 4699,pp. 117-126, 2006.

[3] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, third edition, 1999.

[4] Z. Bai, D. Day, J. W. Demmel, and J. J. Dongarra. A test matrix collection for non-Hermitian eigenvalue problems (release 1.0). Technical Report CS-97-355, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, March 1997. Also available online from `http://math.nist.gov/MatrixMarket`.

[5] Z. Bai, J. W. Demmel, and M. Gu. An inverse free parallel spectral divide and conquer algorithm for nonsymmetric eigenproblems. *Numer. Math.*, 76(3):279–308, 1997.

[6] T. Betcke, N. J. Higham, V. Mehrmann, C. Schröder, and F. Tisseur. NLEVP: a collection of nonlinear eigenvalue problems. *MIMS EPrint*, 2011.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. W. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[8] D. Boley. Solving the generalized eigenvalue problem on a synchronous linear processor array. *Parallel Comput.*, 3(2):153–166, 1986.

[9] K. Braman, R. Byers, and R. Mathias. The multishift QR algorithm. I. Maintaining well-focused shifts and level 3 performance. *SIAM J. Matrix Anal. Appl.*, 23(4):929–947, 2002.

[10] K. Braman, R. Byers, and R. Mathias. The multishift QR algorithm. II. Aggressive early deflation. *SIAM J. Matrix Anal. Appl.*, 23(4):948–973, 2002.

[11] A. Bunse-Gerstner and H. Faßbender. On the generalized Schur decomposition of a matrix pencil for parallel computation. *SIAM J. Sci. Statist. Comput.*, 12(4):911–939, 1991.

[12] R. Byers and D. Kressner. Structured condition numbers for invariant subspaces. *SIAM J. Matrix Anal. Appl.*, 28:326–347, 2006.

[13] Y. Chahlaoui and P. Van Dooren. A collection of benchmark examples for model reduction of linear time invariant dynamical systems. SLICOT working note 2002-2, 2002. Available online from `http://www.slicot.org`.

[14] J.-P. Charlier and P. Van Dooren. A Jacobi-like algorithm for computing the generalized Schur form of a regular pencil. *J. Comput. Appl. Math.*, 27(1-2):17–36, 1989. Reprinted in *Parallel algorithms for numerical linear algebra*, 17–36, North-Holland, Amsterdam, 1990.

[15] K. Dackland and B. Kågström. Blocked algorithms and software for reduction of a regular matrix pair to generalized Schur form. *ACM Trans. Math. Software*, 25(4):425–454, 1999.

[16] J. W. Demmel and B. Kågström. The generalized Schur decomposition of an arbitrary pencil $A - \lambda B$: robust software with error bounds and applications. II. Software and applications. *ACM Trans. Math. Software*, 19(2):175–201, 1993.

[17] F. D. Freitas, J. Rommes, and N. Martins. Gramian-based reduction method applied to large sparse power system descriptor models. *Power Systems, IEEE Transactions on*, 23:1258–1270, Aug 2008.

[18] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[19] R. Granat, B. Kågström, and D. Kressner. Parallel eigenvalue reordering in real Schur forms. *Concurrency and Computation: Practice and Experience*, 21(9):1225–1250, 2009.

[20] R. Granat, B. Kågström, and D. Kressner. A novel parallel QR algorithm for hybrid distributed memory HPC systems. *SIAM J. Sci. Comput.*, 32(4):2345–2378, 2010.

[21] R. Granat, B. Kågström, D. Kressner, and M. Shao. Parallel library software for the multishift QR algorithm with aggressive early deflation. Technical report, 2013. Available from `http://anchp.epfl.ch`.

[22] E. S. Huss-Lederman, S. Quintana-Ortí, X. Sun, and Y.-J. Y. Wu. Parallel spectral division using the matrix sign function for the generalized eigenproblem. *International Journal of High Speed Computing*, 11(1):1–14, 2000.

[23] N. Martins J. Rommes and F. Damasceno. Computing rightmost eigenvalues for small-signal stability assessment of large-scale power systems. *Power Systems, IEEE Transactions on*, 25:929 – 938, Dec 2010.

[24] B. Kågström and D. Kressner. Multishift variants of the QZ algorithm with aggressive early deflation. *SIAM J. Matrix Anal. Appl.*, 29(1):199–227, 2006. Also appeared as LAPACK working note 173.

[25] B. Kågström, D. Kressner, E. S. Quintana-Ortí, and G. Quintana-Ortí. Blocked algorithms for the reduction to Hessenberg-triangular form revisited. *BIT*, 48(3):563–584, 2008.

[26] B. Kågström and P. Poromaa. Computing eigenspaces with specified eigenvalues of a regular matrix pair (A, B) and condition estimation: theory, algorithms and software. *Numer. Algorithms*, 12(3-4):369–407, 1996.

[27] L. Karlsson and B. Kågström. Efficient reduction from block Hessenberg form to Hessenberg form using shared memory. In *Proceedings of the 10th International Conference on Applied Parallel and Scientific Computing - Volume 2*, pages 258–268. Springer-Verlag, 2012.

[28] L. Karlsson, B. Kågström, and E. Wadbro. Fine-grained bulge-chasing kernels for strongly scalable parallel qr algorithms. *Parallel Comput.*, 2013.

[29] L. Karlsson, D. Kressner, and B. Lang. Optimally packed chains of bulges in multishift QR algorithms. *ACM Trans. Math. Software*, 2014. To appear.

[30] J. G. Korvink and B. R. Evgenii. Oberwolfach benchmark collection. In P. Benner, V. Mehrmann, and D. C. Sorensen, editors, *Dimension Reduction of Large-Scale Systems*, volume 45 of *Lecture Notes in Computational Science and Engineering*, pages 311–316. Springer, Heidelberg, 2005.

[31] D. Kressner. *Numerical Methods for General and Structured Eigenvalue Problems*, volume 46 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, Berlin, 2005.

[32] A.N. Malyshev. Parallel algorithm for solving some spectral problems of linear algebra. *Linear Algebra Appl.*, 188/189:489–520, 1993.

[33] C. B. Moler and G. W. Stewart. An algorithm for generalized matrix eigenvalue problems. *SIAM J. Numer. Anal.*, 10:241–256, 1973.

[34] T. Sakurai and H. Tadano. Cirr: a Rayleigh-Ritz type method with contour integral for generalized eigenvalue problems. *Hokkaido Math. J*, 36:745–757, 2007.

[35] C. F. Van Loan. *Generalized Singular Values with Algorithms and Applications*. PhD thesis, The University of Michigan, 1973.

[36] R. C. Ward. The combination shift QZ algorithm. *SIAM J. Numer. Anal.*, 12(6):835–853, 1975.

[37] R. C. Ward. Balancing the generalized eigenvalue problem. *SIAM J. Sci. Statist. Comput.*, 2(2):141–152, 1981.

[38] D. S. Watkins. Performance of the QZ algorithm in the presence of infinite eigenvalues. *SIAM J. Matrix Anal. Appl.*, 22(2):364–375, 2000.

[39] D. S. Watkins. *The matrix eigenvalue problem.* SIAM, Philadelphia, PA, 2007.

[40] D. S. Watkins and L. Elsner. Theory of decomposition and bulge-chasing algorithms for the generalized eigenvalue problem. *SIAM J. Matrix Anal. Appl.*, 15:943–967, 1994.