Parallel Algorithms and Library Software for the Generalized Eigenvalue Problem on Distributed Memory Computer Systems

Björn Adlerborn

Licentiate Thesis

DEPARTMENT OF COMPUTING SCIENCE
UMEÅ UNIVERSITY, SWEDEN

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

adler@cs.umu.se

ISBN 978-91-7601-491-2
ISSN 0348-0542
UMINF 16.11

Printed by Print & Media, Umeå University, 2016

Abstract

We present and discuss algorithms and library software for solving the generalized non-symmetric eigenvalue problem (GNEP) on high performance computing (HPC) platforms with distributed memory. Such problems occur frequently in computational science and engineering, and our contributions make it possible to solve GNEPs fast and accurate in parallel using state-of-the-art HPC systems. A generalized eigenvalue problem corresponds to finding scalars λ and vectors x such that $Ax = \lambda Bx$, where A and B are real square matrices. A nonzero x that satisfies the GNEP equation is called an eigenvector of the ordered pair (A,B), and the scalar λ is the associated eigenvalue. Our contributions include parallel algorithms for transforming a matrix pair (A,B) to a generalized Schur form (S,T), where S is quasi upper triangular and T is upper triangular. The eigenvalues are revealed from the diagonals of S and T. Moreover, for a specified set of eigenvalues an associated pair of deflating subspaces can be computed, which typically is requested in various applications. In the first stage the matrix pair (A,B) is reduced to a Hessenberg-triangular form (H,T), where H is upper triangular with one nonzero subdiagonal and T is upper triangular, in a finite number of steps. The second stage reduces the matrix pair further to generalized Schur form (S,T) using an iterative QZ-based method. Outgoing from a one-stage method for the reduction from (A,B) to (H,T), a novel parallel algorithm is developed. In brief, a delayed update technique is applied to several partial steps, involving low level operations, before associated accumulated transformations are applied in a blocked fashion which together with a wave-front task scheduler makes the algorithm scale when running in a parallel setting. The potential presence of infinite eigenvalues makes a generalized eigenvalue problem ill-conditioned. Therefore the parallel algorithm for the second stage, reduction to (S,T) form, continuously scan for and robustly deflate infinite eigenvalues. This will reduce the impact so that they do not interfere with other real eigenvalues or are misinterpreted as real eigenvalues. In addition, our parallel iterative QZ-based algorithm makes use of multiple implicit shifts and an aggressive early deflation (AED) technique, which radically speeds up the convergence. The multi-shift strategy is based on independent chains of so called coupled bulges and computational windows which is an important source of making the algorithm scalable. The parallel algorithms have been implemented in state-of-the-art library software. The performance is demonstrated and evaluated using up to 1600 CPU cores for problems with matrices as large as 100000×100000. Our library software is described in a User Guide. The software is, optionally, tunable via a set of parameters for various thresholds and buffer sizes etc. These parameters are discussed, and recommended values are specified which should result in reasonable performance on HPC systems similar to the ones we have been running on.

Preface

The Licentiate Thesis consists of the the following three papers and an introduction including a summary of the papers.

Paper I Björn Adlerborn, Bo Kågström, and Daniel Kressner. A parallel QZ algorithm for distributed memory HPC systems[1]. In *SIAM J. Scientific Computing*, 36(5), pages 480–503. 2015.

Paper II Björn Adlerborn, Bo Kågström, and Lars Karlsson. Distributed One-Stage Hessenberg-Triangular Reduction with Wavefront Scheduling. *Report UMINF* 16.10. Dept. of Computing Science, Umeå University, Sweden, 2016 (to be submitted).

Paper III Björn Adlerborn, Bo Kågström, and Daniel Kressner. PDHGEQZ User Guide. *Report UMINF* 15.14. Dept. of Computing Science, Umeå University, Sweden, 2015.

[1] Reprinted by permission of *Society for Industrial and Applied Mathematics.*

Acknowledgements

First of all, I would like to thank my supervisors Bo Kågström and Lars Karlsson, for their great enthusiasm, inspiration, and encouragement, and for providing exceptional knowledge and support, and for always making so much of their time available.

Thanks go to Meiyue Shao, Daniel Kressner and Robert Granat for fruitful discussions on parallel QZ algorithms.

Thanks also to the colleagues and staff at the High Performance Computer Center North (HPC2N), for access to the HPC systems Abisko and Akka and for their excellent user support.

I also would like thank Anna for being there for me, making me a better person, making my life valuable, and giving birth to and being such a great mother and inspiration for our dearest daughter Emelie.

Financial support has been provided 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. This work was also partly funded from the European Unions Horizon 2020 research and innovation programme under the NLAFET grant agreement No 671633.

Umeå, May 2016

Björn Adlerborn

Contents

Chapter 1

Introduction

Solving large-scale problems efficiently and effectively on modern parallel high performance computing (HPC) platforms requires both a good parallel algorithm for the considered problem as well as very good knowledge of the underlying computer architecture. To facilitate the use of such parallel HPC systems, it is important to provide various software tools so that engineers and scientists can focus on solving their applications. With long tradition, numerical software libraries that include ready to use computational routines provide a well-functioning tool for this purpose. In this way, the burden of handling the architecture issues is mainly laid on the developers of parallel algorithms and software.

Today's parallel HPC architectures are hierarchical and are getting more and more heterogeneous in several dimensions (e.g., multicore processors, accelerators, high-speed interconnect networks). In this thesis, we focus on distributed memory architectures with multicore nodes, but common for all parallel HPC systems is that they have a complex memory hierarchy that must be utilized and given special attention in order to obtain a high portion of the theoretical peak performance. How to succeed is highly dependent on what problems to solve. Some of them lead to algorithms that are straightforward to parallelize, while many have strong dependencies between computational steps and associated data flows; the latter is the case for the dense matrix computational problems studied in this thesis.

One paramount issue concerns the management of complex memory hierarchies, which aim at avoiding unnecessary data movements between memory layers defined by on chip multi-level caches and local as well as remote memory. In practice, this means that matrix elementwise computations are restructured (or new algorithms are designed) so that blocked (submatrix) operations are used as much as possible. Ideally, if most computations can be expressed as matrix-matrix operations, it makes it possible to reuse data as much as possible at the different memory layers and thereby obtain near to optimal performance. Equally important is to balance the computational load (defined

by tasks) across all participating processes, keep them active and avoid them from going idle. To restructure and rebalance the computational load during execution, which even may include redundant computations, can be an efficient way of reducing communication and idle time.

In this thesis, we investigate and propose parallel algorithms for solving the generalized non-symmetric eigenvalue problem (GNEP)

$$Ax = \lambda Bx \qquad (x \neq 0), \tag{1.1}$$

for dense real square matrices A and B, using a two-stage method, executing on distributed memory machines. GNEPs emerge frequently, for example when solving differential-algebraic equations, in model reductions, and in the linearization of (non-linear) quadratic eigenvalue problems, and boil down to finding eigenvalues, eigenvectors and deflating subspaces of a general matrix pair (A, B). In many applications, e.g., in control system design and analysis, eigenvectors are not needed and it is enough to know a pair of deflating subspaces associated with a specified spectrum. An example is stable subspaces associated to all eigenvalues within the unit circle (or in the left complex plane).

When B is nonsingular, equation (1.1) can be transformed to a standard eigenvalue problem $Cx = \lambda Ix$ with $C = B^{-1}A$, but this is not recommended since if B is close to singular (i.e. ill-conditioned) the computation of C may affect the conditioning of other well-conditioned finite eigenvalues. Moreover, if B is a singular matrix, the GNEP has one or several infinite eigenvalues and in finite precision arithmetic there is a big risk that large finite and infinite eigenvalues are mixed up. Therefore, in practice the GNEP formulation is kept and (A, B) is treated as a matrix pair in all computations.

The two-stage method, illustrated in Figure 1 for 10×10 matrices, first reduces the matrix pair to an upper Hessenberg-triangular form (H, T), where H is an upper Hessenberg matrix (has one nonzero subdiagonal below the main diagonal) and T is an upper triangular matrix. The second stage further reduces the pair (H, T) to generalized real Schur form (S, T), where S is an upper quasi-triangular matrix, possibly with 2×2 blocks along the diagonal, and T remains upper triangular. Each 2×2 diagonal block of (S, T) corresponds to a complex conjugate pair of eigenvalues and each 1×1 block $(s_{i,i}, t_{i,i})$ corresponds to a finite eigenvalue $\lambda_i = s_{i,i}/t_{i,i}$ for $t_{i,i} \neq 0$ and to an infinite eigenvalue ∞ when $t_{i,i} = 0$. The two-stage reductions are performed using *novel parallel two-sided transformation based algorithms*, where each algorithm computes two sequences of matrix transformations that are applied to the matrix pair (A, B) from left and right, respectively.

1.1 Background

Already in 1973, Moler and Stewart [24] presented a two-stage transformation based algorithm, that follow the description above, for solving the dense generalized eigenvalue problem. In the first stage, the matrix pair (A, B) is

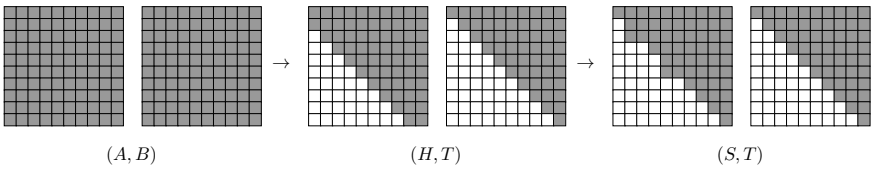

Figure 1: Reduction of a general matrix pair (A, B) to generalized real Schur form (S, T) using a two-stage method. In the first stage, (A, B) is reduced to upper Hessenberg-triangular form (H, T). In the second stage, the pair (H, T) is further reduced to (S, T) with 1×1 and 2×2 blocks along the main diagonal, corresponding to infinite or real eigenvalues and complex conjugate pair of eigenvalues, respectively.

reduced to HT form in a finite number of steps; the columns of A and B are reduced from left to right where a crucial step is to remove the undesirable fill-in caused by left and right transformations. In the second stage, the pair is further reduced to generalized real Schur form (S, T) by an iterative method, known as the *QZ algorithm*, which is a generalization and extension of the *QR algorithm*, independently proposed by Francis [15] and Kublanovskaya [23] for the standard eigenvalue problem. Throughout the years, several improvements have been proposed, and here follows a brief description of the most relevant for our study.

A cache-blocked approach was proposed by Dackland and Kågström, [12],[13], dividing the HT reduction into two separate sub-stages, first to a block HT form (H-part has several nonzero subdiagonals, algorithm is rich in matrix-matrix operations), followed by a procedure based upon Givens rotations that completes the reduction to proper upper HT form. Another one-stage cache-blocked approach using mainly level-3 BLAS, i.e. matrix-matrix operations, was proposed by Kågström, Kressner, E.S. Quintana-Ortí, and G. Quintana-Ortí [22]. The latter showed that dividing the HT reduction into two separate stages is not always better than to keep it as a single stage.

A blocked approach for the QZ algorithm, was proposed by Dackland and Kågström [14], that showed a speedup of 2–5 obtained over the unblocked algorithm. The QR algorithm, used to reduce a Hessenberg matrix to real Schur form when solving the standard eigenvalue problem, extended with improved use of several shifts and a technique for speeding up convergence, called aggressive early deflation(AED), was proposed by Braman, Byers, and Mathias [10, 11]. The multishift and AED techniques were later extended to the QZ algorithm by Kågström and Kressner [21], which greatly increased the performance when comparing with previous blocked and unblocked versions.

My master thesis, a parallel formulation for the reduction from a block HT form to proper upper HT form, together with [13] formed the first, to our knowledge, parallel reduction of a general matrix pair (A, B) to HT form,

see [1]. This parallel two-staged HT reduction approach shows that although the first stage involves much more flops than the second stage, the latter dominates the overall execution time. However, the second stage scales somewhat better than the first stage.

In [2, 4], we propose a parallel solution for the complete reduction of a general matrix pair to generalized Schur form. The parallel QZ reduction stage is based on and extend [14] but, despite having a more favorable flop count, it dominates the total execution time, leaving room for improvements. However, this implementation is significantly faster than the corresponding parallel QR implementation [20], despite that the flop count for QZ is about two times more than for QR. This is explained by the preliminary use of AED, efficient multishift strategies and accumulated updates applied in a matrix-matrix manner. However, the parallel QR algorithm did later undergo a complete revision, adopting the new efficient multishift strategies and AED, resulting in an efficient parallel solver on (hybrid) distributed memory machines, see [18, 19].

The Papers I–III, in this thesis, present further novel contributions (algorithms and software) to the parallel solution of the generalized eigenvalue problem.

1.2 Data distribution

The target parallel platform is distributed memory machines, where each process has it own set of memory, and the problem is split among the participating processes. However, in practice and in most modern HPC systems, the processes are grouped into compute units which have some shared memory, leading to so called hybrid distributed memory machines. The programming model used in this thesis is message passing and any available shared memory at the CPU nodes is treated and allocated as separate memory units to each process.

Using a two-dimensional block-cyclic data distribution schema of the matrices A and B ensures that each process has a part, of roughly the same size, of the problem data and the computational workload. The P processes are logically arranged into to a $P_\mathrm{r} \times P_\mathrm{c}$ grid, not necessarily square. Moreover, the $N \times N$ matrices A and B are partitioned into $N_\mathrm{b} \times N_\mathrm{b}$ sized blocks, and scattered cyclically across the $P_\mathrm{r} \times P_\mathrm{c}$ grid, see Figure 2 for an illustration.

ScaLAPACK [9], a state-of-the-art library of high-performance linear algebra routines for parallel distributed memory machines, uses the two-dimensional block cyclic distribution schema, generalized such that the data blocks need not be square, where each distributed object is represented by two objects—a pointer to the local data and a globally defined descriptor to define the partitioning and a communication context.

ScaLAPACK is based upon, and includes, the software package BLACS for handling communication and offers point-to-point non blocking send and blocking receive, as well as broadcast send and receive and global summation

(0,0)	(0,1)	(0,2)	(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)	(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)	(2,0)	(2,1)	(2,2)
(0,0)	(0,1)	(0,2)	(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)	(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)	(2,0)	(2,1)	(2,2)

Figure 2: Block cyclic data distribution of a matrix exemplified using a 3×3 grid, where the residence for each of the 36 matrix blocks is given by its process coordinate (p_r, p_c) with $0 \leq p_r < 3 = P_r$ and $0 \leq p_c < 3 = P_c$.

routines. Our algorithms and software are designed to fit into the ScaLAPACK suite of routines, and is therefore restricted to use the communication primitives offered by BLACS. The lack of a non blocking receive limits the degree of algorithmic freedom a bit, but in general, communication via BLACS and operating on distributed objects in ScaLAPACK is straightforward.

Operations on distributed data typically requires communication, for example, when one process holds part of data that other processes need to complete an operation. In order to reduce communication, the data can be reorganized, temporarily, to a subgrid before the computation is performed. Using less processes, means that each process that participates in the computation, will own more data, and potentially reduce the need for communication. After completing the computation, data is typically restored to the original grid to continue using all allocated processes for the remaining computation. ScaLAPACK provides a redistribution routine, and is used in our software, for example, to perform parallel AED, where the AED computational window is spread over several processes. Our heuristics show that it is beneficial to use less computational power by performing the AED computation on a subgrid. The redistribution routine is efficient, but should be used with caution as it increases the memory load, and de facto decreases the core utilization leading to poor scalability in the long run.

1.3 Memory hierarchies and operations

The memory hierarchy of a typical CPU core consists of registers, levels of caches and main memory. However, all computations are performed at the

very top of the hierarchy, i.e. in the registers, and an operation on data stored in the main memory requires data to be transferred and copied all the way up to the top. Once stored at the top, we should try to utilize this data as much as possible before copying another set of data to work on, as this copy procedure is time consuming; each memory level has a latency and a per item cost associated to a memory transfer. Put into practice, (cache-)blocked code is used, where the software is written in such a way that it works on optimized sized chunks of data, performing several operations on one chunk at a time, before the next chunk is addressed. As there are often several layers of cache memories, of different sizes, several layers of blocked code are also common. Instead of performing several elementwise operations, operations can often be bundled and applied in an accumulated way. A simple example is a list of numbers added to each element of a vector; instead of adding numbers individually to the vector, compute the sum once, and then add the sum to the elements of the vector. Another example is the usage of accumulated Givens rotations. A Givens rotation is an orthogonal matrix $G \in \mathbb{R}^{n \times n}$ that applied to another matrix makes a rotation with angle θ in the plane (i, j) spanned by two coordinates axes (here $j = i + 1$):

$$G_{ij}(\theta) = \left[\begin{array}{c|cc|c} I_{i-1} & & & \\ \hline & c & s & \\ & -s & c & \\ \hline & & & I_{n-j} \end{array}\right],$$

where $c^2 + s^2 = 1$ ($c = cos(\theta), s = sin(\theta)$) and I is the identity matrix of size $(i-1) \times (i-1)$ and $(n-j) \times (n-j)$, respectively. The main use of Givens rotations in numerical linear algebra is to zero out, *annihilate*, elements in vectors or matrices: given a, b find c and s such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix},$$

and $r > 0$ does not over- or underflow. We use this annihilation technique to reduce a matrix pair (A, B) to HT-form, where several Givens rotations are bundled, i.e. accumulated into one matrix and subsequently applied to blocks in (A, B). This bundling process enables more coarse-grained matrix-matrix operations which greatly improve the arithmetic performance compared to applying the rotations one by one.

1.4 Redundant computing

Increasing the amount of arithmetic work and let a few processes repeat and do the same work can often be beneficial from a total execution time point of view. Consider the matrix operation

$$U \cdot V,$$

where $U \in \mathbb{R}^{m \times m}, V \in \mathbb{R}^{m \times n}$. Partition $V = \begin{bmatrix} V_0 \\ V_1 \end{bmatrix}$ and let it be distributed over a 2×1 process grid (most likely a subgrid), such that process p_0 holds V_0 and process p_1 holds V_1. We assume that U is stored on both p_0 and p_1.

To perform the operation, p_0 and p_1 need to exchange data. One approach is to let p_0 receive V_1 from p_1, perform the matrix multiplication, and then send updated V_1 back. p_1 will be idle during the computation, waiting for the updated V_1. In a non blocking receive communication environment, p_1 could however perform other tasks, that are independent of V_1. Another approach is to let p_0 and p_1 exchange data such that both have enough to perform the complete operation. This requires some extra workspace, but the upside is that the exchange can be performed, almost perfect, in parallel; both perform non blocking send of their parts of V, and enter the receive mode to receive the data, which is already on the way. Both compute $U \cdot V$, but only save the part of the product they own, that is p_1 discards updated V_0, and vice versa for p_0. This technique requires extra work but has one synchronization point less and reduce idling processes, and is successfully used in our two-stage reduction of a matrix pair (A, B) to generalized Schur form, for example, when applying bundled Givens operations.

Chapter 2

Summary of papers

In the following, a brief summary of each paper in the thesis is given.

2.1 Paper I

Paper I [5] concerns the parallel reduction of a matrix pair in Hessenberg, triangular form (H, T) to generalized real Schur form (S, T). The paper begins with an overview of the generalized Schur decomposition and the structure of the QZ algorithm before moving on to discussing the multishift and AED techniques, with focus on aspects related to the parallel algorithms and implementations. The potential presence of infinite eigenvalues in the generalized eigenvalue problem makes a fundamental difference compared to the standard one. Infinite eigenvalues need to be dealt with, i.e. identified and deflated, before other actions are taken in order to preserve them as infinite, or not having them inflicting damage to other eigenvalues, due to round off errors. A serial and a novel parallel algorithm are discussed and exemplified where the infinite eigenvalues are moved to the top-left or bottom-right corner of the matrix pair, whichever is nearest.

Performance is evaluated using several different problems, on two different parallel HPC systems. Interesting and applicable real world benchmark examples from Matrix Market [8], some with a large fraction of infinite eigenvalues, together with constructed problems of three different types demonstrate execution times for different grid configurations. The problem size n ranges from 4000 to 32000, and up to 100 cores are utilized to solve the problems in parallel, demonstrating increasing speedup as the problem size and number of cores increases. These problems are however rather small in a context of what modern HPC systems are capable of, so in order to utilize more compute power, a constructed 100000×100000 benchmark problem is solved in parallel using up to 1600 cores, and performance is evaluated and compared with the parallel solver for standard eigenvalue problems [19] for a similar problem. Even

though the standard eigenvalue problem requires less than half of the number of operations to complete compared to the generalized eigenvalue problem [17], execution time ratios show that our solver takes substantially less than twice the time to complete.

2.2 Paper II

Paper II [3] concerns the parallel reduction of a general matrix pair (A, B) to Hessenberg-triangular form (H, T). Based on the sequential cache-blocked algorithm [22], this parallel formulation makes use of Givens rotations to reduce the matrix pair with a novel static wavefront scheduling algorithm. The sequential algorithm and its blocking strategy are briefly described, before moving on to a discussion on how different parts of the algorithm have been redesigned to work in parallel. Since a straightforward parallelization strategy proves to be poorly scalable, due to a high fraction of idle time among the participating processors, a scheduler is implemented with the aim to maximize process utilization and execute the shortest possible sequence of parallel steps. At each step, the scheduler select a parallel task to execute such that

- the degree of parallelism is maximized,
- tasks with more remaining work is chosen over tasks with less work.

Two different HPC systems are used to evaluate the parallel performance; weak and strong scaling is measured, visualized and discussed. Results, using up to 961 mpi-processes, indicate that our implementation scales but suffers from bottlenecks, related to synchronization points, in two of its major subroutines.

2.3 Paper III

Paper III [6] is a User Guide for the `PDHGEQZ` software; library software routines to solve the generalized eigenvalue problem for dense and real matrix pairs (A, B) in parallel on multicore HPC systems. The guide mainly describes software and parameters related to Paper I, but also includes a description of routines related to Paper II and routines from other earlier work. Installation and building instructions, for a Linux like system, are presented, followed by a software hierarchy overview of how routines are related and called.

The calling sequences for the main driver routines with input and output parameters are described in detail. Moreover, the set of tunable parameters and a description of their usage and default values are discussed. The default value for parameters may need tuning to reach the best possible performance of the `PDHGEQZ` software executing on a new target architecture, however, the defaults should give reasonable performance on systems similar to the ones we have been running on.

During the build process, internal tests are performed to make sure the software work as intended. Both sequential and parallel tests are performed, with validation of the computed results. System software requirements are listed so users can prepare their systems before the build process and installation is initiated.

Chapter 3

Future work

Our solution to the generalized eigenvalue problem provides eigenvalues and deflating subspaces, but presently does not compute eigenvectors. Given a matrix pair in generalized Schur form, LAPACK [7] offers serial routines for computing both left and right eigenvectors. Combining those with the accelerating techniques proposed by Gates et al. in [16] and our own experiences will be a good base for formulating a parallel distributed memory algorithm.

Our parallel algorithms for computing the generalized Schur form scale with the number of processors, but there is room for improvements. A great challenge is to develop novel architecture-aware algorithms that expose as much parallelism as possible in today's and future extreme-scale HPC systems. This and many other challenges will be investigated within the Horizon 2020 project *Parallel Numerical Linear Algebra for Future Extreme-Scale Systems* with acronym NLAFET, coordinated by Umeå University. The NLAFET overall aim is to enable a radical improvement in the performance and scalability of a wide range of real-world applications relying on linear algebra software for future extreme-scale systems. For more information see the NLAFET website: `http://www.nlafet.eu`

Bibliography

[1] B. Adlerborn, K. Dackland, and B. Kågström. Parallel two-stage reduction of a regular matrix pair to Hessenberg-Triangular form. In T. Sørevik, F. Manne, A. H. Gebremedhin, and R. Moe, editors, *Applied Parallel Computing, PARA 2000*, LNCS 1947, pages 92–102. Springer Berlin Heidelberg, 2000.

[2] 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, J. Haataja, J. Järvinen, M. Lyly, P. Råback, and V. Savolainen, editors, *Applied Parallel Computing, PARA 2002*, LNCS 2367, pages 319–328. Springer-Verlag, 2002.

[3] B. Adlerborn, L. Karlsson, and B. Kågström. Distributed One-Stage Hessenberg-Triangular Reduction with Wavefront Scheduling. *Report UMINF* 16.10, Dept. of Computing Science, Umeå University, Sweden, 2016.

[4] B. Adlerborn, B. Kågström, and D. Kressner. Parallel variants of the multishift QZ algorithm with advanced deflation techniques. In B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, editors, *Applied Parallel Computing, PARA 2006*, LNCS 4699, pages 117–126. Springer Berlin Heidelberg, 2006.

[5] B. Adlerborn, B. Kågström, and D. Kressner. A Parallel QZ Algorithm for distributed memory HPC-systems. *SIAM J. Sci. Comput.*, 36(5):C480–C503, 2014.

[6] B. Adlerborn, B. Kågström, and D. Kressner. PDHGEQZ User Guide. *Report UMINF* 15.12, Dept. of Computing Science, Umeå University, Sweden, 2015.

[7] 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.

[8] 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`.

[9] 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.

[10] 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.

[11] 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.

[12] K. Dackland and B. Kågström. Reduction of a Regular Matrix Pair (A, B) to Block Hessenberg Triangular Form. In J. Dongarra, K. Madsen, and J. Waśniewski, editors, *Applied Parallel Computing, PARA 1995*, LNCS 1041, pages 125–133. Springer Berlin Heidelberg, 1995.

[13] K. Dackland and B. Kågström. A ScaLAPACK-Style Algorithm for Reducing a Regular Matrix Pair to Block Hessenberg-Triangular Form. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Applied Parallel Computing, PARA 1998*, LNCS 1541, pages 95–103. Springer Berlin Heidelberg, 1998.

[14] 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.

[15] J. G. F. Francis. The QR Transformation. A Unitary Analogue to the LR Transformation - Part 1. *The Computer Journal*, 4(3):265–271, 1961.

[16] M. Gates, A. Haidar, and J. Dongarra. Accelerating computation of eigenvectors in the dense nonsymmetric eigenvalue problem. In M. Daydé, O. Marques, and K. Nakajima, editors, *High Performance Computing for Computational Science, VECPAR 2014*, LNCS 8969, pages 182–191. Springer International Publishing, 2015.

[17] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 4th edition, 2012.

[18] 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.

[19] R. Granat, B. Kågström, D. Kressner, and M. Shao. Parallel library software for the multishift QR algorithm with aggressive early deflation. *ACM Trans. Math. Software*, 41(4), 2015.

[20] G. Henry, D. S. Watkins, and J. J. Dongarra. A parallel implementation of the nonsymmetric QR algorithm for distributed memory architectures. *SIAM J. Sci. Comput.*, 24(1):284–311, 2002.

[21] 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.

[22] 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.

[23] V.N. Kublanovskaya. On some algorithms for the solution of the complete eigenvalue problem. *USSR Computational Mathematics and Mathematical Physics*, 1(3):637 – 657, 1962.

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

I

Paper I

A parallel QZ algorithm for distributed memory HPC systems*

Björn Adlerborn, Bo Kågström, and Daniel Kressner

Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden
{adler,bokg}@cs.umu.se
SB–MATHICSE–ANCHP, EPF Lausanne, Station 8, CH-1015 Lausanne, Switzerland
daniel.kressner@epfl.ch

Abstract: Appearing frequently in applications, generalized eigenvalue problems represent one of the core problems in numerical linear algebra. The QZ algorithm of 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.

A parallel QZ algorithm for distributed memory HPC systems

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

Abstract

Appearing frequently in applications, generalized eigenvalue problems represent one of the core problems in numerical linear algebra. The QZ algorithm of 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 [32] is the most widely used algorithm for addressing generalized eigenvalue problems with dense matrices. Since then, it has undergone several modifications [14, 34]. In particular, significant speedups on serial machines have been obtained in [23] 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 [19, 20] 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.

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, 13] as well as spectral divide-and-conquer algorithms [5, 21, 31].

[†]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)

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 or applying the GUPTRI algorithm [15].

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*

$$Q^T A Z = S, \qquad Q^T B Z = T, \tag{1}$$

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 [33].

1.2 Structure of the QZ algorithm

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

$$Q^T A Z = H, \qquad Q^T B Z = T, \tag{2}$$

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 [32] uses Givens rotations to zero out each entry below the subdiagonal of A. Its memory access pattern causes this algorithm perform rather poorly on standard computing architectures. To address this, a blocked two-stage approach has been proposed by Dackland and Kågström [14]. 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 [26]. 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 [24] 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 [35] 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 us 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 [25] and its parallelization is discussed in [18].

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 [23].

2 Parallel algorithms

In what follows, we assume that the reader is familiar with the basics of the implicit shifted QZ algorithm, see [30, 37] 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 (generically nonzero) 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 [38] and shown to be numerically backward stable in [23]. Let Z_0 be a Householder reflector that maps $T^{-1}e_1$ to a multiple of the first unit vector: $Z_0^T T^{-1} e_1 = \gamma e_1$ for some $\gamma \neq 0$. Then $T Z_0 e_1 = \gamma^{-1} e_1$, showing that applying Z_0 from the right, which affects the first three columns only, results in the nonzero pattern

$$
H \leftarrow H Z_0 = \begin{bmatrix}
\hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\
\hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\
\hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\
\hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\
0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\
0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\
0 & 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots &
\end{bmatrix}, \quad
T \leftarrow T Z_0 = \begin{bmatrix}
\hat{\mathsf{X}} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\
\hat{0} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\
\hat{0} & \hat{\mathsf{X}} & \hat{\mathsf{X}} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\
0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\
0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \mathsf{X} & \cdots \\
0 & 0 & 0 & 0 & 0 & \mathsf{X} & \mathsf{X} & \cdots \\
0 & 0 & 0 & 0 & 0 & 0 & \mathsf{X} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots &
\end{bmatrix}.
$$

Here, and in the following, $\hat{0}$ denotes a zero entry newly introduced by the current transformation. 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 [36]. 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:

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

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

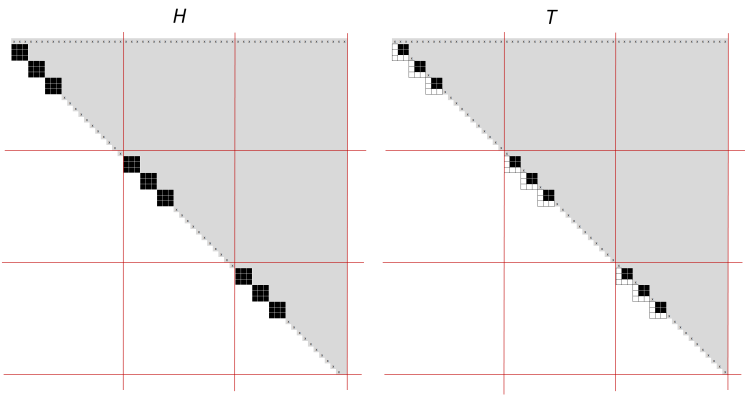

Figure 1: Three bulge chains, where each chain consists of three bulges (black boxes). The black 3×3 blocks in the H-part are dense, while the associated blocks in the T-part have zero elements in the last row and first column, respectively. Only parts of the matrices are displayed. The solid red lines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

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 [19]. 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 [19] 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 [19] 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 (`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

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

[1] In fact, as recently shown in [28] for the QR algorithm, it is possible and beneficial to pack the bulges even closer.

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 two smaller subproblems. Such deflations typically occur at the bottom right corner or, less frequently, at the top left corner. In both cases, one of the subproblems is tiny and can be solved instantaneously. When a deflation occurs in the middle it would, in principle, be advantageous in a parallel environment to solve both subproblems simultaneously. In practice, however, this event seems to be too rare to justify the resulting higher complexity of the implementation. In our implementation, the two subproblems are solved subsequently.

Aggressive early deflation (AED) is a technique proposed by Braman, Byers and Mathias [10] for the QR algorithm, that complements (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 [23]. 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 [20], 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 $n_{\mathsf{AED}} \leq 201$, does it make 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 [18] 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 [19, 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 [20] 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 $P_{\mathsf{AED}} \times P_{\mathsf{AED}}$ 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 us to deflate an infinite eigenvalue. In the absence of roundoff error, this reordering is automatically effected by QZ iterations [36]. 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 [32]. 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:

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

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

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

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

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

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.

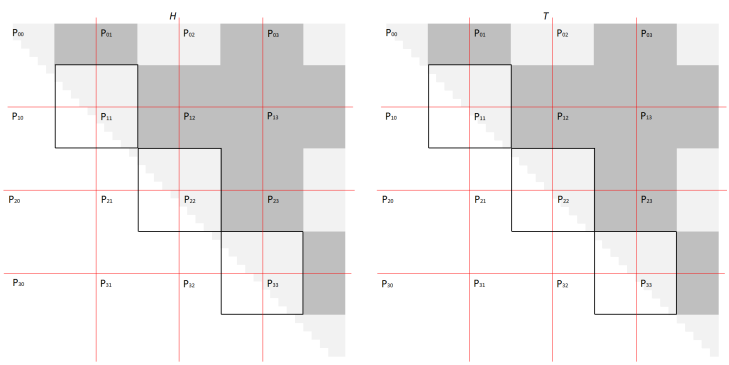

Figure 2: Crossborder layout of three windows during the parallel deflation of infinite eigenvalues. Computational windows and process borders indicated by black and red solid lines, respectively. Darker-gray areas indicate regions that need to be updated after the windows have been processed.

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 [19], 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 repetition of 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.

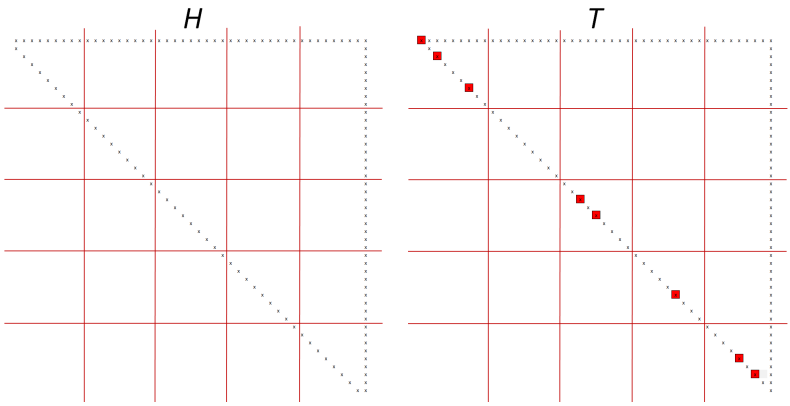

Figure 3: Example where 8 zeros have been identified on the diagonal of T, indicated by filled red squares. The grid is of size 5×5 and process borders are indicated by red solid lines.

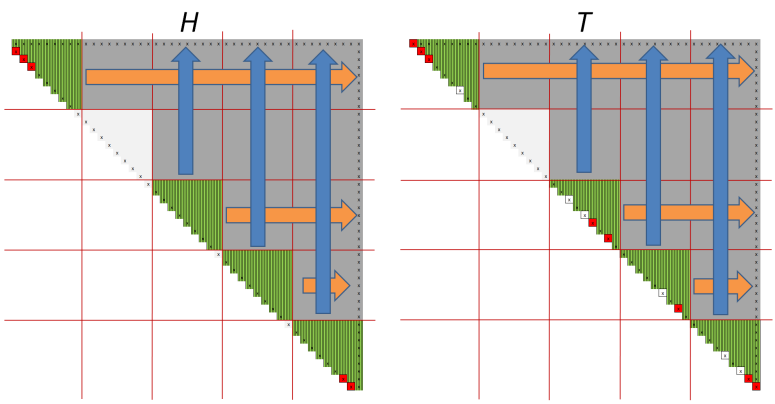

Figure 4: *Intra-block* chase where all diagonal blocks are processed in parallel. The four active diagonal blocks, within which the zero diagonal entries of T are moved up or down, are marked in green with vertical stripes. The old positions of the zero diagonal entries are marked by white squares, 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 gray.

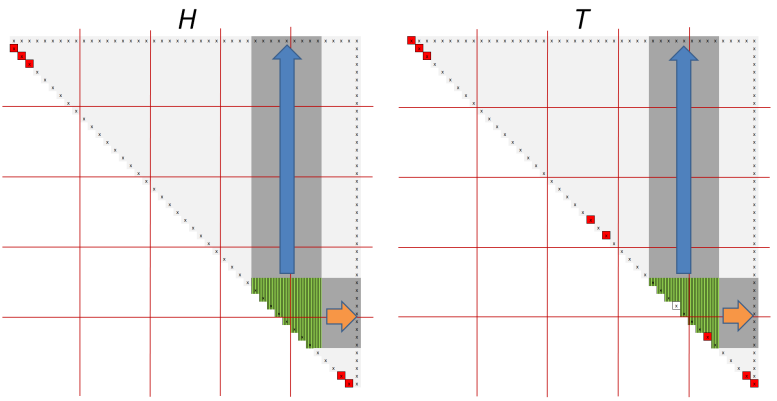

Figure 5: *Inter-block* chase, phase one, where all even-numbered diagonal blocks, indexed by $0 \ldots n_{\text{block}} - 1$ from bottom to top, are processed in parallel. In this case there is only one active diagonal block. See Figure 4 for an explanation of the color marking.

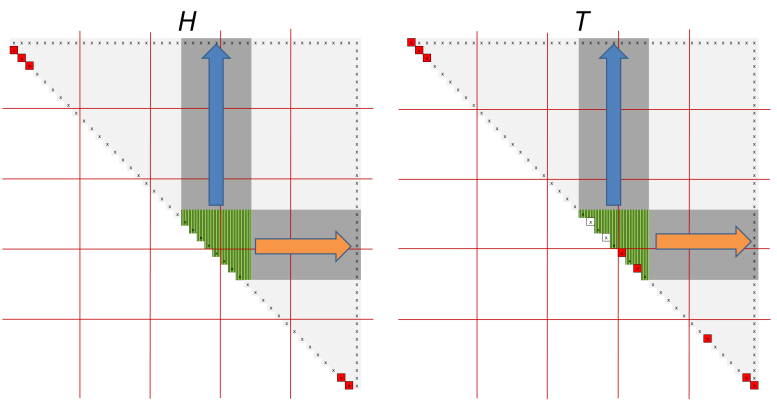

Figure 6: *Inter-block* chase, phase two, where all odd-numbered diagonal blocks are processed in parallel. See Figure 4 for an explanation of the color marking.

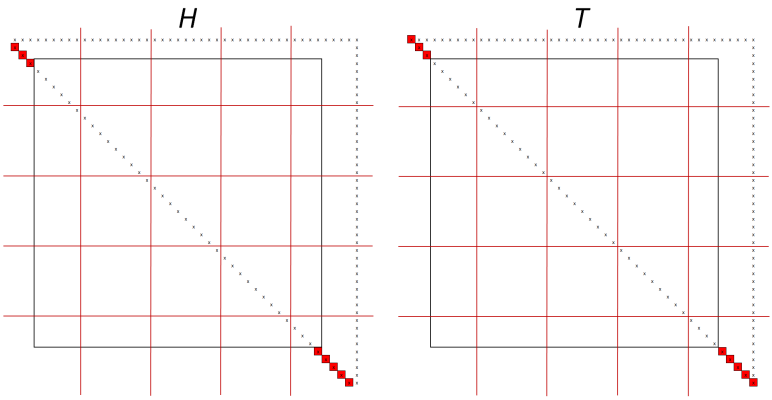

Figure 7: *Intra-block* and *inter-block* chases are repeated until all zeros on the diagonal of T have been moved to the top left or bottom right corners. A total of 8 deflations can be made. The remaining problem size is decreased accordingly, indicated by the black square on H and T.

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 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 [19, 20] 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 [20]. 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 [20]. 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 behavior.

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 behavior 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 [23]). 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 $n = 4\,000$–$32\,000$.

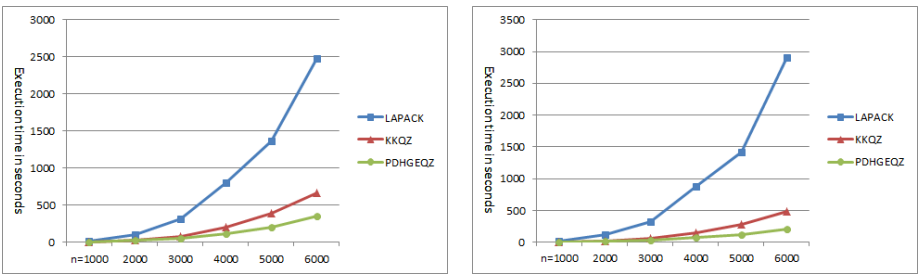

Figure 8: Serial execution times of `PDHGEQZ`, `DHGEQZ`, and `KKQZ` for *Hessrand1* matrix pairs on Akka (left figure) and Abisko (right).

The figures for the *Hessrand1* model in Table 2 indicate a parallel performance comparable to the one obtained for the parallel QR algorithm [20] 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` [20] 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 [20]. 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 perform about 3.5 times more flops than the QR algorithm [17], the execution time ratios vary between 0.8 and 2.1. This is explained by a more effective AED within `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 $A_{22} \in \mathbb{R}^{m\times m}$ 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	[23]
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	[22]
xingo3012, Bz2	20 944	17910	Brazilian Interconnect Power System	[22]
bips07_1998, Bz3	15 066	13046	Brazilian Interconnect Power System	[16]
bips07_2476, Bz4	16 861	14336	Brazilian Interconnect Power System	[16]
bips07_3078, Bz5	21 128	18017	Brazilian Interconnect Power System	[16]
railtrack2, RTR[1..5]	scalable	-	Palindromic quadratic eigenvalue problem	[6]
convective	11 730	0	2D convective thermal flow problems	[29]
gyro	17 361	0	butterfly gyroscope	[29]
steel1	5 177	0	heat transfer in steel profile	[29]
steel2	20 209	0	heat transfer in steel profile	[29]
supersonic	11 730	407	supersonic engine inlet	[29]
t2dal	4 257	0	2D micropyros thruster, linear FE	[29]
t2dah	11 445	0	2D micropyros thruster, quadratic FE	[29]
t3dl	20 360	0	3D micropyros thruster, linear FE	[29]
mna2	9 223	1146	modified nodal analysis	[12]
mna3	4 863	416	modified nodal analysis	[12]
mna5	10 913	123	modified nodal analysis	[12]

The first column in Table 8 shows the benchmark names and, in some cases, the corresponding 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 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 Brazilian 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 infinite 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 [20].

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 [27]. 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 [24], 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 shown 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 performing parallel AED.

For the QZ sweeps, both `PDHGEQZ` and `PDHGEQZ1` call `PDHGEQZ5` and provide shifts, the 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).

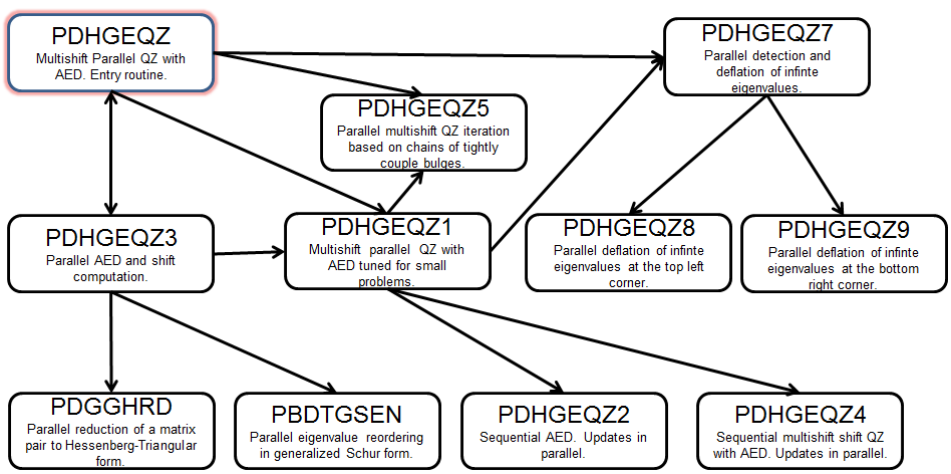

Figure 9: Software hierarchy for `PDGEHQZ`.

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. *ACM Trans. Math. Software*, 39(2):7:1–7:28, February 2013.

[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] 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`.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] R. Granat, B. Kågström, D. Kressner, and M. Shao. Parallel library software for the multishift QR algorithm with aggressive early deflation. Technical report, 2013. Revised June 2014. Available from `http://anchp.epfl.ch`.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] L. Karlsson, B. Kågström, and E. Wadbro. Fine-grained bulge-chasing kernels for strongly scalable parallel QR algorithms. *Parallel Comput.*, 2014. To appear.

[28] L. Karlsson, D. Kressner, and B. Lang. Optimally packed chains of bulges in multishift QR algorithms. *ACM Trans. Math. Software*, 40(2):12:1–12:15, 2014.

[29] 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.

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

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

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

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

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

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

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

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

[38] 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.

II

Paper II

Distributed One-Stage Hessenberg-Triangular Reduction with Wavefront Scheduling

Björn Adlerborn, Bo Kågström, and Lars Karlsson

Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden
{adler,bokg,larsk}@cs.umu.se

Abstract: A novel parallel formulation of Hessenberg-triangular reduction of a regular matrix pair on distributed memory computers is presented. The formulation is based on a sequential cache-blocked algorithm by Kågström, Kressner, E.S. Quintana-Ortí, and G. Quintana-Ortí (2008). A static scheduling algorithm is proposed that addresses the problem of underutilized processes caused by two-sided updates of matrix pairs based on sequences of rotations. Experiments using up to 961 processes demonstrate that the new algorithm is an improvement of the state of the art but also identifies factors that currently limit its scalability.

Distributed One-Stage Hessenberg-Triangular Reduction with Wavefront Scheduling*

Björn Adlerborn, Lars Karlsson, and Bo Kågström

Abstract

A novel parallel formulation of Hessenberg-triangular reduction of a regular matrix pair on distributed memory computers is presented. The formulation is based on a sequential cache-blocked algorithm by Kågström, Kressner, E.S. Quintana-Ortí, and G. Quintana-Ortí (2008). A static scheduling algorithm is proposed that addresses the problem of underutilized processes caused by two-sided updates of matrix pairs based on sequences of rotations. Experiments using up to 961 processes demonstrate that the new formulation is an improvement of the state of the art and also identify factors that limit its scalability.

1 Introduction

For any matrix pair (A, B), where $A, B \in \mathbb{R}^{n\times n}$, there exist orthogonal matrices $Q, Z \in \mathbb{R}^{n\times n}$, not necessarily unique, such that $Q^T AZ = H$ is upper Hessenberg and $Q^T BZ = T$ is upper triangular. The resulting pair (H, T) is said to be in *Hessenberg-Triangular (HT) form* and the act of reducing (A, B) to (H, T) is referred as *HT reduction*. One application of HT reduction is as a preprocessing step used in various numerical methods such as the QZ algorithm for the non-symmetric generalized eigenvalue problem [4, 3, 9, 14, 10].

Moler and Stewart [13] proposed in 1973 an algorithm for HT reduction that is exclusively based on Givens rotations. Kågström, Kressner, E.S. Quintana-Ortí, and G. Quintana-Ortí [11] proposed in 2008 a cache-blocked variant of Moler and Stewart's algorithm. They express most of the arithmetic operations in terms of matrix–matrix multiplications involving small orthogonal matrices obtained by explicitly accumulating groups of rotations using a technique proposed by Lang [12]. Both of these algorithms are sequential, but the cache-blocked algorithm can to a limited extent scale on systems with shared memory by using a parallel matrix–matrix multiplication routine.

The algorithms above use a *one-stage approach* in the sense that they reduce the matrix pair directly to HT form. There is also a *two-stage approach* that first reduces the matrix pair to a block HT form followed by a bulge-chasing procedure that completes the reduction to proper HT form [2, 1, 8]. Two-stage algorithms are arguably more complicated and have a higher arithmetic cost. A fundamental difference between the *one-stage* and the *two-stage* approach is the usage of Householder reflections in the *two-stage* approach. Householder reflections are used in the first stage to reduce column entries in the matrix A. When these are applied to the matrix B, they will, repeatedly, destroy the structure by producing fill-in in the lower triangular part of the matrix B and hence cause extra work. However, the

Report UMINF 16.10, Dept. of Computing Science, Umeå University, Sweden

arithmetic and parallel gain from using long reflections and applying them in a parallel, blocked manner is greater than the extra work of restoring B to its original triangular form. At some point, the extra work becomes substantial, so the algorithm shifts to the second stage, at some heuristically determined breakpoint, to instead use a bulge-chasing procedure causing less fill-in in B. Householder reflections are also used to reduce a single matrix to Hessenberg form, used as a preprocessing step in the standard eigenvalue problem, and since there is no fill-in to be dealt with, Householder reflections are used exclusively, resulting in a highly efficient reduction where 80% [5, 6] of the operations are performed in level 3 BLAS matrix-matrix operations.

For the generalized case, recent results show that a sequential cache-blocked one-stage approach can outperform or at least compete with a two-stage approach [11]. In this paper, we propose a novel parallel formulation of a cache-blocked one-stage algorithm [11, Algorithm 3.2] hereafter referred to as `KKQQ` after its inventors.

The algorithms mentioned above have in common that they first reduce the matrix B to upper triangular form using a standard QR factorization. Specifically, a QR factorization $B = Q_0 R$ is computed, then B is overwritten by R, Q is set to Q_0, and A is overwritten by $Q_0^T A$. Since these steps are common to all HT reduction algorithms and parallel formulations for them are well understood [7], we assume from now on that the input matrix pair (A, B) already has B in upper triangular form.

The remainder of the paper is organized as follows. Notation and terminology are described in Section 2. The sequential `KKQQ` algorithm is recalled in Section 3. An overview of the new parallel formulation of `KKQQ` is given in Section 4. Various aspects of the parallel formulation are described in Sections 5–8. Computational experiments are reported and analyzed in Section 9, and Section 10 concludes and mentions future work.

2 Notation and terminology

Section 2.1 introduces notation and terminology related to numerical linear algebra and Section 2.2 introduces notation and terminology related to parallel and distributed computing.

2.1 Numerical linear algebra

A *rotation* in the $(k, k+1)$-plane is an $n \times n$ real orthogonal matrix of the form

$$G = \begin{bmatrix} I_{k-1} & & \\ & \begin{matrix} c & s \\ -s & c \end{matrix} & \\ & & I_{n-k-1} \end{bmatrix},$$

where $c^2 + s^2 = 1$ and I_k is the $k \times k$ identity matrix. In the matrix multiplication GA, the rotation G is said to *act on* rows k and $k+1$ of A, since only these two rows of the product differ from the corresponding rows of A. Similarly, in the multiplication AG, the rotation acts on columns k and $k+1$. A rotation as defined here is a restricted form of a Givens rotation acting on two adjacent rows/columns.

A *transformation* is an $n \times n$ orthogonal matrix of the form

$$U = \begin{bmatrix} I_{k-1} & & \\ & \hat{U} & \\ & & I_{n-k-m+1} \end{bmatrix},$$

where $\hat{U}$ is an orthogonal matrix of size $m \times m$. A transformation is said to *act on* rows $k : k + m - 1$ in the matrix multiplication UA and on the corresponding set of columns in AU. Note that a rotation is just a special case of a transformation with

$$\hat{U} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}.$$

A *rotation sequence* $\langle G_1, G_2, \ldots, G_r \rangle$ is an ordered set of rotations acting on rows (columns) from the bottom up (from right to left). Formally, if G_i acts on rows/columns k and $k+1$, then G_{i+1} acts on rows/columns $k-1$ and k.

A *transformation sequence* $\langle U_1, U_2, \ldots, U_r \rangle$ is an ordered set of transformations acting on rows (columns) from the bottom up (from right to left). Formally, if U_i acts on rows/columns $k_1 : k_2$ and U_{i+1} acts on rows/columns $k_1' : k_2'$, then $k_1' \leq k_1 \leq k_2' \leq k_2$. Note that a rotation sequence is a special case of a transformation sequence.

A *rotation graph* $\mathcal{R} = (\mathcal{V}, \mathcal{E})$ is a partially ordered set of rotations. A vertex $v \in \mathcal{V}$ represents a rotation and is uniquely labeled by a pair of integers (i, j). An edge $(u, v) \in \mathcal{E}$ represents a precedence constraint on the two rotations and states that u must be applied before v to preserve correctness of some numerical computation. The set of vertex labels in $\mathcal{R}$ is defined by a positive integer s and two sequences of integers $\langle \ell_1, \ldots, \ell_s \rangle$ and $\langle u_1, \ldots, u_s \rangle$ as follows:

$$\{(i, j) \mid 1 \leq i \leq s \quad \text{and} \quad \ell_i \leq j \leq u_i\}.$$

Moreover, the lower and upper bounds ℓ_i and u_i are constrained such that $\ell_{i+1} \in \{\ell_i, \ell_i + 1\}$ and $u_{i+1} \in \{u_i, u_i + 1\}$. The edge set $\mathcal{E}$ is defined by the following rules on the vertex labels:

1. If u is labeled (i, j) and v is labeled $(i, j-1)$, then $(u, v) \in \mathcal{E}$.
2. If u is labeled (i, j) and v is labeled $(i+1, j+1)$, then $(u, v) \in \mathcal{E}$.

For a given i, the subset of the vertices whose labels are in the set $\{(i, j) \mid \ell_i \leq j \leq u_i\}$ form a rotation sequence referred to as *sequence i of $\mathcal{R}$*. A rotation graph captures all possible ways to reorder the application of a set of rotations without altering the numerical result compared to the baseline of applying the sequences in the order $1, 2, \ldots, s$. Figure 1 provides a small example of a rotation graph with two sequences of rotations.

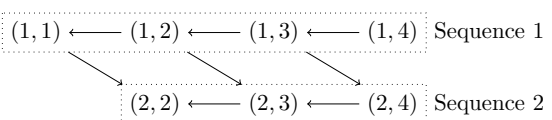

Figure 1: Example of a rotation graph with $s = 2$, $\ell_1 = 1$, $\ell_2 = 2$, and $u_1 = u_2 = 4$.

A *rotation supernode* $\mathcal{R}'$ of a rotation graph $\mathcal{R}$ is a subgraph of $\mathcal{R}$ with no directed path that both starts and ends in $\mathcal{R}'$ and contains a vertex not in $\mathcal{R}'$. Given a partitioning of the vertices of a rotation graph $\mathcal{R}$ into disjoint rotation supernodes, the graph that is induced by contracting each supernode is a directed acyclic graph referred to as a *rotation supergraph*. A rotation supergraph generalizes the concept of a rotation graph to coarser units of computation. Each rotation supernode can be explicitly accumulated to form a transformation and then the rotation supergraph can be applied using matrix–matrix multiplications based on these transformations. Figure 2 illustrates one of many possible ways to form a rotation supergraph (bottom) from a given rotation graph (top).

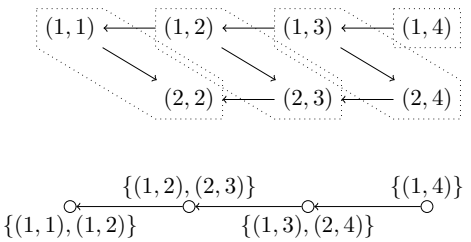

Figure 2: Illustration of a partitioning of a rotation graph into rotation supernodes (top) and the resulting rotation supergraph (bottom).

2.2 Parallel computing

The parallel computer consists of $P = P_\mathrm{r}P_\mathrm{c}$ processors/cores, each running a process with its own private memory. The processes are arranged in a logical two-dimensional mesh of size $P_\mathrm{r} \times P_\mathrm{c}$ and are labeled with (p, q) where $p \in \{0, 1, \ldots, P_\mathrm{r} - 1\}$ is the mesh row index and $q \in \{0, 1, \ldots, P_\mathrm{c} - 1\}$ is the mesh column index.

The processes communicate by sending explicit messages. Point-to-point messages have non-blocking send semantics and blocking receive semantics, while all collective operations are blocking. These are the semantics used in the Basic Linear Algebra Communication Subprograms (BLACS) library [7], which we used in the implementation. The same semantics can also be obtained using the Message Passing Interface (MPI).

A matrix A is said to be distributed over the process mesh with a *two-dimensional block-cyclic distribution* with block size n_b if the matrix element a_{ij} is assigned to process (p, q), where $p = \lfloor (i-1)/n_\mathrm{b} \rfloor \bmod P_\mathrm{r}$ and $q = \lfloor (j-1)/n_\mathrm{b} \rfloor \bmod P_\mathrm{c}$.

3 The sequential KKQQ HT reduction algorithm

This section recalls the KKQQ HT reduction algorithm [11] since it is the foundation of our parallel formulation. Refer to the original publication for details.

The idea of Moler and Stewart's algorithm [13] is to systematically reduce the columns of A from left to right using a sequence of rotations applied from the bottom up for each column. After reducing a column, the resulting sequence of rotations is applied to the upper triangular matrix B, which creates fill in the first sub-diagonal and thus changes the structure of B from upper triangular to upper Hessenberg. The next step is to remove the fill in B by a sequence of rotations applied from the right. This step can be viewed as an RQ factorization of a Hessenberg matrix. The resulting rotations are then applied also to A. After applying this procedure to the first $n-2$ columns of A, the HT reduction is complete. The orthogonal matrices Q and Z that encode the transformation from (A, B) to (H, T) can be obtained by accumulating the rotations applied from the left into Q and the rotations applied from the right into Z. The main idea of the KKQQ algorithm is to delay a large fraction of the work involved in the application of rotations in Moler and Stewart's algorithm until the work can be applied more efficiently (in terms of communication through the memory hierarchy) using matrix–matrix multiplications.

Algorithm 1 gives an overview of the KKQQ algorithm and can be described as follows.

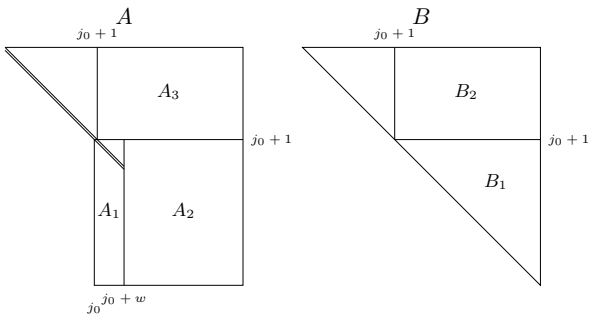

Figure 3: Block partitioning of the matrices A and B used by Algorithm 1.

The outer loop on line 1 loops over column panels of A of width w from left to right and j_0 denotes the first column index of the current panel. The first row index of the panel is $j_0 + 1$. The panel is denoted by A_1 in Figure 3. The current panel width is determined on line 2. The matrices A and B are logically partitioned as in Figure 3. Two rotation graphs $\mathcal{R}_{\text{left}}$ (for rotations applied from the left) and $\mathcal{R}_{\text{right}}$ (for rotations applied from the right) are initialized on line 4. The rotation graphs are empty in the sense that there are no rotations yet associated with the vertices. The remainder of the outer loop body consists of three phases: the *rotation construction phase*, where rotations are constructed with a minimum of work performed, the *rotation accumulation phase*, where the rotations are partitioned into rotation supernodes and explicitly accumulated, and the *delayed update phase*, where the remaining work is performed by applying the transformations. The three phases are described in more detail below.

Rotation construction: The rotation construction phase encompasses the entire inner loop on line 5. This loop iterates over the columns in the current panel from left to right and j denotes the current column index. The current column is brought up-to-date with respect to delayed updates from previous iterations in the inner loop on line 6. The current column is then reduced on line 7, which generates sequence $j - j_0 + 1$ of $\mathcal{R}_{\text{left}}$. This newly constructed sequence is then applied to B_1 from the left on line 8, which creates fill in its first sub-diagonal. The block B_1 is then reduced back to upper triangular form on line 9, which generates sequence $j - j_0 + 1$ of $\mathcal{R}_{\text{right}}$. This newly constructed sequence is then applied to A_1 and A_2 from the right on line 10.

Rotation accumulation: The rotation construction phase has generated the two rotation graphs $\mathcal{R}_{\text{left}}$ and $\mathcal{R}_{\text{right}}$. The purpose of the rotation accumulation phase is to partition these graphs into rotation supernodes of an appropriate size and then explicitly accumulate each supernode into a transformation. The supernodes should in general span all $\hat{w}$ sequences (effectively resulting in a rotation supergraph that is linear) and their size should be such that the transformations are of size close to $2\hat{w} \times 2\hat{w}$ to minimize the overhead of the accumulation [12]. All of this occurs on line 11.

Algorithm 1: The KKQQ [11, Algorithm 3.2] HT reduction algorithm

Data: Matrices $A, B \in \mathbb{R}^{n\times n}$, where B is upper triangular, orthogonal matrices $Q, Z \in \mathbb{R}^{n\times n}$, and a block size $w \in \{1, 2, \ldots, n\}$.

```
   // Loop over panels of width w from left to right
1  for j0 ← 1 : w : n − 2 do
       // Determine the current block size
2      ŵ ← min{w, n − j0 − 1};
       // THE ROTATION CONSTRUCTION PHASE
3      Partition A and B as in Figure 3;
4      Let R_left and R_right be empty rotation graphs (no rotations yet attached to the vertices) with ŵ
         sequences, lower bounds ℓ_i = j0 + i, and upper bounds u_i = n − 1 for i = 1, 2, ..., ŵ;
       // Loop over the columns in the panel from left to right
5      for j ← j0 : 1 : j0 + ŵ − 1 do
6          Apply sequences 1, 2, ..., j − j0 of R_left to the j'th column of A;
7          Reduce the j'th column of A and add the rotations as sequence j − j0 + 1 of R_left;
8          Apply sequence j − j0 + 1 of R_left to B1 from the left;
9          Reduce B1 from the right and add the rotations as sequence j − j0 + 1 of R_right;
10         Apply sequence j − j0 + 1 of R_right to A1 and A2 from the right;
       // THE ROTATION ACCUMULATION PHASE
11     Partition R_left and R_right into rotation supergraphs and accumulate each rotation supernode into
         a transformation;
       // THE DELAYED UPDATE PHASE
12     Apply the transformations in R_left to A2 from the left;
13     Apply the transformations in R_left to Q^T from the left;
14     Apply the transformations in R_right to A3 from the right;
15     Apply the transformations in R_right to B2 from the right;
16     Apply the transformations in R_right to Z from the right;
```

Delayed updates: The transformations are applied using matrix–matrix multiplications to parts of A, B, Q, and Z in the delayed update phase. The rotations from the left are applied to A_2 and Q^T on lines 12–13. The rotations from the right are applied to A_3, B_2, and Z on lines 14–16.

4 Overview of the parallel formulation

Our parallel formulation of Algorithm 1 consists of several parts that are parallelized in different ways. This section gives an overview of the algorithm and identifies the parts and connects them to the underlying sequential algorithm. The details of each part are given separately in Sections 5–8.

The input matrices A, B, Q^T, and Z are assumed to be identically distributed across the process mesh using a two-dimensional block-cyclic distribution with block size n_b. The notation $A^{(p,q)}$ refers to the submatrix of the distributed matrix A that is assigned to process (p, q). Similarly, the notations $A^{(p,*)}$ and $A^{(*,q)}$ refer to the submatrices assigned to mesh row p and mesh column q, respectively.

The rotation graphs $\mathcal{R}_\mathrm{left}$ and $\mathcal{R}_\mathrm{right}$ are represented by two-dimensional arrays of size $n \times \hat{w}$ and are replicated on all processes.

The following are the main parts of the parallel formulation. The update and reduction of the current column of A (lines 6–7) is referred to as UPDATEANDREDUCECOLUMN and is described in Section 5. The application of a sequence of rotations from the left (line 8) is referred to as

ROWUPDATE and is described in Section 6. The reduction of B back to triangular form (line 9) is referred to as RQFACTORIZATION and is described in Section 7. The application of a sequence of rotations from the right (line 10) is referred to as COLUPDATE and is conceptually similar to ROWUPDATE. The accumulation of rotations into transformations (line 11) is referred to as ACCUMULATE and is described in Section 8. The application of a sequence of transformations from the left (lines 12–13) is referred to as BLOCKROWUPDATE and is conceptually similar to ROWUPDATE. Finally, the application of a sequence of transformations from the right (lines 14–16) is referred to as BLOCKCOLUPDATE and is also conceptually similar to ROWUPDATE.

5 Updating and reducing a single column

The input is a rotation graph and a partial column of a distributed matrix. The purpose is to apply the rotation graph to the column and then reduce it by a new sequence of rotations. Since applying a rotation sequence to only one column is inherently sequential, parallelism can be extracted only by pipelining the application of multiple sequences. Specifically, one process can apply rotations from sequence i while another process applies rotations from sequence $i+1$. With s sequences, up to s processes can be used in parallel with this pipelining approach. In practice, however, such a parallelization scheme is very fine-grained and leads to a lot of parallel overhead—especially in a distributed memory environment—and therefore requires a sufficiently long column and sufficiently many sequences to yield any speedup. Therefore, we dynamically decide on a subset of the processes (ranging from a single process to the entire mesh) onto which we redistribute the column and apply the sequences in parallel. While this part of the algorithm accounts for a tiny proportion of the overall work, its limited scalability makes it a theoretical bottleneck that ultimately limits the overall scalability of our parallel HT reduction.

6 Wavefront scheduling of a rotation sequence

The dominating computational pattern in Algorithm 1 is the application of a sequence of rotations or transformations and is manifested in ROWUPDATE, COLUPDATE, BLOCKROWUPDATE, and BLOCKCOLUPDATE. There are also close connections to the pattern in RQFACTORIZATION. This section describes a novel scheduling algorithm that is capable of using the processes efficiently and generates a pattern of computation that resembles wavefronts, hence the name.

We treat in this section only the special case of applying a sequence of rotations from the left to a dense matrix. The method readily extends to upper triangular matrices, to sequences applied from the right, and to sequences of transformations. We assume that the sequence of rotations is replicated on all processes.

A first observation is that applying rotations from the left does not cause any flow of information across columns of the matrix. Hence, each column can be independently updated. In our parallel setting, this implies that there will be neither communication nor synchronization between processes on different mesh columns. Therefore, we may assume without loss of generality that the matrix is distributed on a $P_\mathrm{r} \times 1$ mesh, i.e., a mesh with a single column. For a matrix distributed on a general mesh, one simply applies the scheduling algorithm independently on each mesh column.

The rest of this section is organized as follows. Section 6.1 motivates the need for a more versatile scheduling algorithm by illustrating why a straightforward approach leads to poor

scalability. Section 6.2 gives a high-level overview of the algorithm. Fundamental building blocks of the algorithm are detailed in Section 6.3, and the final details of the algorithm as a whole are given in Section 6.4.

6.1 Why a straightforward approach scales poorly

The straightforward approach of applying each rotation completely one after the other does not scale because most of the time only one of the P processes are active and occasionally two processes are active. This is illustrated by the Gantt-chart in Figure 4. Each rotation in the sequence is classified as *local* if the two affected rows belong to the same process and as *cross-border* if the two rows belong to different processes. Hence, roughly a fraction $(n_\mathrm{b}-1)/n_\mathrm{b}$ of all rotations are local and only a fraction $1/n_\mathrm{b}$ are cross-border.

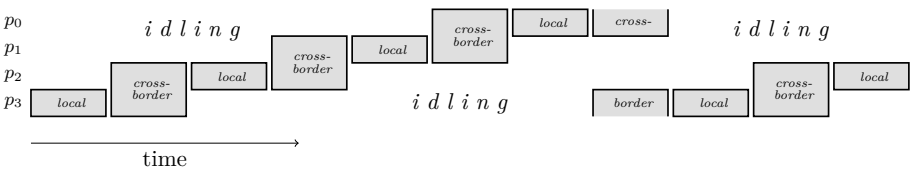

Figure 4: Gantt-chart for the straightforward application of a sequence of rotations from the left using a 4×1 mesh. At most two processes are active at the same time.

6.2 Overview of the wavefront scheduling algorithm

To improve on the straightforward application it becomes necessary to split the application of a rotation into several independent operations and introduce parallelism through pipelining. Specifically, the columns are partitioned into $\hat{N}_\mathrm{b}$ blocks of size $\hat{n}_\mathrm{b}$. The block size $\hat{n}_\mathrm{b}$ is independent of the distribution block size n_b, but is closely related to the degree of concurrency and should be chosen such that $\hat{N}_\mathrm{b} \geq P_\mathrm{r}$ in order to use all processes. With fewer than P_r blocks, there is not enough concurrency to keep all processes busy at the same time.

The $\hat{N}_\mathrm{b}$ distributed column blocks are referred to as *fragments* and play a key role in the scheduling algorithm. The operations necessary to update a fragment are decomposed into an alternating sequence of local and cross-border *actions*. A *local action* is the application of a maximal contiguous subsequence of local rotations, and a *cross-border action* is the application of a cross-border rotation. Due to the flow of data upwards in each column, the actions associated with a particular fragment need to be performed in a strictly sequential order.

Consider a single fragment at any given moment during the computation. Either the fragment has already been completely updated or there is a uniquely identified next action to perform on that fragment. If the next action is a local action, then the fragment is associated with the process that owns the distribution block affected by the local rotations. If the action is instead a cross-border action, then the fragment is associated with a pair of adjacent processes, namely those that own the two distribution blocks affected by the cross-border rotation. In essence, each fragment is at any point in time associated either with a particular process or with a particular pair of adjacent processes. This association is used to make the scheduling algorithm more efficient.

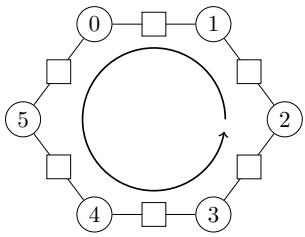

Figure 5: The twelve slots (local slots as circles and cross-border slots as squares) for the case of $P = 6$ processes. The circular arrow shows how the slots are ordered.

Associated with each process and with each pair of adjacent processes is a *slot*. A slot contains all the fragments that are currently associated with its related process(es). There are P_r *local slots*, each associated with a single process, and also P *cross-border slots*, each associated with a pair of adjacent processes. As an example, consider the twelve slots for the case $P_\mathrm{r} = 6$ illustrated in Figure 5. The local slots (and also the processes) are identified by circles. The cross-border slots are identified by squares. The adjacency of slots and processes is illustrated by lines. A fragment systematically moves from slot to slot in response to the completion of its actions. The direction in which a fragment moves is indicated by an arrow in Figure 5.

The wavefront scheduling algorithm revolves around the concept of a parallel step. A *parallel step* is a set of actions of the same type (local or cross-border) for which no two actions belong to fragments residing in the same slot. A parallel step is *maximal* if it involves one action from every slot of the chosen type. Since the type of action in a parallel step is homogeneous, one can refer to the steps as either local or cross-border. The actions in a parallel step can be performed in a perfectly parallel fashion (see Section 6.3 below) and a maximal step leads to no idling, which makes parallel steps useful as building blocks for an efficient schedule. The aim of the wavefront scheduling algorithm is to construct and execute a shortest possible sequence of parallel steps.

How to obtain a minimal sequence of parallel steps is an open problem. We conjecture that using a greedy algorithm that adheres to the following rules will yield a close-to-optimal solution.

1. Choose between a local and cross-border parallel step based on which type of step will lead to the execution of the most actions.

2. Choose from each slot of the appropriate type (local or cross-border) the fragment that has the most remaining actions.

The rationale behind Rule 1 is to greedily do as much work as possible. The rationale behind Rule 2 is to avoid ending up with a few fragments with many remaining actions and is based on the critical path scheduling heuristic.

6.3 Executing a parallel step

6.3.1 Executing a local parallel step

A local parallel step is perfectly parallel since it involves only local actions and hence requires neither communication nor synchronization. Ideally, the parallel step is maximal and then every process has exactly one task to perform and the tasks have roughly the same execution time as a consequence of the homogeneous block size $\hat{n}_b$.

6.3.2 Executing a cross-border parallel step

A cross-border parallel step involves pairwise exchanges of data between pairs of adjacent processes. Ideally, the parallel step is maximal and in this case every process would be involved in two unrelated cross-border actions.

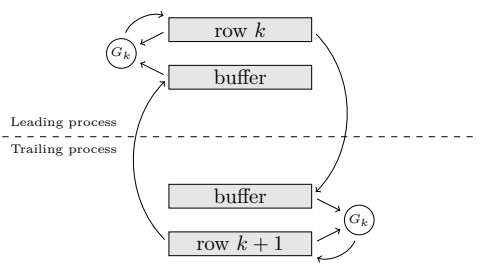

Figure 6: Illustration of a cross-border rotation G_k applied to rows k and $k+1$ of a fragment. The two processes begin by exchanging rows and finish by updating their respective local rows.

Figure 6 illustrates how the work of applying a cross-border rotation to a fragment is coordinated between the two processes involved in the operation. A cross-border rotation G_k is applied to rows k and $k+1$. The former is held by the so-called *leading process* and the latter by the so-called *trailing process*. Both processes begin by sending their respective local rows to the other process. Now each process has a copy of both rows and finishes by updating its local row.

Algorithm 2: Execution of a cross-border parallel step

```
1  Let p ∈ {0, 1, ..., P − 1} be the rank of this process;
2  Let L denote the action (if any) in which this process is the leading process;
3  Let T denote the action (if any) in which this process is the trailing process;
4  Let k_L and k_T denote the indices of the corresponding cross-border rotations;
5  if L is defined then
6      Send row k_L of the fragment associated with L to process (p + 1) mod P;
7  if T is defined then
8      Send row k_T + 1 of the fragment associated with T to process (p − 1) mod P;
9      Receive a row from process (p − 1) mod P into a buffer;
10     Update row k_T + 1 of the fragment associated with T using rotation G_{k_T};
11 if L is defined then
12     Receive a row from process (p + 1) mod P into a buffer;
13     Update row k_L of the fragment associated with L using rotation G_{k_L};
```

Since a process can be involved in two unrelated cross-border actions in one parallel step, one needs to schedule the sends and receives in a way that avoids deadlock. Algorithm 2 describes one solution to this problem for a general (i.e., not necessarily maximal) cross-border parallel step. A basic observation that is fundamental to the communication algorithm is that if a process is involved in two actions, then it will be the leading process in one of them and the trailing process in the other. That Algorithm 2 is deadlock-free is shown in Proposition 1.

Proposition 1. *Algorithm 2 is deadlock-free for any number of processes assuming that sends are non blocking.*

Proof. We first show that deadlock cannot occur for a maximal parallel step. Then we argue that deadlock cannot occur for non-maximal parallel steps either.

Consider a maximal parallel step. Label the two sends by s_1 and s_2 in the order that they are executed by Algorithm 2. Similarly, label the two receives r_1 and r_2 in the order they are executed. Since the step is maximal, both L and T will be defined and hence all then-clauses will be executed. Only the ordering of the sends and receives are relevant for the purpose of analyzing for deadlock. Consider any process $p_k \in \{0, 1, \ldots, P-1\}$. Figure 7 illustrates the sequencing due to program order of the two sends and two receives (middle column) executed by process p_k. Also shown are the sends and receives of the two adjacent processes (left and right columns, respectively) and the matching of sends and receives. Figure 7 is merely a template from which a complete dependence graph can be constructed for any given P. In the extreme case of $P = 2$, the left and right columns are actually the same. In the extreme case of $P = 1$, there is only one column and each send is matched by a receive on the same process.

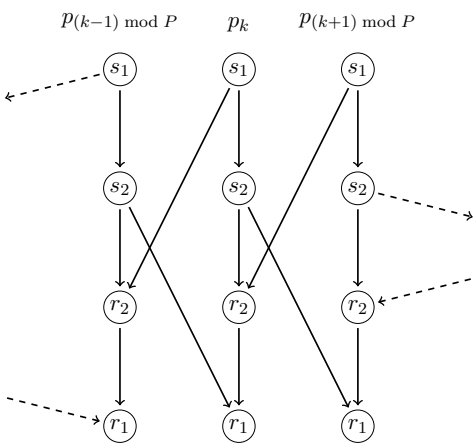

Figure 7: Dependencies between sends and receives in Algorithm 2 illustrated for three adjacent processes.

There can be at most one ready task per column, due to the (vertical) program order dependencies. Furthermore, there is one process dedicated to each column and therefore every ready task will eventually be executed. As a consequence, the only way for deadlock to occur is if the dependence graph contains a directed cycle. Since every edge in Figure 7 is directed

downwards, a directed cycle cannot exist regardless of P. This shows that Algorithm 2 is deadlock-free for maximal parallel steps.

Suppose that the step is not maximal. Then we effectively need to remove some of the nodes and edges in the dependence graph. The arguments used for the maximal case remain valid and hence the algorithm is deadlock-free also for this case. □

6.4 The wavefront scheduling algorithm

This section describes the details of the wavefront scheduling algorithm (Algorithm 3).

Algorithm 3: Wavefront scheduling

```
1  S_L is the set of local slots;
2  S_B is the set of cross-border slots;
3  loop
       // Select non-empty slots of the same type
4      S ← {s ∈ S_L ∪ S_B : Size(s) > 0};
5      n_L ← |S ∩ S_L|;
6      n_B ← |S ∩ S_B|;
7      if n_L > n_B then
8          S ← S ∩ S_L;
9      else
10         S ← S ∩ S_B;
       // Terminate if all actions have been performed
11     if S = ∅ then terminate;
       // Extract one fragment from each selected slot
12     F ← {Select(s) : s ∈ S};
       // Perform the parallel steps
13     if n_L > n_B then
14         Perform a local parallel step on the fragments in F (Section 6.3.1);
15     else
16         Perform a cross-border parallel step on the fragments in F (Section 6.3.2);
       // Move (or remove) the updated fragments
17     foreach f ∈ F do
18         if ActionCount(f) > 0 then
19             Slot(f) ← Next(Slot(f));
20         else
21             Undefine Slot(f);
```

At the top level of Algorithm 3 is a loop that continues until all actions have been performed. A subset $\mathcal{S}$ of the slots is chosen such that all slots in the set have the same type and contain at least one fragment. The choice is made according to Rule 1 in Section 6.2. The function `Size` returns for a given slot the number of fragments (possibly zero and at most $\hat{N}_\mathrm{b}$) that currently resides in that slot. The algorithm terminates when there is no fragment in any of the slots. The set $\mathcal{S}$ of slots is then mapped to a set $\mathcal{F}$ of fragments by selecting from each slot the fragment with the most remaining actions. The choice is made according to Rule 2 in Section 6.2. The selection is carried out by the function `Select`. The parallel step defined by the set of fragments $\mathcal{F}$ is performed as described in Section 6.3. Finally, each fragment is either moved to its next slot or removed altogether. The function `Slot` returns the slot in which a given fragment currently resides (or is undefined if the fragment has been completed). The function `Next` returns the next slot relative to a given slot. The function

`ActionCount` returns the number of actions remaining in a given fragment.

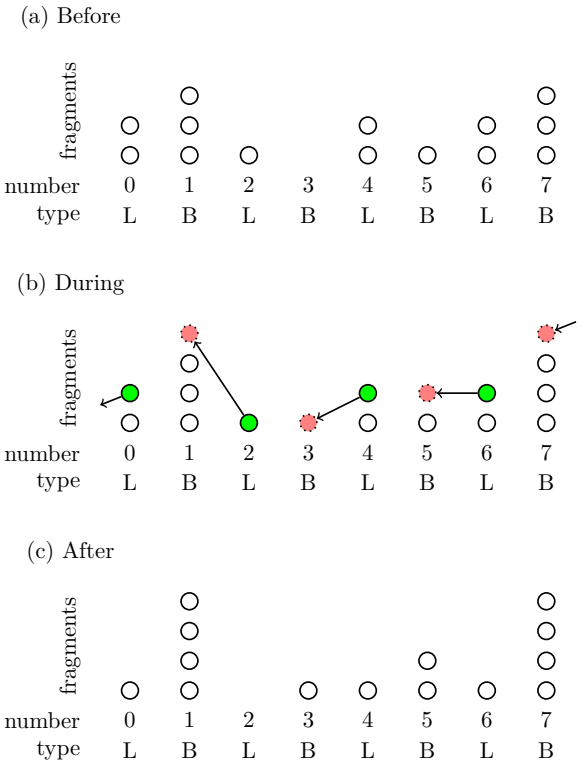

Figure 8: Example illustrating the movement of fragments after an iteration of Algorithm 3 that resulted in the execution of a local parallel step.

Figure 8 provides an example of the movement of fragments after one iteration of Algorithm 3 for the case of $P = 4$ processes. Each fragment is illustrated by a circle and the eight slots are indicated by eight columns labeled 0 through 7 in a clockwise direction relative to Figure 5. Figure 8(a) shows the state of the slots *before* the iteration. Since there are $n_{\mathrm{L}} = 4$ non-empty local (L) slots but only $n_{\mathrm{B}} = 3$ non-empty cross-border (B) slots, Algorithm 3 will choose to perform a local parallel step and set $\mathcal{S} = \{0, 2, 4, 6\}$. One fragment from each of the chosen slots will then be selected, and these are shown in green in Figure 8(b). After completing the parallel step, the four selected fragments move to their respective next slots, i.e., from slot $s \in \{0, 1, \ldots, 7\}$ to $(s - 1) \bmod 8$. The resulting state of the slots after the iteration is shown in Figure 8(c).

The bottom of Figure 9 illustrates a simulated schedule produced by Algorithm 3 for a rotation sequence applied from the right using a 4×4 process mesh. The processes are labeled in row-major order, so the first group of four traces belong to the first mesh row and so on. Besides the pipeline startup and shutdown phases, the processes are active all the time. The

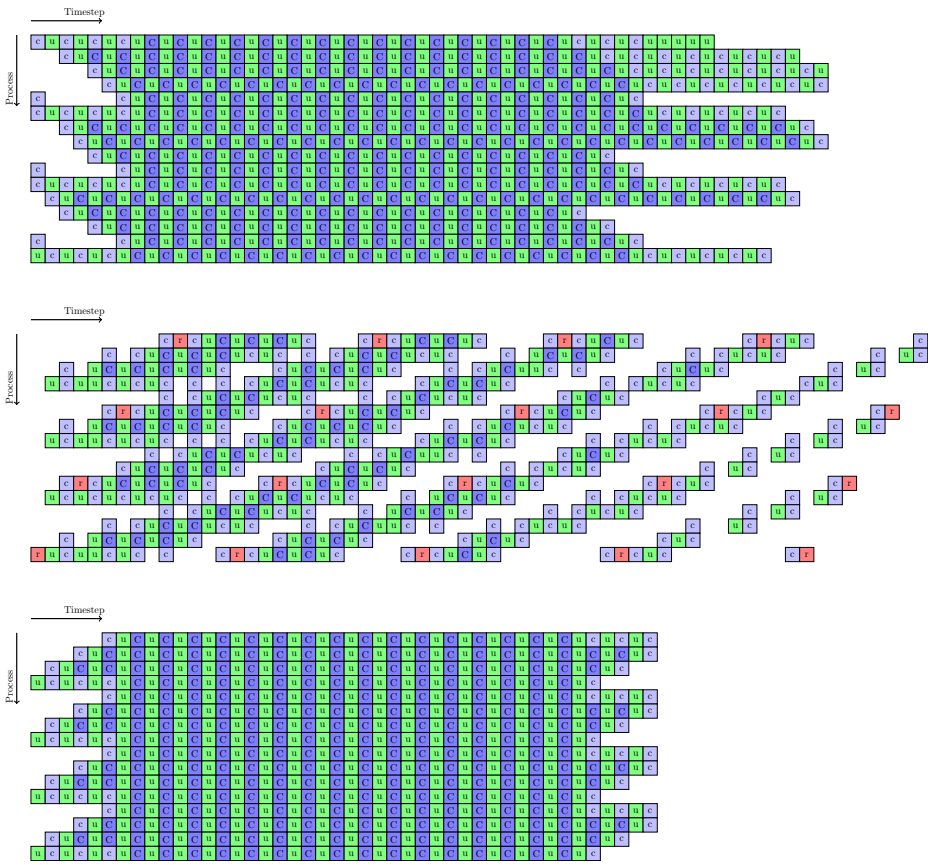

Figure 9: Illustration of simulated schedules produced by variants of the wavefront scheduling algorithm on a 4×4 process mesh. The label 'u' means a local action, 'r' a reduction action, 'c' (lower case) a single cross-border action, and 'C' (upper case) a pair of cross-border actions. *Top:* Rotations applied from the left to an upper triangular matrix (Section 6). *Middle:* Rotations created and applied from the right to reduce a Hessenberg matrix to upper triangular form (Section 7). *Bottom:* Rotations applied from the right to a dense matrix (Section 6).

top of the figure shows a corresponding trace for a rotation sequence applied from the left to an upper triangular matrix. (Note that the time scales in the three subfigures are different.) The mesh columns are less in sync and the load imbalance is more severe than in the dense case. However, the schedule can still activate all processes at the same time.

6.5 Overhead analysis of Algorithm 3

In this section, we analyze the overhead per iteration of the loop in Algorithm 3.

The slot data structures are numbered from 0 to $2P - 1$ and accessed in constant time. Each slot contains a set housing the fragments that currently reside in the slot.

Selecting the type of the parallel step and the non-empty slots can be accomplished in $\Theta(P)$ time by visiting each slot. Extracting from each slot a fragment with the most remaining actions can be accomplished in $\mathcal{O}(\hat{N}_\mathrm{b})$ time if an unordered set data structure is used and in $\mathcal{O}(\log \hat{N}_\mathrm{b})$ time if a priority queue data structure is used. Finally, moving the fragments can be accomplished in $\Theta(1)$ time if an unordered set data structure is used and in $\mathcal{O}(\log \hat{N}_\mathrm{b})$ time if a priority queue data structure is used.

In summary, the overhead per iteration is bounded by $\mathcal{O}(P+\hat{N}_\mathrm{b})$ if an unordered set data structure is used and by $\mathcal{O}(P + \log \hat{N}_\mathrm{b})$ if a priority queue data structure is used. For the special case $\hat{N}_\mathrm{b} = P$, the overhead is $\Theta(P)$ regardless of the underlying data structure.

Note that the context in this section is a mesh of size $P \times 1$. In reality, the scheduling algorithm is applied independently on each mesh row or mesh column. Thus, for a mesh of size $\sqrt{P} \times \sqrt{P}$, replace P with $\sqrt{P}$ in all of the above.

7 Wavefront RQ factorization of a Hessenberg matrix

The aim is to reduce an upper Hessenberg matrix to upper triangular form by creating and applying a rotation sequence from the right. The pattern of computation has many similarities with the pattern analyzed in Section 6 but with the addition that the rotations are not known beforehand. This small difference causes profound effects, since it introduces both the need for collective communication and also causes flow of information perpendicular to the flow introduced by the sequence itself. In this section, we show how to extend the algorithm presented in Section 6 to this computational pattern.

The rotation sequence is applied from the right, so the fragments are now row blocks instead of column blocks. Moreover, the fragments are now aligned with the distribution blocks, i.e., a fragment is the same as a distribution row block.

To generate new rotations and replicate them, we need to introduce a few more actions besides the local and cross-border actions. The last local action on a fragment updates a block on the diagonal. We replace this action with a new *reduction* action and remove the following (and final) cross-border action (if any). The purpose of a reduction action is to reduce the diagonal block and the column immediately to the left. After reducing the diagonal block, the new rotations need to be replicated using a new *reduction-broadcast* action.

Since the rotations are not known beforehand, the fragments can no longer make progress independently. To account for this, we distinguish between *active* and *inactive* fragments. The active fragments are those that can perform their next action and the inactive fragments are the remaining ones, i.e., those whose next action depends on rotations not yet locally available. Each slot now contains a mix of active and inactive fragments. The function `ActiveSize` returns the number of active fragments in a given slot.

Algorithm 4: Wavefront scheduling for the RQ factorization of a Hessenberg matrix

$\mathcal{S}^i_{\mathrm{L}}$ is the set of local slots on process mesh row $i \in \{0, 1, \ldots, P_{\mathrm{r}} - 1\}$;
$\mathcal{S}^i_{\mathrm{B}}$ is the set of cross-border slots on process mesh row $i \in \{0, 1, \ldots, P_{\mathrm{r}} - 1\}$;
Let (p, q) be the mesh row and column indices, respectively, of this process;
loop
    `// Select active slots`
    foreach $i \in \{0, 1, \ldots, P_{\mathrm{r}} - 1\}$ **do**
        $\mathcal{S}^i \leftarrow \{s \in \mathcal{S}^i_{\mathrm{L}} \cup \mathcal{S}^i_{\mathrm{B}} : \texttt{ActiveSize}(s) > 0\}$;
        $n^i_{\mathrm{L}} \leftarrow |\mathcal{S}^i \cap \mathcal{S}^i_{\mathrm{L}}|$;
        $n^i_{\mathrm{B}} \leftarrow |\mathcal{S}^i \cap \mathcal{S}^i_{\mathrm{B}}|$;
        if $n^i_{\mathrm{L}} > n^i_{\mathrm{B}}$ **then**
            $\mathcal{S}^i \leftarrow \mathcal{S}^i \cap \mathcal{S}^i_{\mathrm{L}}$;
        else
            $\mathcal{S}^i \leftarrow \mathcal{S}^i \cap \mathcal{S}^i_{\mathrm{B}}$;
    `// Terminate if all actions have been performed`
    if $\forall i \in \{0, 1, \ldots, P_{\mathrm{r}} - 1\} : \mathcal{S}^i = \emptyset$ **then** terminate;
    `// Extract one active fragment from each selected slot`
    foreach $i \in \{0, 1, \ldots, P_{\mathrm{r}} - 1\}$ **do**
        $\mathcal{F}^i \leftarrow \{\texttt{ActiveSelect}(s) : s \in \mathcal{S}^i\}$;
    $\mathcal{F} \leftarrow \mathcal{F}^0 \cup \mathcal{F}^1 \cup \cdots \cup \mathcal{F}^{P_{\mathrm{r}}-1}$;
    `// Find an active fragment (if any) whose next action is a reduction`
    $\mathcal{R} \leftarrow \{f \in \mathcal{F} : \texttt{Reduction}(f)\}$;
    `// Perform the parallel steps`
    if $n^p_{\mathrm{L}} > n^p_{\mathrm{B}}$ **then**
        Perform a *local parallel step* on the fragments in $\mathcal{F}^p$ (Section 6.3.1);
    else
        Perform a *cross-border parallel step* on the fragments in $\mathcal{F}^p$ (Section 6.3.2);
    if $\mathcal{R} \neq \emptyset$ **then**
        $\mathcal{R} = \{r\}$;
        Perform a *reduction-broadcast action* on the fragment r;
    `// Move (or remove) the updated fragments`
    foreach $f \in \mathcal{F}$ **do**
        if $\texttt{ActionCount}(f) > 0$ **then**
            $\texttt{Slot}(f) \leftarrow \texttt{Next}(\texttt{Slot}(f))$;
        else
            Undefine $\texttt{Slot}(f)$;

The details of the extended wavefront scheduling algorithm are presented in Algorithm 4. The new predicate `Reduction`, line 17, returns true if the next action on a given fragment is a reduction action. Unlike the original algorithm, the extended algorithm needs to keep track of the progress of all processes and not only the processes in its own mesh row. The need for this extra bookkeeping is to be able to determine when a process should execute the reduction-broadcast actions, see lines 22–24. The function `ActiveSelect`, which replaces `Select`, returns one of the active fragments from a given slot.

The middle of Figure 9 illustrates a schedule produced by Algorithm 4 using a 4×4 mesh. The reduction-broadcast action introduces a synchronization point that leads to significant overhead throughout the execution. The amount of idling would be less if the number of fragments was larger relative to the number of processes.

7.1 Overhead analysis of Algorithm 4

In this section, we analyze the overhead per iteration of the loop in Algorithm 4.

The following assumes that the mesh size is $\sqrt{P} \times \sqrt{P}$. The number of fragments is denoted by N_b and is independent of P as N_b depends only on the size of the problem size n and the distribution block size n_b.

Selecting active slots can be accomplished in $\Theta(P)$ time by scanning through all of the $\sqrt{P}$ slots associated with each of the $\sqrt{P}$ mesh rows. Extracting from each selected slot a fragment with the most remaining actions can be accomplished in $\mathcal{O}(\sqrt{P}N_\mathrm{b})$ time if an unordered set data structure is used and in $\mathcal{O}(\sqrt{P}\log N_\mathrm{b})$ time if a priority queue data structure is used. Finally, moving the fragments can be accomplished in $\Theta(\sqrt{P})$ time if an unordered set data structure is used and in $\mathcal{O}(\sqrt{P}\log N_\mathrm{b})$ time if a priority queue data structure is used.

In summary, the overhead per iteration is bounded by $\mathcal{O}(P + \sqrt{P}N_\mathrm{b})$ if an unordered set data structure is used and by $\mathcal{O}(P + \sqrt{P}\log N_\mathrm{b})$ if a priority queue data structure is used.

8 Accumulation of rotations into transformation matrices

The rotations from both sides are accumulated into transformations. The transformations should align (whenever possible) such that they act on two full and adjacent distribution block rows/columns. This implies that the typical size of a transformation matrix is $2n_\mathrm{b} \times 2n_\mathrm{b}$. Parallelism in the accumulation is exploited by assigning to each process a subset of the accumulation tasks. After the local accumulation phase, the resulting transformations are replicated across the relevant subsets of the mesh.

9 Computational experiments and results

To analyze the performance, scalability, and bottlenecks of our proposed parallel HT reduction algorithm, we performed a number of computational experiments. Details of the implementation are given in Section 9.1. The setups of the experiments and the parallel computer systems are described in Section 9.2. The results of the experiments are summarized briefly in Section 9.3 followed by details of each experiment in the subsequent Sections 9.4–9.6.

9.1 Implementation details

The algorithm is implemented in Fortran 90/95 and uses the BLAS library for basic matrix computations, the BLACS library for inter-process communication, and auxiliary routines from the ScaLAPACK library.

The block sizes $\hat{n}_\mathrm{b}$ used by the wavefront scheduling algorithm, see Section 6.2, were set on a per-call basis to the largest (although never smaller than 8) that exposes sufficient parallelism to activate all processes; the number of blocks $\hat{N}_\mathrm{b}$ is set to $2 \times P_\mathrm{c}$ for row operations and $2 \times P_\mathrm{r}$ for column operations. For the row update of B, the block size $\hat{n}_\mathrm{b}$ is set to $n/2\hat{N}_\mathrm{b}$ in order to compensate for the load imbalance caused by the upper triangular structure.

The value of w, in Algorithm 1, is typically set to the distribution block size n_b. The gain is two-fold. Firstly, all elements and rotations belong to the same process column when reducing w columns which makes the rotation accumulation and distribution simpler. Secondly, the transformations resulting from w inner loops will never span over more than two rows or columns which makes the blocked update easier to implement.

The input matrices Q and Z are treated as dense without structure.

9.2 Experiment setup

Two different parallel computer systems were used in the experiments: Triolith at the National Supercomputer Centre (NSC) at Linköping University and Abisko at the High Performance Computing Center North (HPC2N), at Umeå University. See Table 1 for details.

Table 1: Information about the Abisko and Triolith systems.

Abisko	**64-bit AMD Opteron Linux Cluster**
Processors	Four AMD Opteron 6238 processors (12 cores) per node
Interconnect	Mellanox Infiniband
Compiler	Intel compiler
Libraries	Intel MPI, ACML 5.3.1 (includes LAPACK functionality), ScaLAPACK 2.0.2
Triolith	**64-bit HP Cluster Platform 3000 with SL230s Gen8 compute nodes**
Processors	Two Intel Xeon E5-2660 processors (8 cores) per node
Interconnect	Mellanox Infiniband
Compiler	Intel compiler
Libraries	Intel MPI, Intel MKL 11.3 (includes ScaLAPACK and LAPACK functionality)

Each *test case* is specified by four integer parameters: the problem size n, the distribution block size n_b, and the process mesh size $P_\mathrm{r} \times P_\mathrm{c}$. Except n, all parameters are tunable and can be chosen to maximize performance. Since exhaustive search for optimal parameters is prohibitively expensive, preliminary experiments were used to determine a reasonable block size n_b. Only square meshes ($P_\mathrm{r} = P_\mathrm{c}$) were considered.

The input matrices are randomly generated with elements drawn from the standard normal distribution. The time required by the initial reduction of B to triangular form is not included in the measurements.

9.3 Summary of the experiments

Three sets of experiments were performed:

- *Experiment 1: Reasonable distribution block sizes*
  The purpose of this experiment was to determine a reasonable distribution block size

to use for the subsequent experiments. The results of the experiment indicate that $n_b = 100$ is reasonable on both machines. See Section 9.4 for details.

- *Experiment 2: Scalability*
  The purpose of this experiment was to evaluate the weak and strong scalability relative to a state-of-the-art sequential implementation [11]. See Section 9.5 for details.

- *Experiment 3: Bottlenecks*
  The purpose of this experiment was to characterize the cost and scalability of the major parts of the parallel algorithm and identify bottlenecks that currently limit its scalability. See Section 9.6 for details.

9.4 Experiment 1: Reasonable distribution block sizes

The distribution block sizes $n_b \in \{40, 60, 80, \ldots, 160\}$ were tested on a problem of size $n = 4000$ and mesh of size $P_r = P_c = 4$ with the aim of finding a reasonable block size to use for the subsequent experiments. The parallel execution times (in seconds) on both machines are shown in Table 2. These results indicate that $n_b = 100$ is reasonable on both machines.

Table 2: Impact of n_b on Triolith and Abisko ($n = 4000$, $P_r = P_c = 4$). Times in seconds.

	$n_b = 40$	$n_b = 60$	$n_b = 80$	$n_b = 100$	$n_b = 120$	$n_b = 160$
Triolith	32	28	28	27	27	28
Abisko	66	57	54	53	53	56

More detailed experiments show that a smaller block size, down to 60, will make the serial code somewhat faster, and the parallel slower. A larger block size, up to 140 will make the parallel version somewhat faster, but the serial implementation slower. The gain and loss in performance is however less than 10%, so using $n_b = 100$ is fair and close to optimal for the experiments discussed in Section 9.5 and 9.6.

9.5 Experiment 2: Scalability

The strong scalability of the parallel algorithm was measured in terms of speedup relative to a state-of-the-art sequential implementation of `KKQQ` [11] for problems of size $n \in \{4000, 8000, 12000, 16000\}$ and meshes of size $P_r \times P_c$ for $P_r = P_c \in \{1, 2, \ldots, 10\}$.

The results of the strong scalability experiments are shown in Figure 10. The sequential algorithm is faster on one core than the parallel algorithm. The strong scalability increases with larger problem sizes. The parallel efficiency is low, which indicates that much larger problems than those tested are necessary for the algorithm to run efficiently.

Both memory-constrained and time-constrained forms of weak scalability were analyzed. The *memory-constrained* weak scalability was measured by scaling up the problem size n with the number of cores to keep the memory required per core constant. Specifically, for a problem of size n_1 on one core, the problem size on P cores was set to $n_1\sqrt{P}$. The *time-constrained* weak scalability, on the other hand, was measured by scaling up the problem size with the number of cores to keep the flops required per core constant. Specifically, for a problem of size n_1 on one core, the problem size on P cores was set to $n_1\sqrt[3]{p}$.

The results of the weak scalability experiments, with $n_1 = 1000$, are shown in Figure 11. The memory-constrained problem scales well using up to 256 cores on both machines. Adding

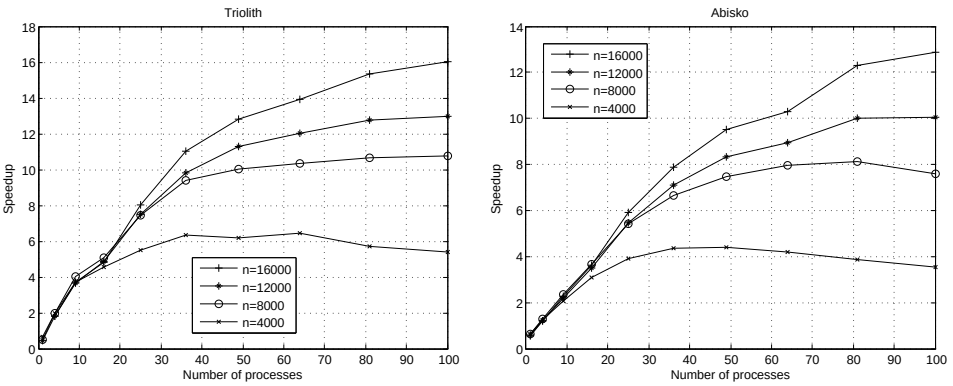

Figure 10: Strong scalability (speedup relative to `KKQQ` on one core).

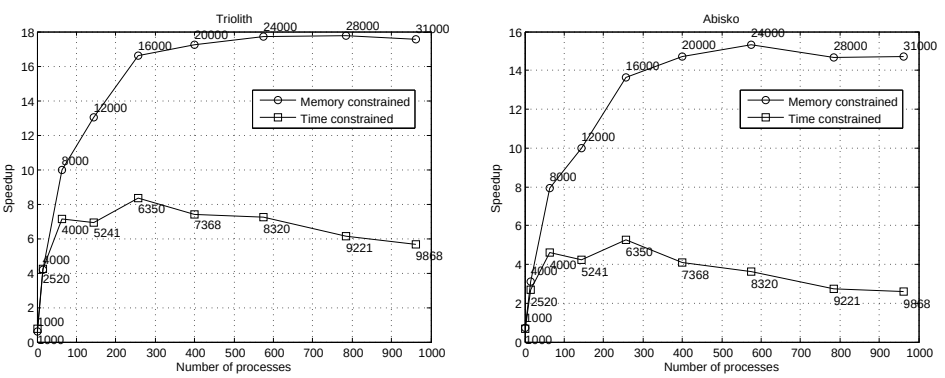

Figure 11: Weak scalability (speedup relative to `KKQQ` on one core). The labels show the scaled problem size n.

more cores is not beneficial, since the time for communication and synchronization will increase more and more, relative to the time for computation. The time-constrained setup scales well using up to 64 cores, adding more cores is not fruitful at all. The setup suffers from a small n_1 resulting in smaller and smaller local problem size when the process mesh size, and therefore the amount of communication, is increased. A larger n_1 allows the use of more cores efficiently. For example, using $n_1 = 2000$ allows for using up to 256 cores on Abisko, instead of 64, before the peak is reached.

9.6 Experiment 3: Bottlenecks

Profiling data was gathered in an effort to identify bottlenecks that can explain the limited scalability observed in Section 9.5. For each call to a major subroutine, measurements of the parallel execution time, the flop count, and the time spent in numerical computation were made. For the purpose of these measurements, barrier synchronizations were inserted before each subroutine call. In this way, load imbalances caused by one subroutine call do not affect the measurements of the next. The names of the subroutines referred to in this section correspond to the definitions in the end of Section 4.

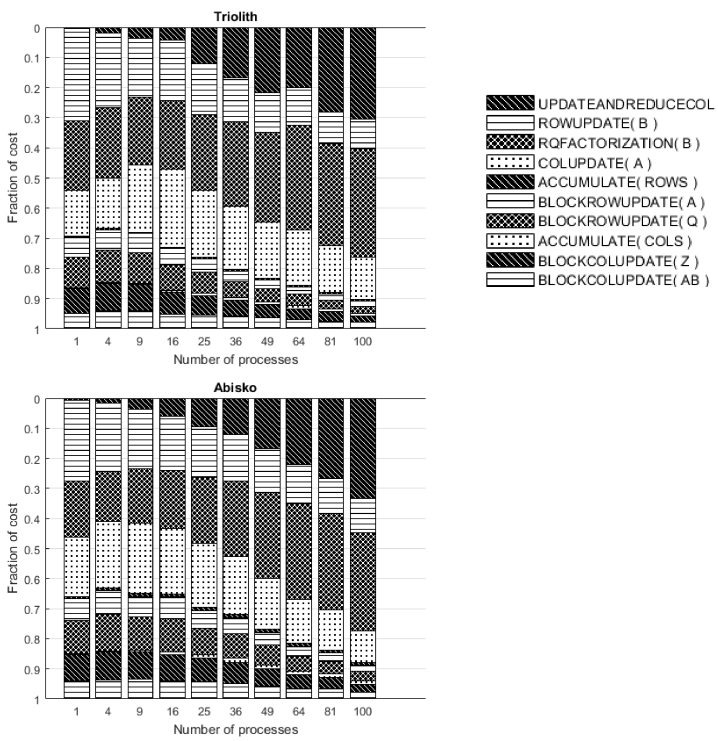

Figure 12: Cost distribution across subroutines for $n = 8000$ on various process meshes. The pattern orderings in the bar plots are the same as in the legend.

The *parallel cost* of a parallel algorithm that takes T seconds to execute on P cores is defined as the product PT. The parallel cost of a subroutine is the parallel cost that is attributable to that subroutine. The relative costs of each subroutine for $n = 8000$ on various meshes are shown in Figure 12. The results are qualitatively similar on both machines. Two bottlenecks can be identified from these results. First, the `UPDATEANDREDUCECOLUMN` subroutine, whose cost is almost negligible on one core, accounts for more than 30% of the cost on 100 cores. This can be understood since this part of the computation is barely parallelizable, as explained in Section 5. Second, the `RQFACTORIZATION` subroutine accounts for almost 40% of the cost on 100 cores whereas it accounts for around 20% on one core. This can be understood in part by the larger overhead of the extended wavefront scheduling algorithm (relative to the standard wavefront scheduling algorithm; see Section 7.1) and in part by the additional synchronization and communication overheads inherent in the computation (see Section 7).

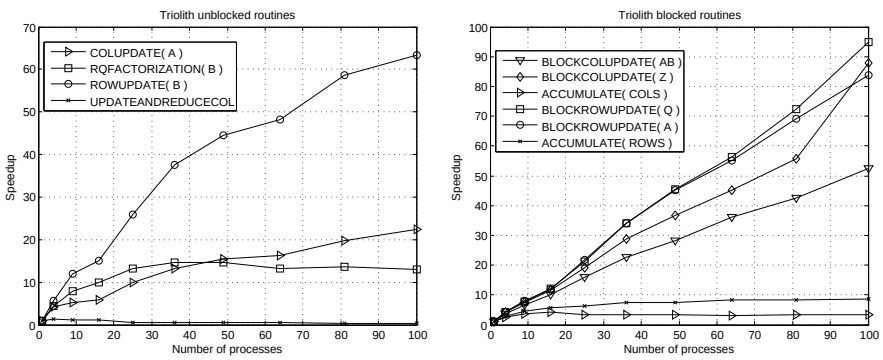

Figure 13: Speedup for each subroutine on Triolith for $n = 8000$.

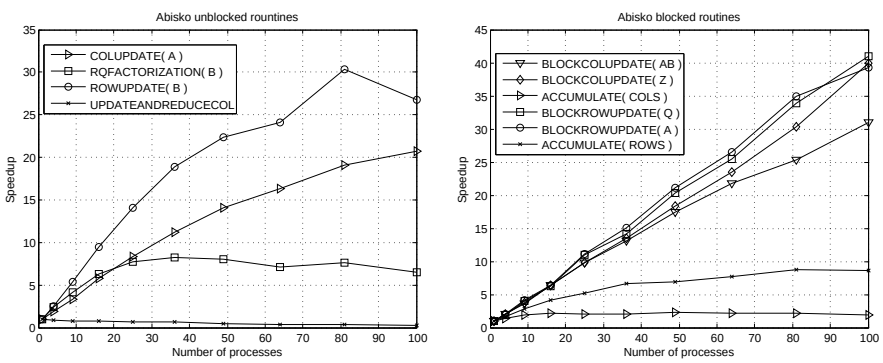

Figure 14: Speedup for each subroutine on Abisko for $n = 8000$.

An alternative view of the profiling data is depicted in Figures 13 and 14. These figures show the relative speedups of the subroutines defined as the ratio of the wall clock time attributable to a subroutine when running on one core to the wall clock time when running the

subroutine on multiple cores. The wall clock times used in these calculations were obtained by dividing the parallel cost of the subroutine with the number of cores. The coarse-grained and highly parallel blocked routines scale essentially linearly on both machines[1]. The unblocked subroutines have poorer scalability, which is especially true for `RQFACTORIZATION` and `UPDATEANDREDUCECOLUMN`. The tool Allinea Map[2] reveals that our parallel algorithm spends 41% of the total time on communication using 16 cores, and the fraction increases to 58% using 36 cores. Looking at the unblocked routines `RQFACTORIZATION` and `UPDATEANDREDUCECOLUMN`, the communication fraction increases from 44% to 76% and from 91% to 94% when going from 16 to 36 cores. For the blocked routine `BLOCKROWUPDATE(A)`, the communication fraction decreases from 38% to 32% when increasing the number of cores from 16 to 36. Thus, the heavy communication in the unblocked routines overshadows the nice scaling properties of the blocked routines.

9.7 The parallel two-stage approach

The sequential two-stage approach can in some cases compete with the sequential (one-stage) `KKQQ` algorithm [11]. However, the parallel two-stage algorithm proposed in [1] do not include the cost-reducing innovations discovered later by [11]. In addition, the complexity of the two-stage approach and the extra arithmetic operations leads to poor performance and scalability. Specifically, running the parallel two-stage algorithm [1] on the weak scaling experiment as in Section 9.5 requires a problem of size $n = 5000$ and a 5×5 mesh before any speedup over sequential `KKQQ` can be observed. The first stage takes less time than the second, despite the workload ratio being in favor of the second stage. The scheduling ideas presented in this paper can potentially be adapted to the two-stage approach.

10 Conclusion

We proposed a novel wavefront scheduling algorithm capable of scheduling sequences of rotations or general transformations on matrices distributed with a two-dimensional block-cyclic distribution. We applied the scheduling algorithm to several parts of a distributed Hessenberg-triangular reduction algorithm and obtained a new formulation of Hessenberg-triangular reduction for distributed memory machines. Experiments show that the parallel implementation of the distributed HT-reduction algorithm is scalable, and proves to be good alternative to the parallel two-staged approach. Its performance is however limited by the scalability of two of the major subroutines. To significantly improve the results, both of these bottlenecks need to be addressed.

The difference of using Householder reflections in the Hessenberg reduction and sequences of Givens rotations in the Hessenberg-triangular reduction has far-reaching consequences. Applying a Householder reflection onto an $n \times n$ matrix involves $\Theta(n^2)$ operations and can use up to $p = n^2$ cores. The communication necessary would be one reduction per row or column of the matrix. With the maximum number of cores and a tree-based reduction, the communication cost would be $(t_s + t_w) \log_2 p$, where t_s is the latency and t_w the inverse bandwidth. On the other hand, applying a sequence of Givens rotations onto an $n \times n$

[1] The multicore processor on Abisko is designed such that each FPU is shared by two cores. When running on 100 cores, there are only 50 FPUs available. This is a potential explanation for why the speedup on Abisko is less than 50 on 100 cores.

[2] http://www.allinea.com/products/map

matrix also involves $\Theta(n^2)$ operations but can use only up to $p = n$ cores. The amount of communication overhead depends on the chosen data distribution and rotation sequences are applied from both sides in the Hessenberg-triangular reduction, which implies that the distribution needs to strike a balance between the costs of the two cases, i.e. updates from left and right. The communication overhead per row or column of the matrix is proportional to the number of distribution block boundaries that need to be traversed. All of this is on top of the added complexity of the wavefront scheduling algorithm that is necessary to keep all cores busy. All said and done, these factors explain some of the difficulties in obtaining a scalable Hessenberg-triangular reduction implementation.

Acknowledgments

We thank the High Performance Computing Center North (HPC2N) at Umeå and National Supercomputer Centre (NSC) at Linköping for providing computational resources and valuable support during test and performance runs. Partial support has been received from the European Unions Horizon 2020 research and innovation programme under the NLAFET grant agreement No 671633, the Swedish Research Council (VR) under grant A0581501, and by eSSENCE, a strategic collaborative e-Science programme funded by the Swedish Government via VR.

References

[1] B. Adlerborn, K. Dackland, and B. Kågström. Parallel two-stage reduction of a regular matrix pair to Hessenberg-Triangular form. In T. Sørevik, F. Manne, A. H. Gebremedhin, and R. Moe, editors, *Applied Parallel Computing, PARA 2000*, LNCS 1947, pages 92–102. Springer Berlin Heidelberg, 2000.

[2] 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, J. Haataja, J. Järvinen, M. Lyly, P. Råback, and V. Savolainen, editors, *Applied Parallel Computing, PARA 2002*, LNCS 2367, pages 319–328. Springer-Verlag, 2002.

[3] B. Adlerborn, B. Kågström, and D. Kressner. Parallel variants of the multishift QZ algorithm with advanced deflation techniques. In B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, editors, *Applied Parallel Computing, PARA 2006*, LNCS 4699, pages 117–126. Springer Berlin Heidelberg, 2006.

[4] B. Adlerborn, B. Kågström, and D. Kressner. A Parallel QZ Algorithm for distributed memory HPC-systems. *SIAM J. Sci. Comput.*, 36(5):C480–C503, 2014.

[5] C Bischof. A summary of block schemes for reducing a general matrix to Hessenberg form. Technical report ANL/MSC-TM-175, Argonne National Laboratory, 1993.

[6] C. H. Bischof and C. F. Van Loan. The *WY* representation for products of Householder matrices. *SIAM J. Sci. Statist. Comput.*, 8(1):S2–S13, 1987.

[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] K. Dackland and B. Kågström. A ScaLAPACK-Style Algorithm for Reducing a Regular Matrix Pair to Block Hessenberg-Triangular Form. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Applied Parallel Computing, PARA 1998*, LNCS 1541, pages 95–103. Springer Berlin Heidelberg, 1998.

[9] 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.

[10] 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.

[11] 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.

[12] B. Lang. Using Level 3 BLAS in Rotation-Based Algorithms. *SIAM J. Sci. Comput.*, 19(2):626–634, 1998.

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

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

III

Paper III

PDHGEQZ User Guide

Björn Adlerborn, Bo Kågström, and Daniel Kressner

Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden
{adler,bokg}@cs.umu.se
SB–MATHICSE–ANCHP, EPF Lausanne, Station 8, CH-1015 Lausanne, Switzerland
daniel.kressner@epfl.ch

Abstract: Given a general matrix pair (A,B) with real entries, we provide software routines for computing a generalized Schur decomposition (S,T). The real and complex conjugate pairs of eigenvalues appears as 1×1 and 2×2 blocks, respectively, along the diagonals of (S,T) and can be reordered in any order. Typically, this functionality is used to compute orthogonal bases for a pair of deflating subspaces corresponding to a selected set of eigenvalues. The routines are written in Fortran 90 and targets distributed memory machines.

PDHGEQZ User Guide*

Björn Adlerborn† Bo Kågström† Daniel Kressner‡

Abstract

Given a general matrix pair (A, B) with real entries, we provide software routines for computing a generalized Schur decomposition (S, T). The real and complex conjugate pairs of eigenvalues appear as 1×1 and 2×2 blocks, respectively, along the diagonals of (S, T) and can be reordered in any order. Typically, this functionality is used to compute orthogonal bases for a pair of deflating subspaces corresponding to a selected set of eigenvalues. The routines are written in Fortran 90 and targets distributed memory machines.

1 Introduction

`PDHGEQZ` is a parallel ScaLAPACK-style [6] package of routines for solving nonsymmetric real generalized eigenvalue problems. The package is written in Fortran 90 and targets distributed memory HPC systems. Using the, small and tightly coupled bulge, multishift QZ algorithm with aggressive early deflation, it computes the generalized Schur decomposition $(S, T_2) = (Q^T H Z, Q^T T_1 Z)$ of an upper Hessenberg matrix H, upper triangular matrix T_1, where $(H, T_1) \in \mathbb{R}^{n \times n}$, such that Q and Z are orthogonal, S is quasi-upper triangular, and T_2 is upper triangular. This document concerns the usage of `PDHGEQZ` and is a supplement to the article [3]. For the description of the algorithm and implementation, we refer to [3] and the references therein (especially, [2, 7, 11, 12, 13]). We have also included a parallel version of the initial Hessenberg-triangular reduction [1], as well as a routine for parallel reordering of eigenvalues of a matrix pair in generalized real Schur form [9]. Moreover, a preliminary version of a serial multishift QZ algorithm with aggressive early deflation is included, see [11], which is used internally within `PDHGEQZ` in favour of the LAPACK [4] routine `DHGEQZ`.

2 Installation

The routines have successfully been tested on both Windows and Linux-like machines. In the following, a Linux-like installation is assumed. There are no pre-built libraries available, so the user must download and build the package before using the software.

Report UMINF 15.12, Dept. of Computing Science, Umeå University, Sweden. This research was conducted using the resources of the High Performance Computing Center North (HPC2N). Financial support has been provided 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.

†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`)

2.1 Prerequisites

To build the library the following is required:

- Fortran 90/95 compiler, or later.
- MPI library, for example, OpenMPI or Intel-MPI.
- BLAS library, for example, OpenBLAS or ACML.
- The LAPACK library, version 3.4.0 or later.
- The ScaLAPACK library, version 2.0.1 or later where BLACS and PBLAS are included.

2.2 Building the package

2.2.1 Download location

The latest version of `PDHGEQZ` can be obtained via the package homepage[1] in a `tar.gz` format with the name `pdhgeqz_latest`.

2.2.2 Structure of the package

After downloading and moved to a proper location, the package is expanded by issuing

```
tar xvfz pdhgeqz_latest.tar.gz
```

The unpacked package is structured in the following way:

- `examples/` This folder contains three different simple drivers:
  - `EXRAND1.f`: Generates a Hessenberg-Triangular problem, *Hessrand1* [3], and computes its generalized real Schur form.
  - `EXRAND2.f`: Generates a random matrix pair (A, B) where the entries are uniformly distributed in the interval $[0, 1]$. The problem is reduced to Hessenberg-triangular form using a QR factorisation of B, producing an upper triangular T, followed by calling `PDGGHRD` which further reduces the (A, T) pair to the desired Hessenberg-triangular form. Once there, we apply our parallel QZ algorithm to compute the generalized real Schur form and eigenvalues.
  - `EXRAND3.f`: This example reads two matrices A and B from two files using the Matrix Market file format [5]. The matrices A and B are stored in the files `mhd4800a.mtx` and `mhd4800b.mtx` respectively, and can be downloaded, with other benchmarks, at the Matrix Market homepage[2]. A and B are treated as dense, although they are sparse ($< 1\%$ nonzero entries), and first reduced to Hessenberg-triangular form before the parallel QZ algorithm is applied, as described for the example `EXRAND2.f`. After the generalized real Schur form is obtained, a reordering of the eigenvalues is performed such that all eigenvalues within the unit circle are moved to the top of the matrix pair, using the routine `PDTGORD`.

[1] PDHGEQZ homepage: https://archive.cs.umu.se/software/pdhgeqz/
[2] http://math.nist.gov/MatrixMarket

- `make_inc/` This folder contains various templates of `make.inc` files for different compiler setups.
- `Makefile` This is the default Makefile for building the `PDHGEQZ` package, which generally does not need to be modified. This Makefile will call the Makefiles stored in subdirectories. After a successful build, the library will be stored in the root folder as the file `libpdhgeqz.a`.
- `make.inc` File included by the Makefile. Should be modified to match the compiler setup.
- `readme` A shorter version of this Users' Guide. Intended as a quick installation quide.
- `src/` This folder contains all source files, in Fortran format, related to the `PDHGEQZ` software, stored in four different subdirectories:
  - `ab2ht/` Source files for the parallel Hessenberg-triangular reduction.
  - `kkqz/` Source files for the sequential QZ solver with multishift and aggressive early deflation.
  - `reorder/` Source files for the parallel generalized eigenvalue reordering.
  - `pdhgehz/` Main source files for the parallel QZ solver with multishifts and aggressive early deflation.
- `testing/` This folder contains test routines.
- `tools/` This folder contains various auxiliary routines, used for generation of matrices, input/output, etc.
- `userguide.pdf` A copy of this User Guide.

2.2.3 Compiling and building the `PDHGEQZ` package

By issuing the command

`make all` - this is the same as issuing `make`,

the library `libpdhgeqz.a` should be compiled, linked, and stored in the root folder. The three examples found in the `examples/` folder as well as the test routine stored in the `testing/` folder will also be built by this command. If only the library is desired, instead issue the command

```
make lib
```

which will compile, link and store the library in the root folder.

During `make all` some quick tests will be performed and hopefully the following will be displayed on the console:

```
% 5 tests out of 5 passed.
```

The script file `runmpi.sh` in the `testing/` folder might need some attention, depending upon what MPI installation the user has. The default MPI execution is `mpirun`.

2.3 Running tests

The three examples provided contain small problems, i.e. $n < 6000$, and should execute rather quickly. The problem size is governed by the parameter `N`. Change this, in file `EXRAND1.f` or `EXRAND2.f`, and rebuild, to run a different sized problem. The example problems can be executed serially, or in parallel with some MPI tool, for example `mpirun` or `mpiexec`.

3 Using the PDHGEQZ library

3.1 ScaLAPACK data layout convention

We follow the convention of ScaLAPACK for the distributed data storage of matrices. Suppose that $P = P_r \cdot P_c$ parallel processors are arranged in a $P_r \times P_c$ 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 to avoid too many special cases in the code, square blocks are assumed, i.e. $m_b = n_b$.

3.2 Software hierarchy

Figure 1 shows an overview of the main routines related to our parallel QZ software, which are based upon routines provided by ScaLAPACK, BLACS, PBLAS, LAPACK, BLAS and MPI. One-directed arrows indicate that one routine calls other routines. For example, `PDHGEQZ1` is called by `PDHGEQZ`, `PDHGEQZ0` and `PDHGEQZ3` and calls four routines: `PDHGEQZ2`, `PDHGEQZ4`, `PDHGEQZ5`, and `PDHGEQZ7`. `PDHGEQZ3` will call `PDHGEQZ0` instead of `PDHGEQZ1` recursively if the AED window is large, i.e. $n_{AED} > 6000$, indicated with the double directed arrow between `PDHGEQZ0` and `PDHGEQZ3`. To keep the stack from growing uncontrollably, due to static allocations, the level of recursive calls are limited; this version only supports one level of recursion. Main entry routine is `PDHGEQZ`, see Figure 2 for an interface description. Depending upon problem size n, `PDHGEQZ` calls `PDHGEQZ0` or `PDHGEQZ1` to perform the reduction; see section 4 and parameter n_{min1}. Table 1 gives a brief description of the routines displayed in Figure 1.

3.3 Calling sequence

The main purpose is to compute the generalised real Schur decomposition of a matrix pair in (upper)Hessenberg-triangular form using the routine `PDHGEQZ`. As a pre-processing step the general square matrix pair must be transformed to Hessenberg-triangular form using the supplied routine `PDGGHRD`. `PDGGHRD` used in conjunction with `PDHGEQZ` are exemplified in the routines found in the folder `examples/`. As a post-processing step, the supplied routine `PDTGORD` can be used to reorder the eigenvalues returned by `PDHGEQZ`. Below follows call sequences for these routines. As previously stated, both `PDGGHRD` and `PDTGORD` are preliminary.

3.3.1 PDHGEQZ

The interface for `PDHGEQZ` is similar to the existing (serial) LAPACK routine `DHGEQZ`, see Figure 2. The main difference between `PDHGEQZ` and `DHGEQZ` is the use of descriptors to define

[3] See Table 2 for `PDHGEQZ` parameters.

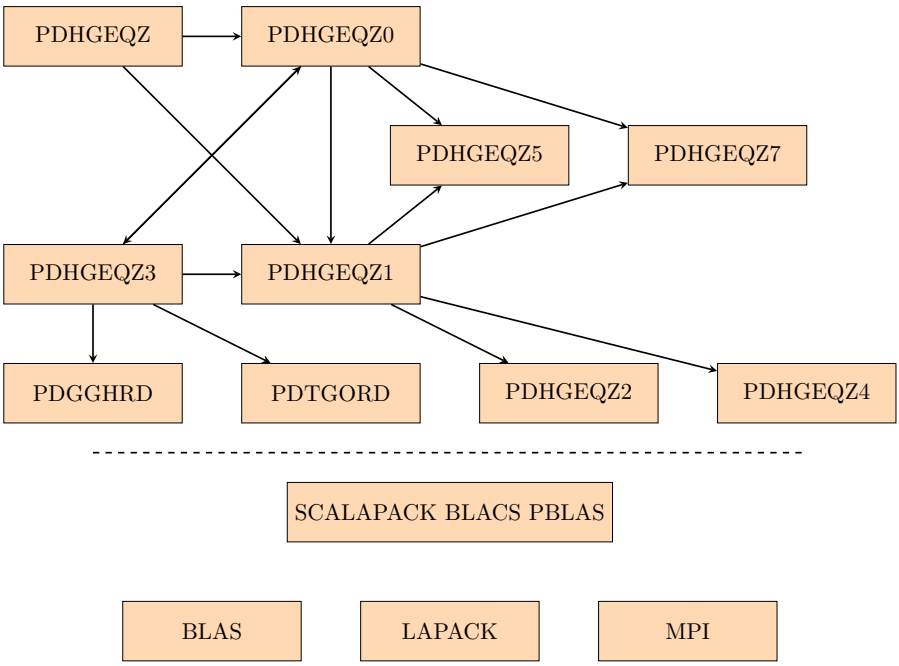

Figure 1: Software hierarchy for `PDGEHQZ`.

partitioning and the globally distributed matrices across the $P_r \times P_c$ process grid, instead of leading dimensions, and that `PDHGEQZ` requires an integer workspace.

Below follows a list and description of the arguments:

- `JOB` (global input) `CHARACTER*1`.
  `= 'E'`: compute only the eigenvalues represented by `ALPHAR`, `ALPHAI`, and `BETA`. The generalized Schur form of (H,T) will not be computed.
  `= 'S'`: compute the generalized Schur form of (H,T) as well as the eigenvalues represented by `ALPHAR`, `ALPHAI`, and `BETA`.

- `COMPQ` (global input) `CHARACTER*1`.
  `= 'N'`: Left generalized Schur vectors (i.e., Q) are not computed.
  `= 'I'`: Q is initialized to the unit matrix, and the orthogonal matrix Q is returned.
  `= 'V'`: Q must contain an orthogonal matrix $Q1$ on entry, and the product $Q1 \cdot Q$ is returned.

- `COMPZ` (global input) `CHARACTER*1`.
  `= 'N'`: Right generalized Schur vectors (i.e., Z) are not computed.
  `= 'I'`: Z is initialized to the unit matrix, and the orthogonal matrix Z is returned.
  `= 'V'`: Z must contain an orthogonal matrix $Z1$ on entry, and the product $Z1 \cdot Z$ is returned.

Table 1: PDHGEQZ routines.

Routine	Description
PDHGEQZ	Main entry routine. Calls PDHGEQZ0 or PDHGEQZ1 depending upon problem size.
PDHGEQZ0	Multishift Parallel QZ with AED. Tuned for larger problems and might do recursive calls when doing AED. Calls PDHGEQZ1 when problem size is split into smaller subproblems.
PDHGEQZ1	Multishift Parallel QZ with sequentially performed AED. Tuned for smaller problems.
PDHGEQZ2	Performs sequential AED, while off-diagonal blocks, outside the AED window, are updated in parallel.
PDHGEQZ3	Performs parallel AED and shift computation.
PDHGEQZ4	Performs sequential multishift QZ with AED, calls KKQZ for this purpose. Called by PDHGEQZ1 when a subproblem, of order less than $n_{\min 3}{}^3$, has been identified. Off-diagonal blocks, outside the subproblem, are updated in parallel.
PDHGEQZ5	Parallel multishift QZ iteration based on chains of tightly coupled bulges.
PDHGEQZ7	Parallel identification and deflation of infinite eigenvalues.
PDTGORD	Reorders a cluster of eigenvalues to the top of the matrix pair (S, T) in generalized real Schur form.
PDGGHRD	Parallel Hessenberg-triangular reduction, used by PDHGEQZ3 to restore the matrix pair to Hessenberg-triangular form.

- N (global input) INTEGER
  The order of the $N \times N$ matrices H, T, Q, and Z.

- ILO, IHI
  It is assumed that H and T is already in upper triangular form in rows and columns $(1 : \mathtt{ILO}-1)$ and $(\mathtt{IHI}+1 : \mathtt{N})$. $1 \leq \mathtt{ILO} \leq \mathtt{IHI} \leq \mathtt{N}$, if $\mathtt{N} > 0$; $\mathtt{ILO} = 1$ and $\mathtt{IHI} = 0$, if $\mathtt{N} = 0$.

- H (global input/output) DOUBLE PRECISION array of dimension (DESCH(9),*).
  DESCH (global and local input) INTEGER array of dimension 9.
  H and DESCH define the distributed matrix H.
  On entry, H contains the upper Hessenberg matrix H. On exit, if JOB = 'S', H is quasi-upper triangular in rows and columns (ILO : IHI), with 1×1 and 2×2 blocks on the diagonal where the 2×2 blocks correspond to complex conjugated pairs of eigenvalues. If JOB='E', H is unspecified on exit.

- T (global input/output) DOUBLE PRECISION array of dimension (DESCT(9),*).
  DESCT (global and local input) INTEGER array of dimension 9.
  T and DESCT define the distributed matrix T.
  On entry, T contains the upper triangular matrix T. If JOB='E', T is unspecified on exit, otherwise T is on exit overwritten by another upper triangular matrix $Q^T \cdot T \cdot Z$.

- ALPHAR (global output) DOUBLE PRECISION array, dimension N
  ALPHAI (global output) DOUBLE PRECISION array, dimension N
  BETA (global output) DOUBLE PRECISION array, dimension N
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), $j = 1, ..., N$, will be the generalized eigenvalues. ALPHAR(j) + ALPHAI(j)*i and BETA(j), $j = 1, ..., N$ are the diagonals of the complex Schur form (H,T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (H,T) were further reduced to triangular form using complex unitary transformations. If ALPHAI(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with ALPHAI(j+1) negative.

```
      SUBROUTINE PDHGEQZ( JOB, COMPQ, COMPZ,
     $      N, ILO, IHI, H, DESCH, T, DESCT,
     $      ALPHAR, ALPHAI, BETA, Q, DESCQ, Z, DESCZ,
     $      WORK, LWORK, IWORK, LIWORK, INFO )

*     ..
*     .. Scalar Arguments ..
*     ..
      CHARACTER          COMPQ, COMPZ, JOB
      INTEGER            IHI, ILO, INFO, N
      INTEGER            LWORK, LIWORK
*     ..
*     .. Array Arguments ..
*     ..
      DOUBLE PRECISION   H(*), T(*), Q(*), Z(*)
      DOUBLE PRECISION   WORK(*), ALPHAI(*), ALPHAR(*), BETA(*)
      INTEGER            IWORK(*), DESCH(9), DESCT(9), DESCQ(9), DESCZ(9)
```

Figure 2: Interface for PDGEHQZ

- Q (global input/output) DOUBLE PRECISION array of dimension (DESCQ(9),*).
  DESCQ (global and local input) INTEGER array of dimension 9. Q and DESCQ define the distributed matrix Q.
  If COMPQ = 'N', Q is not referenced.
  If COMPQ = 'I', Q is initialized to the unit matrix, and on exit it contains the orthogonal matrix Q, where Q^T is the product of the transformations which are applied to the left hand side of H and T.
  If COMPQ = 'V', on entry, Q must contain an orthogonal matrix $Q1$, and on exit this is overwritten by $Q1 \cdot Q$, where Q^T is the product of the transformations which are applied to the left hand side of H and T.

- Z (global input/output) DOUBLE PRECISION array of dimension (DESCZ(9),*).
  DESCZ (global and local input) INTEGER array of dimension 9. Z and DESCZ define the distributed matrix Z.
  If COMPZ = 'N', Z is not referenced.
  If COMPZ = 'I', Z is initialized to the unit matrix, and on exit it contains the orthogonal matrix Z, where Z is the product of the transformations which are applied to the right hand side of H and T.
  If COMPZ = 'V', on entry, Z must contain an orthogonal matrix $Z1$, and on exit this is overwritten by $Z1 \cdot Z$, where Z is the product of the transformations which are applied to the right hand side of H and T.

- WORK (local workspace) DOUBLE PRECISION array of dimension LWORK.
  LWORK (global input) INTEGER.
  If LWORK = -1, then a workspace query is assumed and required workspace is returned in WORK(1) and no further computation is performed.

- IWORK (local workspace) INTEGER array of dimension LIWORK.
  LIWORK (global input) INTEGER.
  If LIWORK = -1, then a workspace query is assumed and required workspace is returned

in IWORK(1) and no further computation is performed.

- INFO (global output) INTEGER
  = 0, successful exit.
  < 0, if INFO = -i, the i-th argument had an illegal value.

3.3.2 PDGGHRD

The interface for PDGGHRD is similar to the existing (serial) LAPACK routine DGGHRD, see Figure 3. The main difference between PDGGHRD and DGGHRD is the use of descriptors to define

```
      SUBROUTINE PDGGHRD( COMPQ, COMPZ,
     $        N, ILO, IHI, A, DESCB, B, DESCB,
     $        Q, DESCQ, Z, DESCZ,
     $        WORK, LWORK, INFO )

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

Figure 3: Interface for PDGGHRD

partitioning and the globally distributed matrices across the $P_r \times P_c$ process grid, instead of leading dimensions.

Below follows a list and description of the arguments:

- COMPQ (global input) CHARACTER*1.
  = 'N': Do not compute Q.
  = 'I': Q is initialized to the unit matrix, and the orthogonal matrix Q is returned.
  = 'V': Q must contain an orthogonal matrix $Q1$ on entry, and the product $Q1 \cdot Q$ is returned.

- COMPZ (global input) CHARACTER*1.
  = 'N': Do not compute Z.
  = 'I': Z is initialized to the unit matrix, and the orthogonal matrix Z is returned.
  = 'V': Z must contain an orthogonal matrix $Z1$ on entry, and the product $Z1 \cdot Z$ is returned.

- N (global input) INTEGER
  The order of the $N \times N$ matrices A, B, Q, and Z.

- ILO, IHI (global input) INTEGER
  It is assumed that A and B is already in upper triangular form in rows and columns $(1 : \mathtt{ILO}-1)$ and $(\mathtt{IHI}+1 : \mathtt{N})$. $1 \leq \mathtt{ILO} \leq \mathtt{IHI} \leq \mathtt{N}$, if $\mathtt{N} > 0$; $\mathtt{ILO} = 1$ and $\mathtt{IHI} = 0$, if $\mathtt{N} = 0$.

- A (global input/output) DOUBLE PRECISION array of dimension (DESCA(9),*).
  DESCA (global and local input) INTEGER array of dimension 9.
  A and DESCA define the distributed matrix A.
  On entry, the square general matrix A to be reduced. On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix $H = Q \cdot A \cdot Z$, and the remaining elements are set to zero.

- B (global input/output) DOUBLE PRECISION array of dimension (DESCB(9),*).
  DESCB (global and local input) INTEGER array of dimension 9.
  B and DESCB define the distributed matrix B.
  On entry, the square upper triangular matrix B. On exit, overwritten by the upper triangular matrix $T = Q \cdot B \cdot Z$. The elements below the diagonal are set to zero.

- Q (global input/output) DOUBLE PRECISION array of dimension (DESCQ(9),*).
  DESCQ (global and local input) INTEGER array of dimension 9. Q and DESCQ define the distributed matrix Q^T.
  If COMPQ = 'N', Q is not referenced.
  If COMPQ = 'I', Q is initialized to the unit matrix, and on exit it contains the orthogonal matrix Q^T, where Q is the product of the transformations which are applied to the left hand side of A and B.
  If COMPQ = 'V', on entry, Q must contain an orthogonal matrix $Q1$, and on exit this is overwritten by $Q1 \cdot Q^T$, where Q is the product of the transformations which are applied to the left hand side of A and B.

- Z (global input/output) DOUBLE PRECISION array of dimension (DESCZ(9),*).
  DESCZ (global and local input) INTEGER array of dimension 9. Z and DESCZ define the distributed matrix Z.
  If COMPZ = 'N', Z is not referenced.
  If COMPZ = 'I', Z is initialized to the unit matrix, and on exit it contains the orthogonal matrix Z, where Z is the product of the transformations which are applied to the right hand side of A and B.
  If COMPZ = 'V', on entry, Z must contain an orthogonal matrix $Z1$, and on exit this is overwritten by $Z1 \cdot Z$, where Z is the product of the transformations which are applied to the right hand side of A and B.

- WORK (local workspace) DOUBLE PRECISION array of dimension LWORK.
  LWORK (global input) INTEGER.
  If LWORK = -1, then a workspace query is assumed and required workspace is returned in WORK(1) and no further computation is performed.

- INFO (global output) INTEGER
  = 0, successful exit.
  $<$ 0, if INFO = -i, the i-th argument had an illegal value.

3.3.3 PDTGORD

The interface for PDTGORD is displayed in Figure 4.

Below follows a list and description of the arguments:

```
      SUBROUTINE PDTGORD( WANTQ, WANTZ,
     $        SEL, PARA, N,
     $        S, DESCS, T, DESCT,
     $        Q, DESCQ, Z, DESCZ,
     $        ALPHAR, ALPHAI, BETA,
     $        M, DWORK, LDWORK,
     $        IWORK, LIWORK, INFO )

*       ..
*       .. Scalar Arguments ..
*       ..
        LOGICAL              WANTQ, WANTZ
        INTEGER              INFO, LIWORK, LDWORK, M, N
*       ..
*       .. Array Arguments ..
*       ..
        INTEGER              SEL( * ), IWORK( * )
        INTEGER              PARA( 6 ), DESCS( 9 ), DESCT( 9 ), DESCQ( 9 ), DESCZ( 9 )
        DOUBLE PRECISION     S(*), T(*), Q(*), Z(*)
        DOUBLE PRECISION     DWORK(*), BETA( * ), ALPHAI( * ), ALPHAR( * )
```

Figure 4: Interface for `PDTGORD`

- `WANTQ` (global input) `LOGICAL`.
  `= .TRUE.`, Update $\mathtt{Q}^T$, with all transformations applied from left hand side to `S` and `T`.
  `= .FALSE.`, `Q` is not referenced.

- `WANTZ` (global input) `LOGICAL`.
  `= .TRUE.`, Update `Z`, with all transformations applied from right hand side to `S` and `T`.
  `= .FALSE.`, `Z` is not referenced.

- `SEL` (global input/output) INTEGER array, dimension `N`.
  `SEL` specifies the eigenvalues in the selected cluster. To select a real eigenvalue $w(j)$, `SEL(j)` must be set to 1. To select a complex conjugate pair of eigenvalues $w(j)$ and $w(j+1)$, corresponding to a 2-by-2 diagonal block, must be set to 1; On output, `SEL` is updated to reflect the performed reordering.

- `PARA` (global input) `INTEGER` array of dimension 6
  `PARA(1)` = maximum number of concurrent computational windows allowed in the algorithm. `0` $<$ `PARA(1)` $\leq min(P_\mathrm{r}, P_\mathrm{c})$ must hold.
  `PARA(2)` = number of eigenvalues in each computational window. `0` $<$ `PARA(2)` $\leq$ `PARA(3)` must hold.
  `PARA(3)` = size of computational window. `PARA(2)` $\leq$ `PARA(3)` $\leq n_\mathrm{b}$ must hold.
  `PARA(4)` = minimal percentage of flops required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations. `0` $\leq$ `PARA(4)` $\leq$ `100` must hold.
  `PARA(5)` = width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form. `0` $<$ `PARA(5)` $\leq n_\mathrm{b}$ must hold.
  `PARA(6)` = the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size. `0` $<$ `PARA(6)` $\leq$ `PARA(2)` must hold.

- N (global input) INTEGER.
  The order of the $N \times N$ matrices S, T, Q, and Z.

- S (global input/output) DOUBLE PRECISION array of dimension (DESCS(9),*).
  DESCS (global and local input) INTEGER array of dimension 9.
  S and DESCS define the distributed matrix S. On entry, the global distributed upper quasi-triangular matrix S, in Schur form. On exit, S is overwritten by the reordered matrix S, again in Schur form, with the selected eigenvalues in the globally leading diagonal blocks of (S, T).

- T (global input/output) DOUBLE PRECISION array of dimension (DESCT(9),*).
  DESCT (global and local input) INTEGER array of dimension 9.
  T and DESCT define the distributed matrix T. On entry, the global distributed upper triangular matrix T. On exit, T is overwritten by the reordered matrix T, again in upper triangular form, with the selected eigenvalues in the globally leading diagonal blocks of (S, T).

- Q (global input/output) DOUBLE PRECISION array of dimension (DESCQ(9),*).
  DESCQ (global and local input) INTEGER array of dimension 9.
  Q and DESCQ define the distributed matrix Q. On entry, if WANTQ = .TRUE., the global distributed matrix Q of left generalized Schur vectors. On exit, WANTQ = .TRUE., Q has been postmultiplied by the global orthogonal transformation matrix, applied from the left, which reorders the matrix pair (S, T); the leading M columns of Q form left orthonormal bases for the specified deflating subspaces. If WANTQ = .FALSE., Q is not referenced.

- Z (global input/output) DOUBLE PRECISION array of dimension (DESCZ(9),*).
  DESCZ (global and local input) INTEGER array of dimension 9.
  Z and DESCZ define the distributed matrix Z. On entry, if WANTZ = .TRUE., the global distributed matrix Z of generalized right Schur vectors. On exit, WANTZ = .TRUE., Z has been postmultiplied by the global orthogonal transformation matrix, applied from the right, which reorders (S, T); the leading M columns of Z form right orthonormal bases for the specified deflating subspaces. If WANTZ = .FALSE., Z is not referenced.

- ALPHAR (global output) DOUBLE PRECISION array, dimension N
  ALPHAI (global output) DOUBLE PRECISION array, dimension N
  BETA (global output) DOUBLE PRECISION array, dimension N
  On exit, (ALPHAR(j) + ALPHAI(j)*i)/BETA(j), $j = 1, ..., N$, will be the generalized eigenvalues. ALPHAR(j) + ALPHAI(j)*i and BETA(j), $j = 1, ..., N$ are the diagonals of the complex Schur form (S,T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (S,T) were further reduced to triangular form using complex unitary transformations. If ALPHAI(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with ALPHAI(j+1) negative.

- M (global output) INTEGER.
  The dimension of the specified pair of left and right eigenspaces (deflating subspaces). 0 $\leq$ M $\leq$ N. If M = 0, then no eigenvalues have been reordered.

- DWORK (local workspace) DOUBLE PRECISION array of dimension LDWORK.
  LDWORK (global input) INTEGER.
  If LDWORK = -1, then a workspace query is assumed and required workspace is returned in DWORK(1) and no further computation is performed.

- IWORK (local workspace) INTEGER array of dimension LIWORK.
  LIWORK (global input) INTEGER.
  If LIWORK = -1, then a workspace query is assumed and required workspace is returned in IWORK(1) and no further computation is performed.

- INFO (global output) INTEGER.
  = 0, successful exit.
  $<$ 0, if INFO = -i, the i-th argument had an illegal value. If the i-th argument is an array and the j-entry had an illegal value, then INFO = -(i*1000+j).

4 Parameters related to PDHGEQZ and subroutines

PDHGEQZ offers tuning of machine dependent parameters for experienced users. However, there is always a default value provided, and these settings should provide reasonable performance for machines similar to those we have performed our performance-runs on, see [3].

Many of these parameters are dependent upon the value of n_{b}, which the end user chooses a value for when setting up descriptors for the distributed matrices. For a suitable value of n_{b} correlated to the problem size n, see [3] Section 3.2. The parameters are stored in the files piparmq.f and pilanvx.f, found in the folder src/pdhgeqz/. PILAENVX(ISPEC = 50...56) and PILAENVX(ISPEC = 80...85) are parameters related to PDHGEQZ, listed and described in Table 2.

5 Terms of usage

The PDHGEQZ library is freely available for academic (non-commercial) use, and is provided on an "as is" basis. Any use of the PDHGEQZ library should be acknowledged by citing paper [3] and this User Guide.

6 Conclusions and future work

We have presented the high performance software package PDHGEQZ. The latest version of the package along with updated information and documentation will always be available for download from the PDHGEQZ website. We welcome bug-reports, comments and suggestions from users.

Acknowledgments

The authors are grateful to Lars Karlsson, and Meiyue Shao for helpful discussions on parallel QZ algorithms. We thank Åke Sandgren and the rest of the group at the High Performance Computing Center North (HPC2N) for providing computational resources and valuable support during test and performance runs.

Table 2: Tunable parameters for PDHGEQZ

ISPEC	Name	Description	Default value
50	$n_{\min 1}$	Threshold for when to choose PDHGEQZ0 instead of PDHGEQZ1; matrices of order less than this value are reduced by PDHGEQZ1.	6000
51	n_{AED}	Size of the aggressive early deflation window.	see [3] Section 3.2
52	NIBBLE	Threshold for when to skip repeated AED and instead do a multishift QZ sweep.	see [10] Section 3.4
53	n_{shift}	The number of simultaneous shifts in a multishift QZ iteration (in PDHGEQZ0).	see [3] Section 3.2
54	$n_{\min 2}$	When current problem size is less than this value, PDHGEQZ0 calls PDHGEQZ1 to perform the remaining reduction.	201
55	$n_{\min 3}$	When current problem size is less than this value PDHGEQZ1 stops executing and instead performs the remaining reduction serially.	201
56	P_{AED}	Number of processes to use when performing parallel AED.	see [3] Section 3.2
80	NUMWIN	Maximum number of concurrent computational windows (for parallel reordering only).	$min(p_{\mathrm{r}}, p_{\mathrm{c}}, n/n_{\mathrm{b}})$
81	WINEIG	Number of eigenvalues/bulges in each window (for parallel reordering only).	$n_{\mathrm{b}}/2$
82	WINSIZE	Computational window size (for parallel reordering only).	n_{b}
83	MMULT	Minimal percentage of flops required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations (for parallel reordering only).	0. Throughout our tests, the matrix-matrix multiplications were faster than or equivalent to pipelined transformations. This is probably caused by the small problem sizes we are running on, i.e. the size of the AED windown. For larger n and thereby larger n_{AED}, the value for MMULT might need fine tuning. However, a value of 0 will always work and will probably give good enough performance.
84	NCB	Width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form (for parallel reordering only).	$\min(n_{\mathrm{b}}, 32)$
85	WNEICR	The maximum number of eigenvalues moved together over a process border (for parallel reordering only).	same as WINEIG

References

[1] B. Adlerborn, L. Karlsson, and B. Kågström. Distributed One-Stage Hessenberg-Triangular Reduction with Wavefront Scheduling. *Report UMINF* 16.10, Dept. of Computing Science, Umeå University, Sweden, 2016.

[2] B. Adlerborn, B. Kågström, and D. Kressner. Parallel variants of the multishift QZ algorithm with advanced deflation techniques. In B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, editors, *Applied Parallel Computing, PARA 2006*, LNCS 4699, pages 117–126. Springer Berlin Heidelberg, 2006.

[3] B. Adlerborn, B. Kågström, and D. Kressner. A Parallel QZ Algorithm for distributed memory HPC-systems. *SIAM J. Sci. Comput.*, 36(5):C480–C503, 2014.

[4] 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.

[5] 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`.

[6] 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.

[7] 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.

[8] R. Granat and B. Kågström. Parallel solvers for Sylvester-type matrix equations with applications in condition estimation, Part II. *ACM Trans. Math. Software*, 37(3), 2007.

[9] 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.

[10] R. Granat, B. Kågström, D. Kressner, and M. Shao. Parallel library software for the multishift QR algorithm with aggressive early deflation. *ACM Trans. Math. Software*, 41(4), 2015.

[11] 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.

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

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

A List and description of supplied drivers and auxiliary routines.

Beside the routines briefly described in Table 1, we list the other provided routines in Tables 3–7.

Table 3: Routines not listed in Figure 1 but related to, and called by, the main routines of PDHGEQZ.

Routine	Description
PDHGEQZ8	Called by PDHGEQZ7 to perform parallel chase and deflation of infinite eigenvalues at the top left corner of (H, T).
PDHGEQZ9	Called by PDHGEQZ7 to perform parallel chase and deflation of infinite eigenvalues at the bottom right corner of (H, T).
PDHGEQZA	Called by PDHGEQZ5 to create bulges within a diagonal block of (H, T), followed by updates in parallel of off-diagonal elements.
PDHGEQZB	Called by PDHGEQZ5 to chase bulges within a diagonal block of (H, T), followed by updates in parallel of off-diagonal elements.
PDHGEQZ6	Called by PDHGEQZ2, PDHGEQZ4, PDHGEQZ8, PDHGEQZ9, PDHGEQZA, and PDHGEQZB to update off-diagonal entries in parallel.
PDLACP4	Called by PDHGEQZ2, PDHGEQZ4, PDHGEQZ5, PDHGEQZ8 and PDHGEQZ9 to copy a global diagonal block of (H, T) to a local workspace copy, or vice versa.
PDROT	Performs a planar rotation, in parallel.
DHGEQZ5	Serial chase of bulges within a diagonal block of (H, T)
DHGEQZ7	Serial chase of infinite eigenvalues, along the diagonal of T, up or down.

Table 4: Routines related to parallel reordering. These routines are also part of the SCASY library, see [8] and SCASY homepage http://www8.cs.umu.se/∼granat/scasy.html, although slightly modified here

Routine	Description
PDTGSEN	Reorders a cluster of eigenvalues to the top of the matrix pair (S, T) in real generalized Schur form. Also provides functionality to compute condition number estimate for eigenvalues and eigenspaces.
BDLAGPP	Computes a transformation matrix Q resulting from performed swaps of diagonal blocks.
BDTGEXC	Moves a diagonal block from one position to another.
BDTGEX2	Swaps two adjacent diagonal blocks.

Table 5: Routines related to the parallel Hessenberg-triangular reduction.

Routine	Description
PDGGHRD	Parallel reduction of a matrix pair to Hessenberg-triangular form - main routine.
UPDATEANDREDUCECOLUMN	Applies previous row updates and reduces a column.
UPDATEANDREDUCECOLUMN_ROOT	Applies previous row updates and reduces a column. This version uses a single core to perform the updates and reduction and is called internally by UPDATEANDREDUCECOLUMN.
KRNLUPDATEANDREDUCECOLUMN	Applies previous row updates and reduces a column - kernel version.
SLIVERROWUPDATE	Applies row updates.
KRNLROWUPDATE	Applies row updates - kernel version.
SLIVERHESSCOLUMNUPDATE	Reduces a matrix in Hessenberg form to triangular form.
KRNLCOLUMNANNIHILATE	Annihilates sub diagonal entries - kernel version.
SLIVERCOLUMNUPDATE	Applies column updates.
KRNLCOLUMNUPDATE	Applies column updates - kernel version.
ACCUMULATEROWROTATIONS	Accumulates row rotations into transformation matrices.
KRNLACCUMULATEROWROTATIONS	Accumulates row rotations into transformation matrices - kernel version.
ACCUMULATECOLUMNROTATIONS	Accumulates column rotations into transformation matrices.
KRNLACCUMULATECOLUMNROTATIONS	Accumulates column rotations into transformation matrices - kernel version.
BLOCKSLIVERROWUPDATE	Applies accumulated transformation on block rows.
BLOCKSLIVERCOLUMNUPDATE	Applies accumulated transformations on block columns.
DUOBLOCKSLIVERCOLUMNUPDATE	Applies accumulated transformations on block columns. This routine takes two matrices as input and updates them at the same time.
GRN2LRN	Computes a local range from a global range.

Table 6: Routines related to the serial multishift QZ, with aggressive early deflation, KKQZ.

Routine	Description
KKQZ	Serial multishift QZ with aggressive early deflation - main routine.
KKQZCONF	Sets up parameters for the KKQZ routine. This routine needs to be called before any call to PDHGEQZ.
INVHSE	Computes an "inverted" Householder reflector.
QZDACK	Serial and blocked single/double shift QZ, based on [7].
QZDACKIT	Performs a blocked single/double QZ iteration, based on [7].
QZEARLY	Performs aggressive early deflation.
QZFCOL	Forms a multiple of the first column of the shift polynomial for the generalized eigenvalue problem.
QZINF	Identifies and deflates infinite eigenvalues.
QZLAP	Serial single/double shift QZ, based on the LAPACK routine DHGEQZ.
QZLAPIT	Performs a single/double QZ iteration, based on the LAPACK routine DHGEQZ.

Table 7: Routines supplied in the tools/ folder.

Routine	Description
PDMATGEN2	Generates a matrix with entries choosen from a uniform distribtution $\in [0, 1]$.
PDLAPRNT	Prints a distributed matrix.
PQZHELPS	Contains routines for calculating residuals, and misc. short helper routines.
MMIO	Contains misc. routines for reading and writing Matrix Market files.
PDRDMMM	Reads a Matrix Market matrix file and stores the content in a globally distributed matrix.