LAPACK Working Note #216: A novel parallel QR algorithm for hybrid distributed memory HPC systems*

R. Granat[1] Bo Kågström[1] D. Kressner[2]

Abstract

A novel variant of the parallel QR algorithm for solving dense nonsymmetric eigenvalue problems on hybrid distributed high performance computing (HPC) systems is presented. For this purpose, we introduce the concept of multi-window bulge chain chasing and parallelize aggressive early deflation. The multi-window approach ensures that most computations when chasing chains of bulges are performed in level 3 BLAS operations, while the aim of aggressive early deflation is to speed up the convergence of the QR algorithm. Mixed MPI-OpenMP coding techniques are utilized for porting the codes to distributed memory platforms with multithreaded nodes, such as multicore processors. Numerous numerical experiments confirm the superior performance of our parallel QR algorithm in comparison with the existing ScaLAPACK code, leading to an implementation that is one to two orders of magnitude faster for sufficiently large problems, including a number of examples from applications.

Keywords: Eigenvalue problem, nonsymmetric QR algorithm, multishift, bulge chasing, parallel computations, level 3 performance, aggressive early deflation, parallel algorithms, hybrid distributed memory systems.

1 Introduction

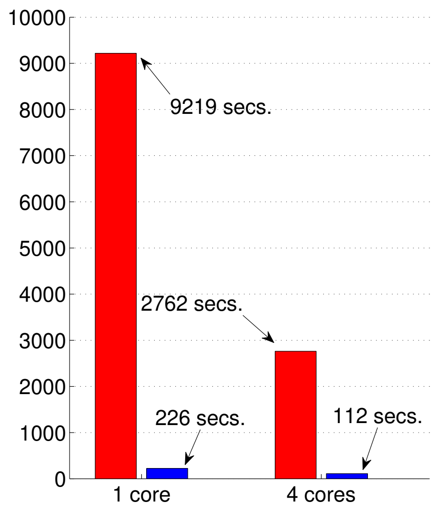
Computing the eigenvalues of a matrix $A \in \mathbb{R}^{n\times n}$ is at the very heart of numerical linear algebra, with applications coming from a broad range of science and engineering. With the increased complexity of mathematical models and availability of HPC systems, there is a growing demand to solve large-scale eigenvalue problems.

Iterative eigensolvers, such as Krylov subspace or Jacobi-Davidson methods [8], have been developed with the aim of addressing such large-scale problems. However, in some applications it might be difficult or even impossible to make use of iterative methods that presume the availability of direct LU factorizations or good preconditioners of $A - \sigma I$ for several different

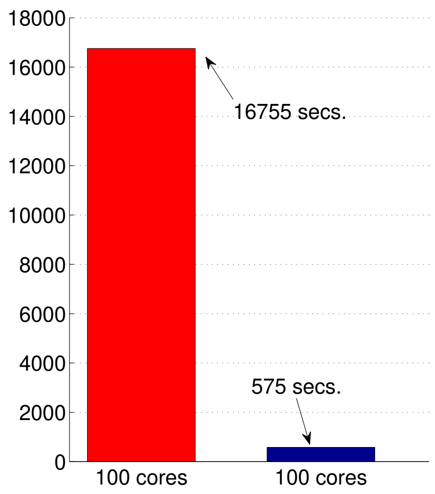
[1]Department of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden. {granat,bokg}@cs.umu.se

[2]Seminar for Applied Mathematics, ETH Zurich, Switzerland. kressner@math.ethz.ch

*Technical Report UMINF-09.06, Department of Computing Science, Umeå University, Sweden, and Research Report 2009-15, Seminar for applied mathematics (SAM), ETH Zurich, Switzerland. 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* under grant VR 70625701 and by the *Swedish Foundation for Strategic Research* under grant A3 02:128.



(a)



(b)

Figure 1: Performance of existing ScaLAPACK implementation `PDLAHQR` (red) vs. newly proposed implementation `PDHSEQR` (blue) on Intel Xeon quadcore nodes when computing the Schur form of a dense random matrix reduced to Hessenberg form. (a) Execution times for a 4000×4000 matrix using 1 or all 4 cores of a single node. (b) Execution times for a $16\,000 \times 16\,000$ matrix using all 100 cores of 25 nodes.

shifts σ. To use iterative methods effectively, one typically requires some knowledge on the locations of eigenvalues of interest. Moreover, there is always the (slight) danger that an eigenvalue is missed, which may have perilous consequences in, e.g., a stability analysis. Finally, by their nature, standard iterative methods are ineffective in situations where a large number of eigenvalues and eigenvectors needs to be computed, as in linear-quadratic optimal control [48] or density functional theory [50]. In contrast, eigensolvers based on similarity transformations, such as the QR algorithm, require no additional knowledge about the problem at hand and, since all eigenvalues are computed anyway, there is no danger to miss an eigenvalue. We conclude that an urgent need for highly performant parallel variants of direct eigensolvers can be expected to persist also in the future.

Often motivated by applications in computational chemistry, particular attention has been paid to parallel direct eigensolvers for *symmetric* matrices, see [4, 13, 62] for some recent work. Less emphasis has been put on the more general dense nonsymmetric case, which is the focus of this paper. The current state-of-the-art parallel implementation of the QR algorithm for solving nonsymmetric eigenvalue problems is the ScaLAPACK [56] routine `PDLAHQR` [36]. In this paper, we propose a significantly improved parallel variant of the QR algorithm. Figure 1 provides a representative sample of the speedups that can be expected when using our new implementation `PDHSEQR`. We refer to Section 4 for more details on the experimental setup.

1.1 Review of earlier work

Parallelizing the nonsymmetric QR algorithm is a non-trivial task. Many of the early attempts achieved neither an even workload nor sufficient scalability, see, e.g., [15, 25, 27, 28, 57].

A major obstacle was the sequential nature of the once-popular double implicit shift QR algorithm, which was considered for parallelization at that time.

More recent attempts to solve the scalability problems were presented in [35, 55, 58, 59, 60, 63], especially when focus turned to small-bulge multishift variants with (Cartesian) two-dimensional (2D) block cyclic data layout [36]. However, as will be discussed below, a remaining problem so far has been a seemingly non-tractable trade-off problem between local node speed and global scalability. We refer to the introduction of [36] for a more detailed history of the parallel QR algorithm until the end of the last century.

State-of-the-art serial implementation

Compared to Francis' original description [26, 45], two major ingredients contribute to the high efficiency of the current LAPACK [3] implementation of the QR algorithm [20].

1. Instead of only a single bulge, containing two (or slightly more) shifts [6, 65], a *chain of several tightly coupled bulges*, each containing two shifts, is chased in the course of one *multishift QR iteration*. Independently proposed in [17, 46], this idea allows to perform most of the computational work in terms of matrix-matrix multiplications and to benefit from highly efficient level 3 BLAS [24, 40]. It is worth noting that the experiments in [17] demonstrate quite remarkable parallel speedup on an SMP system (Origin2000), by simply linking the serial implementation of the QR algorithm with multithreaded BLAS.

2. Introduced by Braman, Byers, and Mathias [18], *aggressive early deflation (AED)* allows to detect converged eigenvalues much earlier than conventional deflation strategies, such as the classical subdiagonal criterion. In effect, the entire QR algorithm requires significantly less iterations, and henceforth less operations, until completion. A theoretical analysis of AED can be found in [44]. Braman [16] has investigated how this deflation strategy could be extended to force deflations in the middle of the matrix, possibly leading to a divide-and-conquer QR algorithm.

State-of-the-art parallel implementation

`PDLAHQR`, the current parallel multishift QR algorithm implemented in ScaLAPACK, was developed by Henry, Watkins, and Dongarra [36]. It is – to the best of our knowledge – the only publicly available parallel implementation of the nonsymmetric QR algorithm. The main idea of this algorithm is to chase a *chain of loosely coupled bulges*, each containing two shifts, during a QR iteration. Good scaling properties are achieved by pipelining the 3×3 Householder reflectors, generated in the course of the QR iteration, throughout the processor mesh for updates. Unfortunately, because of the small size of the Householder reflectors, the innermost computational kernel `DLAREF` operates with level 1 speed [36], which causes the uniprocessor speed to be far below practical peak performance.

1.2 Motivation and organization of this work

Despite its good scalability, the current ScaLAPACK implementation of the QR algorithm often represents a severe time-consuming bottleneck in applications that involve the parallel computation of the Schur decomposition of a matrix. We have observed this phenomenon in our work on parallel Schur-based solvers for linear and quadratic matrix equations [32, 31, 34]. So far, the lack of a modern and highly efficient parallel QR algorithm has rendered these solvers slow and less competitive in comparison with fully iterative methods, such as the sign-function iteration [19, 54] for solving matrix equations from control-related applications in parallel [9, 10, 11]. One motivation of this paper is to use our new parallel variant of the nonsymmetric QR algorithm to increase the attractiveness of Schur-based methods from the perspective of an improved *Total-Time-to-Delivery* (TTD).

The rest of this paper is organized as follows. In Section 2, we present our strategy for parallelizing the multishift QR algorithm, essentially a careful combination of ideas from [33] and the state-of-the-art LAPACK/ScaLAPACK implementations. Section 3 provides an overview of the new routine `PDHSEQR` and illustrates selected implementation details. In Section 4, we present a broad range of experimental results confirming the superior performance of `PDHSEQR` in comparison with the existing ScaLAPACK implementation. Some conclusions and an outline of future work can be found in Section 5. Finally, Appendix A contains supplementary experimental data for various application oriented examples.

2 Algorithms

Given a matrix $A \in \mathbb{R}^{n\times n}$, the goal of the QR algorithm is to compute a *Schur decomposition* [29]

$$Z^T AZ = T, \tag{1}$$

where $Z \in \mathbb{R}^{n\times n}$ is orthogonal and $T \in \mathbb{R}^{n\times n}$ is quasi-upper triangular with diagonal blocks of size 1×1 and 2×2 corresponding to real eigenvalues and complex conjugate pairs of eigenvalues, respectively. This is the standard approach to solving non-symmetric eigenvalue problems, that is, computing eigenvalues and invariant subspaces (or eigenvectors) of a general dense matrix A. Note that the matrix T is called a *real Schur form* of A and its diagonal blocks (eigenvalues) can occur in any order along the diagonal.

Any modern implementation of the QR algorithm starts with a decomposition

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

where H is in upper *Hessenberg form* [29] and Q is orthogonal. Efficient parallel algorithms for this Hessenberg reduction, which can be attained within a finite number of orthogonal transformations, are described in [12, 21] and implemented in the ScaLAPACK routine `PDGEHRD`. Note that the term QR algorithm often refers only to the second iterative part, after (2) has been computed. We will follow this convention throughout the rest of this paper.

Optionally, *balancing* [29] can be used before applying any orthogonal transformation to A, with the aim of (i) reducing the norm of A and (ii) isolating eigenvalues that can be deflated without performing any floating point operations. We have implemented balancing in `PDGEBAL`, a straightforward ScaLAPACK implementation of the corresponding LAPACK routine `DGEBAL`. In many software packages, balancing is by default turned on. See, however, recent examples by Watkins [66], for which it is advisable to turn part (i) of balancing off.

To produce an invariant subspace corresponding to a specified set of eigenvalues, the decomposition (1) needs to be post-processed by *reordering the eigenvalues* of T [7], for which a blocked parallel algorithm is described in [33]. Note that eigenvalue reordering is also an important part of AED.

In the following, we describe our approach to parallelizing the QR algorithm, which relies on experiences from our work on a parallel eigenvalue reordering algorithm [33] mentioned above. In this context, the key to high node speed *and* good scalability is the concept of a parallel *multi-window* approach, combined with delaying and accumulating orthogonal transformations [17, 23, 43, 47]. In what follows, we assume that the reader is somewhat familiar with the basics of the implicit shifted QR algorithm, see, e.g., [42, 67, 68] for an introduction.

2.1 Data partitioning and process organization

We make use of the following well-known ScaLAPACK [14] conventions of a distributed memory environment:

- The $p = P_r P_c$ parallel processes are organized into a $P_r \times P_c$ rectangular mesh labeled from $(0, 0)$ to $(P_r - 1, P_c - 1)$ according to their specific position indices in the mesh.
- All $n \times n$ matrices are distributed over the mesh using a 2-dimensional (2D) block cyclic mapping with block size n_b in the row and column dimensions.

Locally, each process in the mesh may also utilize multithreading, see Section 3.2. This can be seen as adding another level of explicit parallelization by organizing the processes into a three-dimensional $P_r \times P_c \times P_{\mathrm{t}}$ mesh, where the third dimension denotes the number of threads per parallel ScaLAPACK process.

2.2 Parallel bulge chasing

Consider a Hessenberg matrix H and two shifts σ_1, σ_2, such that either $\sigma_1, \sigma_2 \in \mathbb{R}$ or $\overline{\sigma_1} = \sigma_2$. Then the implicit double shift QR algorithm proceeds by computing the first column of the shift polynomial:

$$v = (H - \sigma_1 I)(H - \sigma_2 I)e_1 = \begin{bmatrix} \mathsf{X} \\ \mathsf{X} \\ \mathsf{X} \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Here, and in the following, X denotes arbitrary, typically nonzero, entries. By an orthogonal transformation Q_0, typically a 3×3 Householder reflection, the second and third entries of v are mapped to zero. Applying the corresponding similarity transformation to H results in the nonzero pattern

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

where $\widehat{\mathsf{X}}$ denotes elements that have been updated during the transformation. Note that the Hessenberg structure of H is disturbed by the so called *bulge* in $H(2:4, 1:3)$. In a very specific sense [64], the bulge encodes the information contained in the shifts σ_1, σ_2. By an appropriate 3×3 Householder reflection, the entries $H(3,1)$ and $H(4,1)$ can be eliminated:

$$H \leftarrow Q_1^T H Q_1 = \begin{bmatrix} \times & \hat{\times} & \hat{\times} & \hat{\times} & \times & \times & \times & \cdots \\ \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ \hat{0} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ \hat{0} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ 0 & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \times & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & \times & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & 0 & \times & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix},$$

which *chases* the bulge one step down along the subdiagonal. The bulge can be chased further down by repeating this process in an analogous manner:

$$H \leftarrow Q_2^T H Q_2 = \begin{bmatrix} \times & \times & \hat{\times} & \hat{\times} & \hat{\times} & \times & \times & \cdots \\ \times & \times & \hat{\times} & \hat{\times} & \hat{\times} & \times & \times & \cdots \\ 0 & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ 0 & \hat{0} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \hat{\times} & \hat{\times} & \cdots \\ 0 & \hat{0} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \hat{\times} & \hat{\times} & \cdots \\ 0 & 0 & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \times & \times & \cdots \\ 0 & 0 & 0 & 0 & 0 & \times & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}, \quad H \leftarrow Q_3^T H Q_3 = \begin{bmatrix} \times & \times & \times & \hat{\times} & \hat{\times} & \hat{\times} & \times & \cdots \\ \times & \times & \times & \hat{\times} & \hat{\times} & \hat{\times} & \times & \cdots \\ 0 & \times & \times & \hat{\times} & \hat{\times} & \hat{\times} & \times & \cdots \\ 0 & 0 & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ 0 & 0 & \hat{0} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \hat{\times} & \cdots \\ 0 & 0 & \hat{0} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \hat{\times} & \cdots \\ 0 & 0 & 0 & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}. \qquad (3)$$

Early implementations of the QR algorithm continue this process until the bulge vanishes at the bottom right corner, completing the QR iteration. The key to more efficient implementations is to note that another bulge, belonging to a possibly different set of shifts σ_1, σ_2, can be introduced right away without disturbing the first bulge:

$$H \leftarrow Q_4^T H Q_4 = \begin{bmatrix} \hat{\times} & \hat{\times} & \hat{\times} & \times & \times & \times & \times & \cdots \\ {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & {\color{blue}\hat{\times}} & \hat{\times} & \hat{\times} & \hat{\times} & \hat{\times} & \cdots \\ 0 & 0 & 0 & {\color{blue}\times} & {\color{blue}\times} & {\color{blue}\times} & \times & \cdots \\ 0 & 0 & 0 & {\color{blue}\times} & {\color{blue}\times} & {\color{blue}\times} & \times & \cdots \\ 0 & 0 & 0 & {\color{blue}\times} & {\color{blue}\times} & {\color{blue}\times} & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$$

This creates a chain of 2 *tightly coupled* bulges. Chasing an entire chain of bulges instead of a single bulge offers more possibilities for higher node performance and, as demonstrated below, more concurrency/parallelism in the computations.

In ScaLAPACK's `PDLAHQR`, a chain of *loosely* coupled bulges is used, see Figure 2. The bulges are placed at least n_b steps apart, such that each bulge resides on a different diagonal block in the block cyclic distribution of the matrix H. Such an approach achieves good scalability by chasing the bulges in parallel and pipelining the 3×3 Householder reflections, generated during the bulge chasing process, before updating off-diagonal blocks. However, since the updates are performed by calls to `DLAREF`, which has data reuse similar to level 1 BLAS operations [36, Pg. 285], the performance attained on an individual node is typically far below its practical peak performance. To avoid this effect, we adapt ideas from [17, 46] that allowed for level 3 BLAS in serial implementations of the QR algorithm.

Our new implementation `PDHSEQR` uses several chains of tightly coupled bulges. Each of the chains is placed on a different diagonal block, see Figure 2. The number of such chains is determined by the number of available shifts (see Section 2.3), the wanted number of shifts

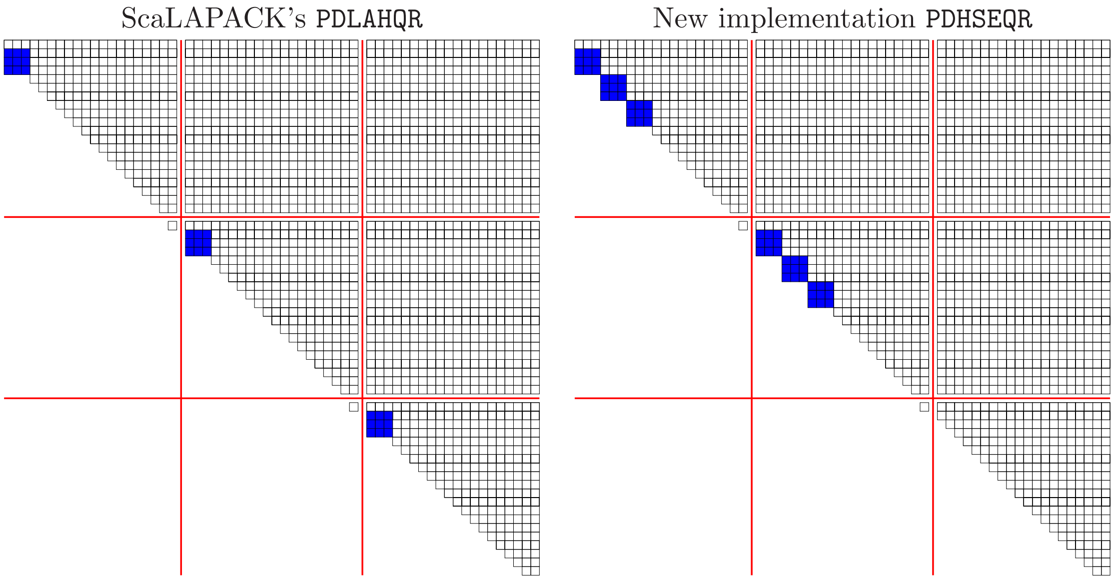


Figure 2: Typical location of bulges in `PDLAHQR` and `PDHSEQR`. Only parts of the matrix are displayed. The solid red lines represent block/process borders.

per chain, and the number of processes utilized. Typically, we choose the number of shifts such that each chain covers at most half of the data layout block. Each chain resides in a *computational window*, within which its bulges are chased.

Assuming a situation as in the right part of Figure 2, the *intra-block* chase of bulge chains proceeds as follows.

- The computational windows are chosen as the diagonal blocks in which chains reside, see the yellow regions in Figure 3.

- Within each computational window, the chain is chased from the top left corner to the bottom right corner. This is performed in parallel and independently. During the chase we perform only *local updates*, that is, only these parts of the matrix which belong to a computational window are updated by the transformations generated during the chase. Depending on the block size, a well-known delay-and-accumulate technique [17, 23, 33, 47] can be used to ensure that an overwhelming majority of the computational work is performed by calls to level 3 BLAS *during* the local chase.

- For each window, the corresponding orthogonal transformations are accumulated into an orthogonal factor of size at most $(n_b - 2) \times (n_b - 2)$. Each orthogonal factor takes the form

$$U = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix} = \qquad ; \tag{4}$$

  that is, U_{21} is upper triangular and U_{12} is lower triangular. These orthogonal factors are broadcasted to the processes holding off-diagonal parts that need to be updated. To avoid conflicts in intersecting scopes (see the (1,2) block in Figure 3), broadcasts are performed in parallel first in the row direction and only afterward in the column

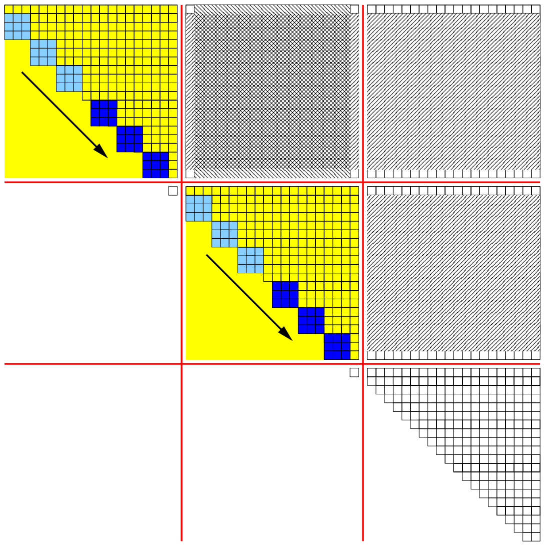


Figure 3: An intra-block chase of the bulge chains.

direction. The updates of the off-diagonal blocks are performed by calls to the level 3 BLAS routine DGEMM (GEneral Matrix Multiply and add operation). Optionally, the structure (4) can be utilized by calls to DGEMM and DTRMM (TRiangular Matrix Multiply operation). Note that these off-diagonal updates are strictly local and require no additional communication.

After the described intra-block chase is completed, all chains reside on the bottom right corners of the diagonal blocks. To move these chains across the process border, the following *inter-block (cross border) chase* is performed.

- Each computational window is chosen to accommodate the initial and target locations of the bulge chain, see Figure 4. In the first round, we only select odd-numbered windows (counted from the bottom). Only after this round is completed, we select even-numbered windows. This odd-even cross-border approach is also used in [33]; it increases concurrency while limiting the amount of extra storage for the overlapping off-diagonal update regions.

- For each selected window in the current round, we create a copy of the window on each side of the border. Then we chase the chain to the bottom of the window, just as in the intra-block chase, and broadcast the corresponding orthogonal factors to the blocks on both sides of the cross border. Note that there are no intersecting scopes. For updating parts outside the windows, neighbor processes holding cross-border regions exchange their data in parallel, and the updates are computed in parallel. Note that the structure (4) of the orthogonal factors aligns with the cross border, so that again a combination of DGEMM and DTRMM can optionally be used.

The intra- and inter-block chases of bulge chains illustrated above describe the generic situation in the middle of a QR iteration. In the beginning and in the end of a QR iteration, the chains are introduced in the top left corner and chased off the bottom right corner, respectively. For these parts, the ideas from the intra- and inter-block chases can be extended

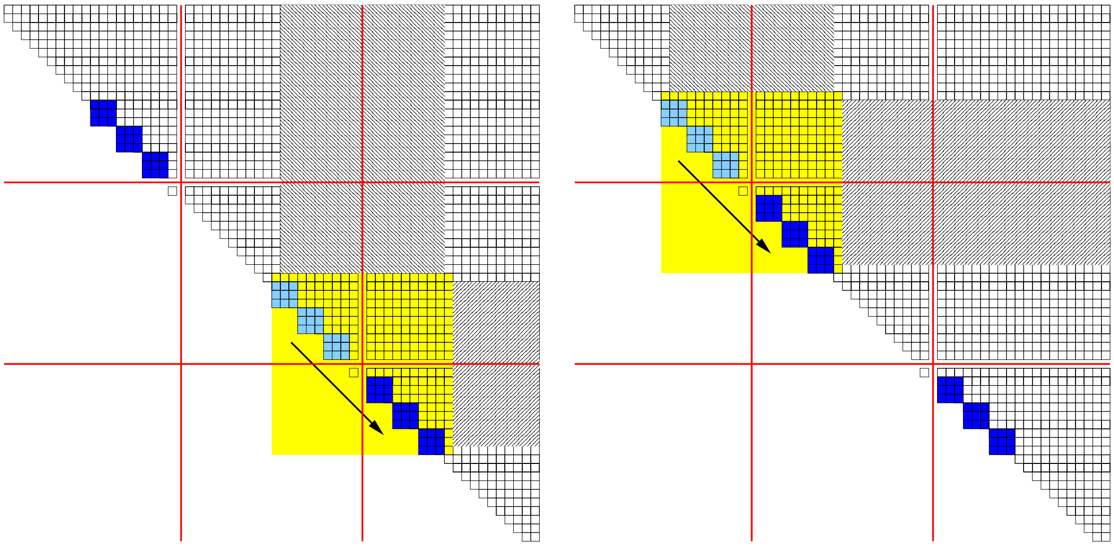


Figure 4: An inter-block (cross process border) chase of the bulge chains.

in an obvious fashion. However, a careful implementation is needed to handle these parts properly.

In exceptional cases, when there is a lack of space on the target side of the cross border for an incoming bulge chain, this chain is delayed and chased across the border as soon as there is sufficient space. Sometimes it is also necessary to handle windows in chunks of size $\min\{P_r, P_c\} - 1$ to avoid conflicts between computational windows with intersecting process scopes.

In this paper, we follow the typical ScaLAPACK approach of scheduling communication and computation statically. For preliminary work on the use of dynamic scheduling, which might be more suitable for distributed memory architectures with multi-core nodes, we refer to [41, 49].

2.3 Parallel aggressive early deflation

In the classical QR algorithm, convergence is detected by checking the subdiagonal entries of the Hessenberg matrix H after each iteration. If the $(i+1, i)$ subdiagonal entry satisfies

$$|h_{i+1,i}| \leq \mathbf{u} \max\{|h_{i,i}|, |h_{i+1,i+1}|\}, \tag{5}$$

where $\mathbf{u}$ denotes the unit roundoff (double precision $\mathbf{u} \approx 1.1 \times 10^{-16}$), then $h_{i+1,i}$ is set to zero and H becomes block upper triangular. The eigenvalue problem *deflates* into two smaller problems associated with the two diagonal blocks of H, which can be treated separately by the QR algorithm. Typically, convergence takes place at the bottom right corner and the size of the lower diagonal block is roughly the number of shifts used in the iteration.

In modern variants of the QR algorithm, the award-winning aggressive early deflation (AED) strategy [18] is used in addition to (5). It often detects convergence much earlier than (5) and significantly reduces the average number of shifts needed to deflate one eigenvalue, see also Table 6 in Section 4.5. In the following, we illustrate the basic principle of AED but refer to [18] for algorithmic details. After having performed a QR iteration, a deflation window size n_{win} is chosen and the $n \times n$ matrix H is partitioned as follows:

$$
H = \begin{array}{c} \\ n-n_{\text{win}}-1 \\ 1 \\ n_{\text{win}} \end{array}
\begin{array}{c} \begin{array}{ccc} n-n_{\text{win}}-1 & 1 & n_{\text{win}} \end{array} \\
\left[\begin{array}{ccc} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ 0 & H_{32} & H_{33} \end{array}\right] \end{array}. \tag{6}
$$

Then a Schur decomposition of H_{33} is computed and H is updated by the corresponding orthogonal similarity transformation. The following diagram illustrates the shape of the updated matrix H for $n_{\text{win}} = 5$:

$$
H \leftarrow \left[\begin{array}{ccc|ccccc}
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & x & x & \hat{x} & \hat{x} & \hat{x} & \hat{x} & \hat{x} \\
\cdots & x & x & \hat{x} & \hat{x} & \hat{x} & \hat{x} & \hat{x} \\
\hline
\cdots & 0 & {\color{red}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} \\
\cdots & 0 & {\color{red}\hat{x}} & 0 & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} \\
\cdots & 0 & {\color{red}\hat{x}} & 0 & 0 & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} \\
\cdots & 0 & {\color{red}\hat{x}} & 0 & 0 & 0 & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} \\
\cdots & 0 & {\color{red}\hat{x}} & 0 & 0 & 0 & 0 & {\color{blue}\hat{x}}
\end{array}\right]
$$

To keep the description simple, we have assumed that all eigenvalues of H_{33} are real. The vector of newly introduced red entries is called *spike*. If the trailing entry of the spike is sufficiently small, say not larger than $\mathbf{u}$ times the Frobenius norm of H_{33}, it can be safely set to zero. As a consequence, H becomes block upper triangular with a deflated eigenvalue at the bottom right corner. Subsequently, the procedure is repeated for the remaining $(n-1)\times(n-1)$ diagonal block. If, however, the trailing spike entry is not sufficiently small then eigenvalue reordering [29] is used to move the undeflatable eigenvalue to the top left corner of H_{33}. This brings a different eigenvalue of H_{33} into the bottom right position. Again, the (updated) trailing entry of the spike is checked for convergence. The entire procedure is repeated until all eigenvalues have been checked for deflation. At the end, hopefully $k \geq 1$ eigenvalues could be deflated and H takes the following form (in this example, $k = 2$ eigenvalues could be deflated):

$$
H \leftarrow \left[\begin{array}{ccc|ccccc}
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & x & x & \hat{x} & \hat{x} & \hat{x} & \hat{x} & \hat{x} \\
\cdots & x & x & \hat{x} & \hat{x} & \hat{x} & \hat{x} & \hat{x} \\
\hline
\cdots & 0 & {\color{red}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} \\
\cdots & 0 & {\color{red}\hat{x}} & 0 & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} \\
\cdots & 0 & {\color{red}\hat{x}} & 0 & 0 & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} \\
\cdots & 0 & 0 & 0 & 0 & 0 & {\color{blue}\hat{x}} & {\color{blue}\hat{x}} \\
\cdots & 0 & 0 & 0 & 0 & 0 & 0 & {\color{blue}\hat{x}}
\end{array}\right]
$$

A Householder reflection of the spike combined with Hessenberg reduction of the top left part of H_{33} turn H back to Hessenberg form:

$$
H \leftarrow \left[\begin{array}{ccc|ccc|cc}
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & x & x & \hat{x} & \hat{x} & \hat{x} & x & x \\
\cdots & x & x & \hat{x} & \hat{x} & \hat{x} & x & x \\
\hline
\cdots & 0 & \hat{x} & \hat{x} & \hat{x} & \hat{x} & \hat{x} & \hat{x} \\
\cdots & 0 & \hat{0} & \hat{x} & \hat{x} & \hat{x} & \hat{x} & \hat{x} \\
\cdots & 0 & \hat{0} & 0 & \hat{x} & \hat{x} & \hat{x} & \hat{x} \\
\hline
\cdots & 0 & 0 & 0 & 0 & 0 & {\color{blue}x} & {\color{blue}x} \\
\cdots & 0 & 0 & 0 & 0 & 0 & 0 & {\color{blue}x}
\end{array}\right].
$$

Table 1: Suggested size of deflation window.

n	n_s	n_{win}
75–150	10	15
150–590	see [20]	
590–3000	64	96
3000–6000	128	192
6000–12000	256	384
12000–24000	512	768
24000–48000	1024	1536
48000–96000	2048	3072
96000–∞	4096	6144

The QR algorithm is continued on the top left $(n-k) \times (n-k)$ submatrix of H.

The eigenvalues of H_{33} that could not be deflated are used as shifts in the next multishift QR iteration. Following the recommendations in [20] we use $n_s = 2n_{\text{win}}/3$ shifts, see also Table 1. If there are less than n_s undeflatable eigenvalues available, the next multishift QR iteration is skipped and another AED is performed.

To perform AED in parallel, we need to discuss (i) the reduction of H_{33} to Schur form, and (ii) the eigenvalue reordering within the AED window.

(i) Table 1 provides suggestions for choosing n_{win} with respect to the total matrix size n, extrapolated from Byers' suggestions [20] for the serial QR algorithm. With a typical data layout block size between 32 and a few hundreds, the AED window will usually not reside on a single block (or process). For modest window sizes, a viable and remarkably efficient option is to simply gather the entire window on one processor and perform the Schur decomposition serially by calling LAPACK's `DHSEQR`. For larger window sizes, this can be expected to become a bottleneck and a parallel algorithm needs to be used. We have evaluated an extension of the LAPACK approach [20] to use one level of recursion and perform the Schur decomposition for AED with the same parallel QR algorithm described in this paper. (The AED on the lower recursion level is performed serially.) Unfortunately, we have observed rather poor scalability when using such an approach, possibly because of the relatively small window sizes. At the moment, we use a modification of ScaLAPACK's `PDLAHQR` (see Section 3.1) to reduce H_{33} to Schur form. When profiling our parallel QR algorithm, see Appendix A, it becomes obvious that the performance of this part of the algorithm has a substantial impact on the overall execution time. Developing a parallel variant of the QR algorithm that scales and performs well for the size of matrices occurring in AED is expedient for further improvements and subject to future research.

(ii) To attain good node performance in the parallel algorithm for the eigenvalue reordering [33] used in the AED process, it is necessary to reorganize AED such that groups instead of individual eigenvalues are to be reordered. For this purpose, a small computational window at the bottom right corner of the AED window is chosen, marked by the yellow region in Figure 5. The size of this window is chosen to be equal to n_b and could be shared by several processes or completely local. Within this smaller window, the ordinary AED is used to identify deflatable eigenvalues and move undeflatable eigenvalues (marked blue in Figure 5) to the top left corner of the smaller window.

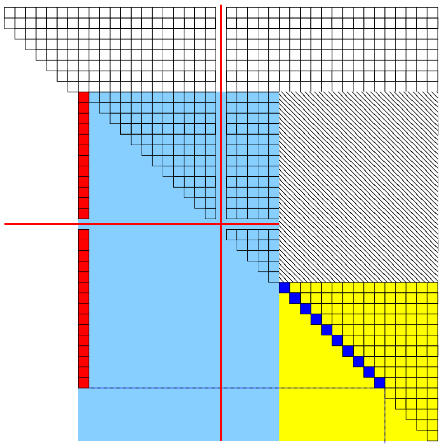


Figure 5: AED within a local computational window of a single process.

The rest of the AED window is updated by the corresponding accumulated orthogonal transformations. Now the entire group of undeflatable eigenvalues can be moved simultaneously by parallel eigenvalue reordering to the top left corner of the AED window. The whole procedure is repeated by placing the next local computational window at the bottom right corner of the remaining AED window.

The orthogonal transformations generated in the course of AED are accumulated into a 2D block-cyclic distributed orthogonal matrix. After AED has been completed, this orthogonal matrix is used to update parts of the Hessenberg matrix outside of the AED window. The update is performed by parallel GEMM using PBLAS [53].

We have performed preliminary experiments with a *partial* AED procedure that still performs a Schur reduction of the AED window but omits the somewhat tricky reordering step. Our observations revealed that such a partial procedure on the one hand deflates a surprisingly substantial part of the eigenvalues corresponding to the bottom-most part of the spike, located below the first undeflatable eigenvalue. On the other hand, it is still less effective than the full AED procedure and leads to an increased number of QR iterations, ultimately increasing the execution time by roughly up to a factor two. Further research into simple heuristic but yet effective variants of AED is needed.

3 Implementation details

3.1 Software hierarchy

Figure 6 provides an overview of the software developed to implement the algorithms described in Section 2. We used a naming convention that is reminiscent of the current LAPACK implementation of the QR algorithm [20]. The entry routine for the new parallel QR algorithm is the ScaLAPACK-like Fortran routine `PDHSEQR`, which branches into `PDLAQR1` for small (sub)matrices or `PDLAQR0` for larger (sub)matrices. We remark that the current ScaLAPACK routine `PDLAHQR` (version 1.8.0) does not properly post-process 2×2 blocks that contain two real eigenvalues. Our modification in `PDLAQR1` partly consists of fixing this bug, which would have severely harmed AED. More importantly, `PDLAQR1` is equipped with a novel multithreaded version of the computational kernel `DLAREF` for applying Householder

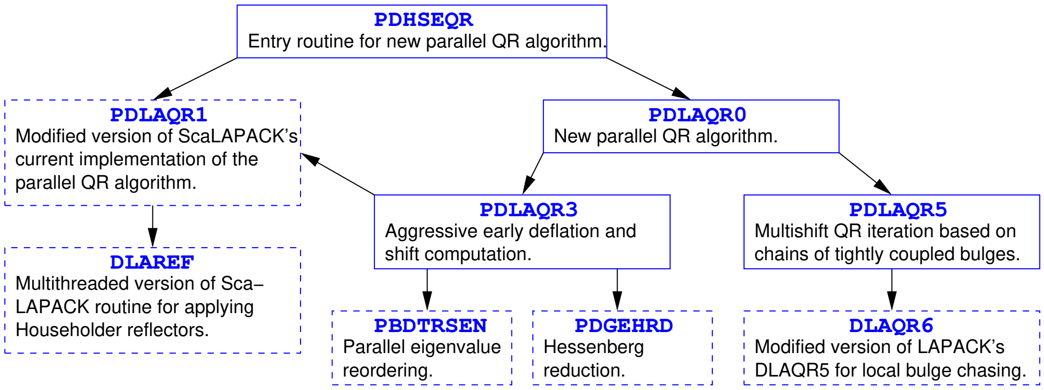


Figure 6: Routines and dependencies for new implementation of the parallel QR algorithm. Dependencies on LAPACK routines, auxiliary ScaLAPACK routines, BLAS, BLACS, and parallel BLAS are not shown.

reflections. The routines `PDLAQR3` and `PDLAQR5` implement the algorithms described in Sections 2.3 and 2.2, respectively. Parallel eigenvalue reordering, which is needed for AED, is implemented in the routine `PBDTRSEN` described in [33].

3.2 Mixing MPI and OpenMP for multithreaded environments

We provide the possibility for local multithreading on each node in the logical process grid by mixing our parallel MPI (BLACS [14]) program with OpenMP directives. These directives are inserted directly in our ScaLAPACK-style code and optionally compiled and executed in a multithreaded environment with SMP-like and/or multicore nodes.

For example, applications of Householder transformations and accumulated orthogonal transformations *within* a computational window are fairly evenly divided inside parallel regions of the code as *independent loop iteration operations* or *disjunct tasks* among the available threads. In the parallel bulge-chase, the level 3 updates of the far-from-diagonal entries in H and the updates of the orthogonal matrix Z are divided between several threads within each node in the logical grid by using a new level of blocking for multithreading.

Alternatively, a highly efficient threaded implementation of the BLAS could be utilized. However, this would not cover all parallelizable operations; moreover, for *small-sized* matrix operations an explicit blocking procedure in combination with a carefully tuned OpenMP parallelization is expected to be more efficient than a threaded BLAS, especially for multicore environments.

4 Experiments

In this section, we present various experiments on two different parallel platforms to confirm the superior performance of our parallel QR algorithm in comparison to the existing ScaLAPACK implementation.

4.1 Hardware and software issues

We utilized the following two computer systems *akka* and *sarek*, both hosted by 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 1.2.6 [52], BLACS 1.1patch3, GOTO BLAS r1.26 [30] LAPACK 3.1.1, ScaLAPACK/PBLAS 1.8.0
sarek	64-bit Opteron Linux Cluster 192 single-core AMD Opteron 2.2GHz dual nodes 64KB L1 cache, 1MB L2 caches, 8GB RAM per node Myrinet-2000 high performance interconnect, 250 MB/sec bandwidth MPICH-GM 1.5.2 [52], BLACS 1.1patch3, GOTO BLAS r0.94 [30] LAPACK 3.1.1, ScaLAPACK/PBLAS 1.7.0

For all our experiments, we used the Fortran 90 compiler `pgf90` version 7.2-4 from the Portland group compiler suite using the flags `-mp -fast -tp k8-64 -fastsse -Mnontemporal -Msmart -Mlarge_arrays -Kieee -Mbounds -Mpreprocess`.

4.2 Performance tuning

The 2D block cyclic distribution described in Section 2.1 depends on a number of parameters, such as the sizes of the individual data layout blocks. Block sizes that are close to optimal for `PDHSEQR` have been determined by extensive tests on a few cores and then used throughout the experiments: for *sarek* we use the block factor $n_b = 160$, and on *akka* we use the block factor $n_b = 50$. This striking difference merely reflects the different characteristics of the architectures. On *akka*, the size of the L1 cache local to each core and the shared L2 cache on each node benefit from operations on relatively small blocks. On *sarek*, the larger blocks represent a close to optimal explicit blocking for the memory hierarchy of each individual processor. Alternatively, auto-tuning may be used to determine good values for n_b and other parameters on a given architecture but this is beyond the scope of this paper, see [69, 70, 71] for possible approaches. We found ScaLAPACK's `PDLAHQR` to be quite insensitive to the block size, presumably due to the fact that its inner kernel `DLAREF` is not affected by this choice. However, a tiny block size, say below 10, causes too much communication and should be avoided. We use the same block size for `PDLAHQR` as for `PDHSEQR`. Since the QR algorithm operates on square matrices, it is natural to choose a square process grid, i.e., $P_r = P_c$.

If AED detects a high fraction of eigenvalues in the deflation window to be converged, it can be beneficial to skip the subsequent QR sweep and perform AED once again on a suitably adjusted deflation window. An environment parameter `NIBBLE` is used to tune this behavior: if the percentage of converged eigenvalues is higher than `NIBBLE` then the subsequent QR sweep is skipped. In the serial LAPACK implementation of the QR algorithm, the default value of `NIBBLE` is 14. For large random matrices, we observed for such a parameter setting an average of 10 AEDs performed after each QR sweep, i.e., 90% of the QR sweeps are skipped. For our parallel algorithm, such a low value of `NIBBLE` severely harms speed and scalability; for all tests in this paper, we have set `NIBBLE` to 50, which turned out to be a well-balanced choice.

4.3 Performance metrics

In the following, we let T_p(PDHSEQR) and T_p(PDLAHQR) denote the measured parallel execution time in seconds when executing PDHSEQR and PDLAHQR on p cores.

The performance metrics for parallel scalability and speedup are defined as follows.

S_p(PDHSEQR) denotes the ratio $T_{p_{\min}}$(PDHSEQR)$/T_p$(PDHSEQR), where $p_{\min}$ is a small fixed number of processors and p varies. Typically, $p_{\min}$ is chosen as the smallest number of cores for which the allocated data structures fit in the aggregate main memory. S_p(PDLAHQR) is analogously defined. The metric S_p corresponds to the classical speedup obtained when increasing the number of cores from $p_{\min}$ to p, while the matrix size n remains the same. Typically, S_p is also used to investigate the scalability of an implemented algorithm when increasing p and keeping the data load per node constant.

S_p(PDLAHQR/PDHSEQR) denotes the ratio T_p(PDLAHQR)$/T_p$(PDHSEQR) and corresponds to the speedup obtained when replacing PDLAHQR by PDHSEQR on p cores.

Both metrics are important to gain insights into the performance of the new parallel QR algorithm. In practice, of course, it is S_p(PDLAHQR/PDHSEQR) what matters. It has been several times pointed out in the literature that the classical speedup definition based on S_p(PDHSEQR) or S_p(PDLAHQR) alone tends to unduly reward parallel algorithms with low node speed. In a general setting, two-sided bulge-chasing transformation algorithms executed in distributed memory environments are expected to have a classical parallel speedup of about $O(\sqrt{p/p_{\min}})$ [22].

4.4 Accuracy and reliability

All experiments are conducted in double precision arithmetic ($\epsilon_{\text{mach}} \approx 2.2 \times 10^{-16}$), and we compute the following accuracy measures after the Schur reduction is completed:

- Relative residual norm: $R_r = \dfrac{\|Z^T A Z - T\|_F}{\|A\|_F}$
- Relative orthogonality check: $R_o = \dfrac{\max(\|Z^T Z - I_n\|_F, \|Z Z^T - I_n\|_F)}{\epsilon_{\text{mach}} n}$

Here, $\|\cdot\|_F$ denotes the Frobenius norm of a matrix and A denotes the original unreduced matrix *before* reduction to Hessenberg form. The orthogonal factor Z contains both the transformations to Hessenberg form and the transformations from Hessenberg to Schur form. Since all variants of the QR algorithm described in this paper are numerically backward stable, we expect $R_r \approx 10^{-16}$ for small matrices, with a modest increase as n increases [37]. In fact, on *akka* both PDLAHQR and PDHSEQR usually return with $R_r \approx 10^{-14}$ also for larger matrices. For the orthogonality check, both routines always return a value of R_o not significantly larger than 1.

To check whether the returned matrix T is indeed in real Schur form we traverse the subdiagonal of T and signal an error if there are two consecutive nonzero subdiagonal elements. For none of the experiments reported in this paper, such an error was signaled.

4.5 Performance for random problems

Two classes of random problems are considered:

`fullrand` A full matrix A with pseudo-random entries from a uniform distribution in $[0, 1]$. The matrix H is obtained by reducing A to Hessenberg form.

`hessrand` A Hessenberg matrix H with pseudo-random nonzero entries from a uniform distribution in $[0, 1]$.

The matrices generated by `fullrand` usually have well-conditioned eigenvalues, yielding a convergence pattern of the QR algorithm that is predictable and somewhat typical for "well-behaved" matrices. In contrast, the eigenvalues of the matrices generated by `hessrand` are notoriously ill-conditioned, see [61] for a theoretical explanation of the closely related phenomenon of ill-conditioned random triangular matrices. In effect, the convergence of the QR algorithm becomes rather poor, see also Table 6; we refer to the discussion in [42] for more details.

Table 2: T_p(PDLAHQR) and T_p(PDHSEQR) on *akka* for matrix class `fullrand`.

	$n =$							
$p = P_r \times P_c \times P_t$	4 000		8 000		16 000		32 000	
$1 \times 1 \times 1$	9332	226	79943	1478				
$2 \times 2 \times 1$	2761	112	20161	640				
$4 \times 4 \times 1$	1174	69	8582	265	67779	1644		
$6 \times 6 \times 1$	697	60	5192	194	32920	1007	∞	6218
$8 \times 8 \times 1$	418	57	3671	152	20856	595	∞	4164
$10 \times 10 \times 1$	368	63	2589	165	16755	516	∞	3046
$1 \times 1 \times 4$	2592	173	18324	913				
$2 \times 2 \times 4$	1617	126	11032	591				
$3 \times 3 \times 4$	834	102	5692	408	38834	2327		
$4 \times 4 \times 4$	612	76	3986	277	25823	1332	∞	9250
$5 \times 5 \times 4$	474	70	2971	203	18934	1061	∞	6568

Table 3: T_p(PDLAHQR) and T_p(PDHSEQR) on *akka* for matrix class `hessrand`.

	$n =$							
$p = P_r \times P_c \times P_t$	4 000		8 000		16 000		32 000	
$1 \times 1 \times 1$	7529	577	61119	5303				
$2 \times 2 \times 1$	2760	271	18844	2281				
$4 \times 4 \times 1$	1789	138	11373	838	77081	2932		
$6 \times 6 \times 1$	1066	116	6677	561	58366	1593	∞	2234
$8 \times 8 \times 1$	755	92	5058	458	37976	996	∞	1714
$10 \times 10 \times 1$	588	106	4172	401	29219	888	∞	1599
$1 \times 1 \times 4$	2136	453	17726	3535				
$2 \times 2 \times 4$	1570	284	9834	1953	70792	4887		
$3 \times 3 \times 4$	932	218	5956	1367	40317	3669		
$4 \times 4 \times 4$	687	165	4499	1040	26573	2094	∞	2284
$5 \times 5 \times 4$	593	123	3319	623	23718	1946	∞	2031

Tables 2–5 provide the execution times of PDLAHQR and PDHSEQR for both random problem classes. Notice that for $P_t > 1$, PDLAHQR is linked with our *threaded* version of DLAREF. To

avoid excessive use of computational resources, we had to set a limit of approximately 1500 cpu-hours for the accumulated time used by all p cores on an individual problem. Runs which were found to exceed this limit are marked with ∞ in the tables. For example, the ∞-value in Table 4 for $n = 24000$ and a $6 \times 6 \times 2$ mesh means that the $6 \times 6 \times 2$ mesh did not solve the problem within the cpu-hours limit, while the $4 \times 4 \times 2$ and $8 \times 8 \times 2$ meshes did solve the same problem within the cpu-hours limit. An empty space means that there was insufficient memory available to allocate among the nodes to hold the data structures of the test program.

For the convenience of the reader, the explicitly computed values of the performance metrics S_p can be found in Section A.1 of the appendix. In all examples, the new `PDHSEQR` is faster than `PDLAHQR`. The speedup S_p(`PDLAHQR/PDHSEQR`) ranges from a few times up to almost fifty times, depending on the problem class, the size of the corresponding Hessenberg matrix and the number of utilized processors. The difference in performance between `fullrand` and `hessrand` is largely due to differences in the total number of applied shifts until convergence. In Table 6 below, we provide this number for a selected set of examples.

Table 4: T_p(`PDLAHQR`) and T_p(`PDHSEQR`) on *sarek* for `fullrand`.

$p = P_r \times P_c \times P_t$	$n =$ 6 000		12 000		24 000		48 000	
$1 \times 1 \times 1$	60600	1201						
$2 \times 2 \times 1$	14691	491	138671	3029				
$4 \times 4 \times 1$	4019	256	33877	1165	338393	6783		
$6 \times 6 \times 1$	2025	203	16345	675	144576	3675	∞	27629
$8 \times 8 \times 1$	1213	184	11097	608	∞	2665	∞	17618
$1 \times 1 \times 2$	27554	824						
$2 \times 2 \times 2$	7537	371	66607	2019				
$4 \times 4 \times 2$	2221	201	21935	819	141716	5020		
$6 \times 6 \times 2$	1119	164	13627	547	∞	2859	∞	16108
$8 \times 8 \times 2$	771	149	5877	490	41530	2059	∞	11598

Table 5: T_p(`PDLAHQR`) and T_p(`PDHSEQR`) on *sarek* for `hessrand`.

$p = P_r \times P_c \times P_t$	$n =$ 6 000		12 000		24 000		48 000	
$1 \times 1 \times 1$	40992	4183						
$2 \times 2 \times 1$	14834	1538	135879	4716				
$4 \times 4 \times 1$	5313	773	40751	1834	∞	2984		
$6 \times 6 \times 1$	3386	551	24866	953	∞	1820	∞	9812
$8 \times 8 \times 1$	2326	466	17513	829	∞	1460	∞	6934
$1 \times 1 \times 2$	19870	2546						
$2 \times 2 \times 2$	7906	1245	65868	3679				
$4 \times 4 \times 2$	2948	616	20917	1349	∞	1876		
$6 \times 6 \times 2$	1913	466	12769	908	∞	1260	∞	5850
$8 \times 8 \times 2$	1365	375	9449	636	∞	981	∞	4412

The speedups S_p(`PDLAHQR`) and S_p(`PDHSEQR`) are typically found in the range of 2–3 when the number of processors is increased from p to $4p$ while the matrix size n remains the same. It is evident that `PDLAHQR` often scales significantly better. We think that this is to a large extent – in two different ways – an effect of the low node speed of `PDLAHQR`. On the one

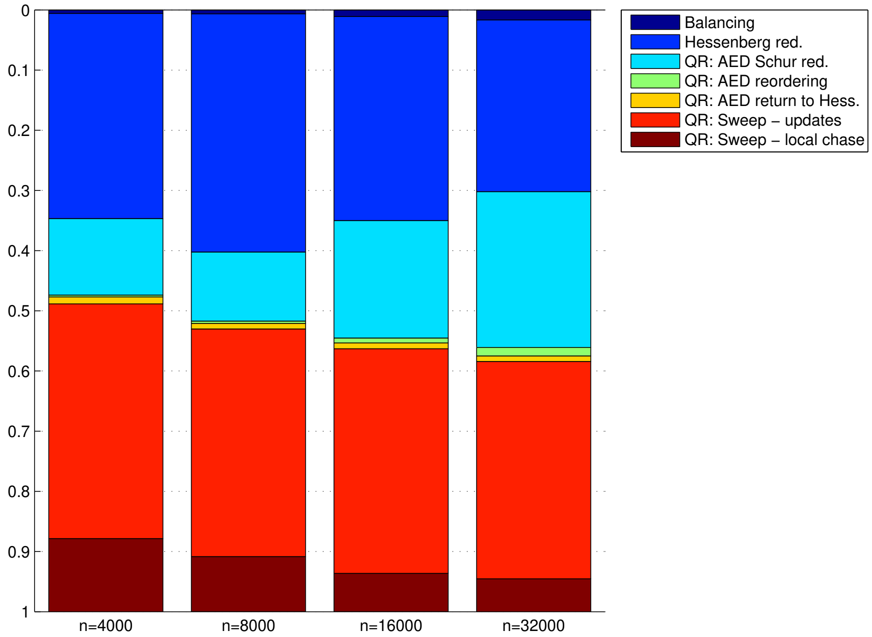


Figure 7: Profiles of T_p for the complete Schur reduction on *akka*; the memory load corresponds to a $4\,000 \times 4\,000$ submatrix per core.

hand, a high execution time on one core makes it disproportionately easy for `PDLAHQR` to scale well. On the other hand, as explained in Section 2.3, the AED in `PDHSEQR` is based on `PDLAHQR`. Since the AED window is usually rather small, `PDHSEQR` does not benefit from the good scalability of `PDLAHQR` as much as it suffers from its low node speed.

To gain more insight into the amount of time spent for AED, Figure 7 provides sample profiles of the execution time for reducing $4\,000 \times 4\,000$ to $32\,000 \times 32\,000$ dense random matrices (`fullrand`), including balancing and Hessenberg reduction, having a fixed memory load on each core corresponding to a $4\,000 \times 4\,000$ matrix. The QR algorithm occupies about 60–70% of the time. As much as 13–28% is spent for AED, mostly for reducing the AED window to Schur form. Note that the portion of time spent on AED increases as n increases; this is due to the increased effectiveness of AED at deflating eigenvalues as the size of the deflation window grows along with n. [1] This confirms the findings of Section 2.3 that significant improvements of our new algorithm for massively parallel computations can only be made if the call to ScaLAPACK's `PDLAHQR` within AED is addressed.

Table 6: Average number of applied shifts to let one eigenvalue converge for `PDLAHQR` and `PDHSEQR`. The figures are identical for *akka* and *sarek*.

$n =$	4 000		6 000		8 000		12 000		16 000		24 000	
						`fullrand`						
	3.93	0.75	3.82	0.48	4.23	0.55	5.32	0.50	4.46	0.46	8.16	0.46
						`hessrand`						
	4.74	1.87	4.49	3.92	4.74	2.21	5.92	4.27	7.77	1.14	N/A	0.00

[1] A dynamic value of `NIBBLE` could possibly be used to keep the balance between the number of performed AEDs and QR sweeps constant.

We note that our threaded version of `DLAREF` already gives a fine boost to the node performance of `PDLAHQR`, which is mainly due to loop parallelization on the nodes. For example, by adding an extra thread on the dual nodes of the Opteron-based platform, we sometimes experience a nearly linear speedup for `PDLAHQR` (e.g, going from $2 \times 2 \times 1$ to $2 \times 2 \times 2$ processors for $n = 12000$ in Table 5). By adding three extra threads on each allocated processor of the quadcore Xeon-based platform, we obtain a speedup of up to about 3 (e.g., going from $4 \times 4 \times 1$ to $4 \times 4 \times 4$ cores for $n = 16000$ in Table 2).

The figures clearly reveal that on *akka* it is generally more beneficial for `PDHSEQR` to allocate one MPI process per core, compared to allocating one MPI process on each node and utilizing multithreading. On the Opteron (NUMA) platform *sarek*, there are more benefits from multithreading but the best choice is still to use only one MPI process per processor.

4.6 Performance without aggressive early deflation

`PDHSEQR` benefits from two separate ingredients, several tightly coupled bulge chains and AED. Of course, both are interwoven. The bulge chains facilitate the undeflatable eigenvalues from AED and, on the other hand, AED works best if a large number of shifts are chased simultaneously. It is still of interest to measure the effect when AED is turned off. For this purpose, we have run experiments on *akka* with 4000×4000 matrices of the class `fullrand`. For example, when using $2 \times 2 \times 1$ cores, `PDLAHQR` consumes 2 761 seconds, `PDHSEQR` consumes 112 seconds, and `PDHSEQR` *without* AED consumes 520 seconds. This demonstrates that both improvements, bulge chains and AED, have a significant impact.

4.7 Performance for benchmark examples

We have performed experiments on a number of benchmark examples from [18, 51] and the NEP collection [5]. The findings are similar as for random matrices: `PDHSEQR` outperforms `PDLAHQR` significantly. More details and can be found in Section A.2. of the appendix.

4.8 A 100.000×100.000 dense eigenvalue problem

To obtain an impression how our parallel QR algorithm as implemented in `PDHSEQR` performs on very large dense eigenvalue problems, we have computed the Schur form of a 100.000×100.000 random matrix (`fullrand`).

The experiment was conducted using 1024 cores of *akka* organized as a 32×32 logical process grid. The whole process took slightly less than 9 hours, where balancing was performed in about 20 minutes, the Hessenberg form was computed in 1 hour and 7 minutes, while the QR algorithm took 7 hours, 3 minutes and 31 seconds. To validate the output we checked the residual, which took about another 23 minutes. Within the QR algorithm, 34 QR iterations were performed but in 22 of these iterations the QR sweep was skipped because AED discovered that more than half of the eigenvalues within the deflation window had converged. In consequence, 80% of the total execution time of the QR algorithm was spent in the parallel AED procedure. On average 0.44 shifts per deflated eigenvalue were needed and the size of the deflation window was 6145×6145 for most iterations.

In the following, we argue that our parallel algorithm is still scalable for $n = 100.000$. Since the QR algorithm in general is an $O(n^3)$ operation (except for situations where no sweeps are performed and the complexity approaches $O(n^2)$, see also Section A.2) and by neglecting the complexity constant we estimate the time for one flop from the 4000×4000 uni-core data in

Table 2, where the memory load is about the same as for the 100.000×100.000 problem, as $t_a = 226/4000^3 \approx 3.53 \times 10^{-9}$. This gives an estimate for $T_p,\ p = 1$ as approximately 981 hours for a 100.000×100.000 `fullrand` problem. Assuming S_p(PDHSEQR) is in line with the assumptions in Section 4.3, i.e., $\sqrt{1024} = 32$, gives a value of $T_p,\ p = 1024$ as roughly 30 hours and 40 minutes. The remaining factor can be explained by that the complexity constant for $n = 100,000$ is probably close to one half of that for $n = 4,000$ (see the converging values of the number of applied shifts for `fullrand` in Table 6) and the real value of S_p(PDHSEQR) is likely higher than 32. Also, in the light of the time ratio of the Hessenberg reduction and the QR algorithm, this 100.000×100.000 problem is consistent with the tendency of an n-driven increased ratio of the QR algorithm as displayed in Figure 7. To conclude, under the given assumptions our novel parallel QR algorithms is indeed scalable for such large-scale problems as 100.000×100.000 matrices.

5 Conclusions and future work

A significantly improved parallel QR algorithm has been presented, incorporating modern techniques such as multiple bulge chain chasing and aggressive early deflation (AED). The resulting implementation outperforms the current ScaLAPACK implementation PDLAHQR significantly and uniformly for all problems under consideration. Still, there is room for further improvement. At the moment, the Schur reduction *within* AED is based on a multithreaded PDLAHQR and represents a bottleneck already for a modest number of processes. Designing a tailored version of PDLAHQR for AED can be expected to diminish the impact of this bottleneck to a certain extent. However, the relatively small size of the AED window will always affect scalability. Simply increasing the size of this window would yield better scalability but also result in more computational work. A more fundamental algorithmic idea might be needed to completely remove this bottleneck.

In the work on this paper, we have benefited from ongoing work on parallelizing the QZ algorithm [1, 2, 39] for generalized eigenvalue problems. Note that, in contrast to the QR algorithm, there is *no* parallel implementation of the QZ algorithm publicly available. We expect that the insights from this paper will cross-fertilize this ongoing work.

We were surprised by the large performance impact of using multithreaded computational kernels in the ScaLAPACK routine PDLAHQR. This encourages further investigation of eigenvalue solvers on multicore processors.

The software developed in this paper is available on request from the authors. We welcome comments and suggestions from users.

Acknowledgments

The authors are grateful to Björn Adlerborn and Lars Karlsson for helpful discussions on the subject and for constructive comments on earlier versions of this paper, and to Åke Sandgren for valuable support in cleaning up and porting the Fortran codes to the various platforms and compilers at HPC2N [38].

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, *Applied Parallel Computing PARA 2002*, volume 2367 of *Lecture Notes in Computer Science*, 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. In B. Kågström et al., editor, *Applied Parallel Computing - State of the Art in Scientific Computing (PARA'06)*, volume 4699, pages 117–126. Lecture Notes in Computer Science, Springer, 2007.

[3] E. Anderson, Z. Bai, C. 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] D. Antonelli and C. Voemel. LAPACK working note 168: PDSYEVR. ScaLAPACK's parallel MRRR algorithm for the symmetric eigenvalue problem. Technical Report UCB/CSD-05-1399, EECS Department, University of California, Berkeley, 2005.

[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] Z. Bai and J. W. Demmel. On a block implementation of the Hessenberg multishift QR iterations. *Internat. J. High Speed Comput.*, 1:97–112, 1989.

[7] Z. Bai and J. W. Demmel. On swapping diagonal blocks in real Schur form. *Linear Algebra Appl.*, 186:73–95, 1993.

[8] Z. Bai, J. W. Demmel, J. J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems*. Software, Environments, and Tools. SIAM, Philadelphia, PA, 2000.

[9] P. Benner, R. Byers, E. S. Quintana-Ortí, and G. Quintana-Ortí. Solving algebraic Riccati equations on parallel computers using Newton's method with exact line search. *Parallel Comput.*, 26(10):1345–1368, 2000.

[10] P. Benner and E. S. Quintana-Ortí. Solving stable generalized Lyapunov equations with the matrix sign function. *Numer. Algorithms*, 20(1):75–100, 1999.

[11] P. Benner, E. S. Quintana-Ortí, and G. Quintana-Ortí. Solving linear and quadratic matrix equations on distributed memory parallel computers. In *IEEE International Symposium on Computer Aided Control System Design*, pages 64–69, 1999.

[12] M. W. Berry, J. J. Dongarra, and Y. Kim. A parallel algorithm for the reduction of a nonsymmetric matrix to block upper-Hessenberg form. *Parallel Comput.*, 21(8):1189–1211, 1995.

[13] P. Bientinesi, I. S. Dhillon, and R. A. van de Geijn. A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. *SIAM J. Sci. Comput.*, 27(1):43–66, 2005.

[14] 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.

[15] D. Boley and R. Maier. A parallel QR algorithm for the nonsymmetric eigenvalue problem. Technical report TR-88-12, Department of Computer Science, University of Minnesota at Minneapolis, 1988.

[16] K. Braman. Middle deflations in the QR algorithm. Talk at Householder Symposium XVII, June 2008.

[17] 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.

[18] 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.

[19] R. Byers. Solving the algebraic Riccati equation with the matrix sign function. *Linear Algebra Appl.*, 85:267–279, 1987.

[20] R. Byers. LAPACK 3.1 xHSEQR: Tuning and Implementation Notes on the Small Bulge Multi-shift QR Algorithm with Aggressive Early Deflation, 2007. LAPACK Working Note 187.

[21] J. Choi, J. J. Dongarra, and D. W. Walker. The design of a parallel dense linear algebra software library: reduction to Hessenberg, tridiagonal, and bidiagonal form. *Numer. Algorithms*, 10(3-4):379–399, 1995.

[22] K. Dackland. Parallel reduction of a regular matrix pair to block-Hessenberg-triangular form – algorithm design and performance modelling. Report UMINF-98.09, Department of Computing Science, Umeå University, Sweden, 1998.

[23] 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.

[24] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

[25] P. J. Eberlein. On the Schur decomposition of a matrix for parallel computation. *IEEE Trans. Comput.*, C-36:167–174, 1987.

[26] J. G. F. Francis. The QR transformation, Parts I and II. *Computer Journal*, 4:265–271, 332–345, 1961, 1962.

[27] G. A. Geist and G. J. Davis. Finding eigenvalues and eigenvectors of unsymmetric matrices using a distributed memory multiprocessor. *Parallel Comput.*, 13:199–209, 1990.

[28] G. A. Geist, R. C. Ward, G. J. Davis, and R. E. Funderlic. Finding eigenvalues and eigenvectors of unsymmetric matrices using a hypercube multiprocessor. In G. Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 1577–1582, 1988.

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

[30] GOTO-BLAS – high-performance BLAS by Kazushige Goto. See `http://www.cs.utexas.edu/users/flame/goto/`.

[31] R. Granat and B. Kågström. Algorithm XXX: The SCASY software library – parallel solvers for Sylvester-type matrix equations with applications in condition estimation, Part II. 2009. *ACM Transactions on Mathematical Software* (submitted July 2007, revised January 2009).

[32] R. Granat and B. Kågström. Parallel solvers for Sylvester-type matrix equations with applications in condition estimation, Part i: Theory and algorithms. 2009. *ACM Transactions on Mathematical Software* (submitted July 2007, revised January 2009).

[33] R. Granat, B. Kågström, and D. Kressner. Parallel eigenvalue reordering in real Schur forms. *Concurrency and Computation: Practice and Experience*, 2008. To appear.

[34] R. Granat, B. Kågström, and D. Kressner. A parallel Schur method for solving continuous-time algebraic Riccati equations. 2008. Accepted for IEEE International Symposium on Computer-Aided Control Systems Design, San Antonio, Texas, 2008.

[35] G. Henry and R. Van de Geijn. Parallelizing the QR algorithm for the unsymmetric algebraic eigenvalue problem: Myths and reality. *SIAM J. Sci. Comput.*, 17:870–883, 1997.

[36] 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.

[37] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, second edition, 2002.

[38] HPC2N - High Performance Computing Center North. See `http://www.hpc2n.umu.se`.

[39] 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.

[40] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Software*, 24(3):268–302, 1998.

[41] L. Karlsson and B. Kågström. A Framework for Dynamic Node Scheduling of Two-Sided Blocked Matrix Computations. In *State of the Art in Scientific and Parallel Computing, PARA 2008, Lecture Notes in Computer Science* (to appear). Springer, 2009.

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

[43] D. Kressner. Block algorithms for reordering standard and generalized Schur forms. *ACM TOMS*, 32(4):521–532, December 2006.

[44] D. Kressner. The effect of aggressive early deflation on the convergence of the QR algorithm. *SIAM J. Matrix Anal. Appl.*, 30(2):805–821, 2008.

[45] V. N. Kublanovskaya. On some algorithms for the solution of the complete eigenvalue problem. *USSR Comp. Math Phys.*, 3:637–657, 1961.

[46] B. Lang. Effiziente Orthogonaltransformationen bei der Eigen- und Singulärwertzerlegung. Habilitationsschrift, 1997.

[47] B. Lang. Using level 3 BLAS in rotation-based algorithms. *SIAM J. Sci. Comput.*, 19(2):626–634, 1998.

[48] A. J. Laub. A Schur method for solving algebraic Riccati equations. *IEEE Trans. Automat. Control*, AC-24:913–921, 1979.

[49] H. Ltaief, J. Kurzak, and J. Dongarra. LAPACK working note 214: Scheduling two-sided transformations using algorithms-by-tiles on multicore architectures. Technical report, Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, 2009.

[50] R. M. Martin. *Electronic Structure: Basic Theory and Practical Methods*. Cambridge University Press, 2004.

[51] Matrix Market - A visual repository of test data for use in comparative studies of algorithms for numerical linear algebra. See `http://math.nist.gov/MatrixMarket/`.

[52] MPI - Message Passing Interface. See `http://www-unix.mcs.anl.gov/mpi/`.

[53] PBLAS - Parallel Basic Linear Algebra Subprograms. See `http://www.netlib.org/scalapack/`.

[54] J. D. Roberts. Linear model reduction and solution of the algebraic Riccati equation by use of the sign function. *Internat. J. Control*, 32(4):677–687, 1980.

[55] T. Schreiber, P. Otto, and F. Hofmann. A new efficient parallelization strategy for the QR algorithm. *Parallel Comput.*, 20:63–75, 1994.

[56] ScaLAPACK Users' Guide. See `http://www.netlib.org/scalapack/slug/`.

[57] G. W. Stewart. A parallel implementation of the QR algorithm. *Parallel Comput.*, 9:187–196, 1987.

[58] R. A. van de Geijn. Storage schemes for parallel eigenvalue algorithms. In G. Golub and P. Van Dooren, editors, *Numerical Linear Algebra Digital Signal Processing and Parallel Algorithms*, pages 639–648. Springer-Verlag, 1988.

[59] R. A. van de Geijn. Deferred shifting schemes for parallel QR methods. *SIAM J. Matrix Anal. Appl.*, 14:180–194, 1993.

[60] R. A. van de Geijn and D. G. Hudson. An efficient parallel implementation of the nonsymmetric QR algorithm. In *Fourth Conference on Hypercube Concurrent Computers and Applications, Monterey, CA*, pages 697–700, 1989.

[61] D. Viswanath and L. N. Trefethen. Condition numbers of random triangular matrices. *SIAM J. Matrix Anal. Appl.*, 19(2):564–581 (electronic), 1998.

[62] V. Volkov and J. W. Demmel. LAPACK working note 197: Using GPUs to accelerate the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices. Technical Report UCB/EECS-2007-179, EECS Department, University of California, Berkeley, 2007.

[63] D. S. Watkins. Shifting strategies for the parallel QR algorithm. *SIAM J. Sci. Comput.*, 15(4):953–958, 1994.

[64] D. S. Watkins. Forward stability and transmission of shifts in the QR algorithm. *SIAM J. Matrix Anal. Appl.*, 16(2):469–487, 1995.

[65] D. S. Watkins. The transmission of shifts and shift blurring in the QR algorithm. *Linear Algebra Appl.*, 241/243:877–896, 1996.

[66] D. S. Watkins. A case where balancing is harmful. *Electron. Trans. Numer. Anal.*, 23:1–4, 2006.

[67] D. S. Watkins. *The matrix eigenvalue problem*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2007. GR and Krylov subspace methods.

[68] D. S. Watkins. The QR algorithm revisited. *SIAM Rev.*, 50(1):133–145, 2008.

[69] R. C. Whaley. Empirically tuning LAPACK's blocking factor for increased performance. In *Proc. International Multiconference on Computer Science and Information Technology (IMCSIT)*, volume 3, pages 303–310, 2008.

[70] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27:3–35, 2001.

[71] Y. Yamamoto. Performance modeling and optimal block size selection for the small-bulge multishift QR algorithm. In M. Guo, L. T. Yang, B. Di Martino, H. Zima, J. J. Dongarra, and F. Tang, editors, *4th International Symposium on Parallel and Distributed Processing and Applications, Sorrento, Italy*, Lecture Notes in Computer Science, pages 451–463. Springer-Verlag, 2006.

A Supplementary numerical data

In the following, we present additional figures and tables to illustrate the performance of our new parallel QR algorithm implemented in `PDHSEQR`. See Section 4.3 for the definition of the performance metrics S_p.

A.1 Speedup and scalability for random matrices

Tables 7–10 contain the values of S_p for the numerical experiments with random matrices reported in Section 4.5. In particular, we want to point out that the reported values of S_p(`PDLAHQR/PDHSEQR`) are kept roughly constant when the number of utilized cores p and the problem size n are scaled up simultaneously such that the memory load on each core is kept fixed (see, e.g., $n = 4,000,\ p = 1 \times 1 \times 1$ and $n = 16,000,\ p = 4 \times 4 \times 1$ in Table 7, and $n = 6,000,\ p = 1 \times 1 \times 1$ and $n = 24,000,\ p = 4 \times 4 \times 1$ in Table 9).

Table 7: S_p(`PDLAHQR`), S_p(`PDHSEQR`) and S_p(`PDLAHQR/PDHSEQR`) on *akka* for `fullrand`.

$p = P_r \times P_c \times P_t$	$n =$ 4 000			8 000			16 000			32 000		
$1 \times 1 \times 1$	1.0	1.0	41.3	1.0	1.0	54.1						
$2 \times 2 \times 1$	3.4	2.0	24.7	4.0	2.3	31.5						
$4 \times 4 \times 1$	7.9	3.3	17.0	9.3	5.6	32.4	1.0	1.0	41.2			
$6 \times 6 \times 1$	13.4	3.8	11.6	15.4	7.6	26.8	2.1	1.6	32.7	∞	1.0	N/A
$8 \times 8 \times 1$	22.3	4.0	7.3	21.8	9.7	24.2	3.2	2.8	35.1	∞	1.6	N/A
$10 \times 10 \times 1$	25.4	3.6	5.8	30.9	9.0	15.7	4.0	3.2	32.5	∞	2.0	N/A
$1 \times 1 \times 4$	3.6	1.3	14.9	4.4	1.6	20.1						
$2 \times 2 \times 4$	5.8	1.8	12.8	7.3	2.5	18.7						
$3 \times 3 \times 4$	11.2	2.2	8.2	14.0	3.6	14.0	1.7	0.7	16.7			
$4 \times 4 \times 4$	15.2	3.0	8.1	20.0	5.3	14.4	2.6	1.2	19.4	∞	0.7	N/A
$5 \times 5 \times 4$	19.7	3.2	6.8	26.9	7.3	13.3	3.6	1.5	17.8	∞	0.9	N/A

Table 8: S_p(`PDLAHQR`), S_p(`PDHSEQR`) and S_p(`PDLAHQR/PDHSEQR`) on *akka* for `hessrand`.

$p = P_r \times P_c \times P_t$	$n =$ 4 000			8 000			16 000			32 000		
$1 \times 1 \times 1$	1.0	1.0	13.0	1.0	1.0	11.5						
$2 \times 2 \times 1$	2.7	2.1	10.2	3.2	2.3	8.3						
$4 \times 4 \times 1$	4.2	4.2	13.0	5.4	6.3	13.5	1.0	1.0	26.3			
$6 \times 6 \times 1$	7.1	5.0	9.2	9.2	9.5	11.9	1.3	1.8	36.6	∞	1.0	N/A
$8 \times 8 \times 1$	10.0	6.3	8.2	12.1	11.6	11.0	2.0	2.9	38.1	∞	1.3	N/A
$10 \times 10 \times 1$	12.8	5.4	5.5	14.6	13.2	10.4	2.6	3.3	32.9	∞	1.4	N/A
$1 \times 1 \times 4$	3.5	1.3	4.7	3.4	1.5	5.0						
$2 \times 2 \times 4$	4.8	2.0	5.5	6.2	2.7	5.0	1.1	0.6	14.5			
$3 \times 3 \times 4$	8.1	2.6	4.3	10.3	3.9	4.4	1.9	0.8	11.0			
$4 \times 4 \times 4$	11.0	3.5	4.2	13.6	5.1	4.3	2.9	1.4	12.7	∞	1.0	N/A
$5 \times 5 \times 4$	12.7	4.7	4.8	18.4	8.5	5.3	3.2	1.5	12.2	∞	1.1	N/A

Table 9: S_p(PDLAHQR), S_p(PDHSEQR) and S_p(PDLAHQR/PDHSEQR) on *sarek* for `fullrand`.

$p = P_r \times P_c \times P_t$	$n =$ 6 000			12 000			24 000			48 000		
$1 \times 1 \times 1$	1.0	1.0	50.5									
$2 \times 2 \times 1$	4.1	2.4	29.9	1.0	1.0	45.8						
$4 \times 4 \times 1$	15.1	4.7	15.7	4.1	2.6	29.1	1.0	1.0	49.9			
$6 \times 6 \times 1$	29.9	5.9	10.0	8.5	4.5	24.2	2.3	1.8	39.3	∞	1.0	N/A
$8 \times 8 \times 1$	50.0	6.5	6.6	12.5	5.0	18.3	N/A	2.5	N/A	∞	1.6	N/A
$1 \times 1 \times 2$	2.2	1.5	24.1									
$2 \times 2 \times 2$	8.0	3.2	21.3	2.1	1.5	33.0						
$4 \times 4 \times 2$	27.3	6.0	14.7	6.3	3.7	26.8	2.4	1.4	28.2			
$6 \times 6 \times 2$	54.2	7.3	11.7	10.2	5.5	24.9	N/A	2.4	N/A	∞	1.7	N/A
$8 \times 8 \times 2$	78.6	8.1	9.2	23.6	6.2	12.0	8.1	3.4	20.2	∞	2.4	N/A

Table 10: S_p(PDLAHQR), S_p(PDHSEQR) and S_p(PDLAHQR/PDHSEQR) on *sarek* for `hessrand`.

$p = P_r \times P_c \times P_t$	$n =$ 6 000			12 000			24 000			48 000		
$1 \times 1 \times 1$	1.0	1.0	9.8									
$2 \times 2 \times 1$	2.8	2.7	9.6	1.0	1.0	28.8						
$4 \times 4 \times 1$	7.7	5.4	6.9	3.3	2.6	22.2	N/A	1.0	N/A			
$6 \times 6 \times 1$	12.1	7.6	6.1	5.5	4.9	26.1	N/A	1.6	N/A	∞	1.0	N/A
$8 \times 8 \times 1$	17.6	9.0	5.0	7.8	5.7	21.1	N/A	2.0	N/A	∞	1.4	N/A
$1 \times 1 \times 2$	2.1	1.6	7.8									
$2 \times 2 \times 2$	5.2	3.4	6.4	2.1	1.3	17.9						
$4 \times 4 \times 2$	13.9	6.8	4.8	6.5	3.5	15.5	N/A	1.6	N/A			
$6 \times 6 \times 2$	21.4	8.9	4.1	10.6	5.2	14.1	N/A	2.4	N/A	∞	1.7	N/A
$8 \times 8 \times 2$	30.0	11.2	3.6	14.4	7.4	14.9	N/A	3.0	N/A	∞	2.2	N/A

A.2 Performance, speedup and scalability for benchmark examples

Table 11 summarizes benchmark examples on which we have performed additional numerical experiments. To keep the presentation compact, we have chosen to run all benchmarks in this section on one of the target platforms only, always using one MPI process per allocated core. Most of the benchmarks are available as data files, in a special *Matrix Market format*, or as (serial) matrix generators, using compressed column (or row) storage, on Matrix Market [51]. For the purpose of running these benchmarks, specialized routines were developed that read in and distribute the data, or generate and distribute the problem across the process mesh.

A.2.1 BBMSN

AED is at its best and deflates a significant amount of eigenvalues right away, eliminating any need for doing QR sweeps. This observation was already made in the original paper [18] of AED. To some extent, this effect is also observed for random matrices of the class `hessrand`, see Tables 3 and 5. Since AED is performed on a small submatrix, the scalability – but certainly not the performance – is negatively affected if only AED and almost no QR iterations need to be performed. For the matrix sizes considered in Figure 8, `PDHSEQR` applied to BBSM is two orders of magnitude faster than `PDLAHQR`.

Table 11: Considered benchmark examples.

Name	Dimension	Type/Structure	Hess. req.	Ref.
BBMSN	n	$S_n = \begin{bmatrix} n & n-1 & n-2 & & 2 & 1 \\ 10^{-3} & 1 & 0 & \cdots & 0 & 0 \\ & 10^{-3} & 2 & & 0 & 0 \\ & & 10^{-3} & \ddots & 0 & 0 \\ & & & \ddots & n-2 & 0 \\ & & & & 10^{-3} & n-1 \end{bmatrix}$	No	[18]
AF23560	23560	Computational fluid dynamics	Yes	[5]
CRYG10000	10000	Material science	Yes	[5]
OLM5000	5000	Computational fluid dynamics	Yes	[5]
DW8192	8192	Electrical engineering	Yes	[5]
MATRAN	n	Sparse random matrix	Yes	[5]
MATPDE	n	Partial differential equations	Yes	[5]
GRCAR	n	$G_n = \begin{bmatrix} 1 & 1 & 1 & 1 & & & & \\ -1 & 1 & 1 & 1 & 1 & & & \\ & -1 & 1 & 1 & 1 & 1 & & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & -1 & 1 & 1 & 1 & 1 \\ & & & & -1 & 1 & 1 & 1 \\ & & & & & -1 & 1 & 1 \\ & & & & & & -1 & 1 \end{bmatrix}$	No	[5]

For the residuals defined in Section 4.4, PDHSEQR typically returns with $R_r \approx 10^{-15}$ while PDLAHQR delivers $R_r \approx 10^{-13}$. The average numbers of shifts for one eigenvalue to converge in PDHSEQR and PDLAHQR are 0.00 and 1.80, respectively.

A.2.2 AF23560

This test matrix of order 23560 arises in the transient stability analysis of a Navier-Stokes equation. Figure 9 displays the measured execution times and performance metrics, while Figure 10 contains a performance profile. The scaling of PDHSEQR is somewhat disappointing, due to the fact that a large fraction of the time is spent for AED. On the other hand, in terms of S_p(PDLAHQR/PDHSEQR), PDHSEQR is superior with a speedup of about 74, 55 and 52 for 64, 100 and 144 cores, respectively.

For this benchmark, the accuracy degrades to a certain degree: both routines typically return $R_r \in [10^{-10}, 10^{-14}]$, but there are some exceptions for which PDLAHQR returns $R_r \approx 10^{-5}$. The average numbers of shifts for one eigenvalue to converge in PDHSEQR and PDLAHQR are 0.35 and 6.51, respectively.

A.2.3 CRYG10000

This is a nonsymmetric eigenvalue problem of order 10000 that arises from the stability analysis of a crystal growth problem. A comparison of the performance of PDLAHQR and PDHSEQR is presented in Figure 11. Both implementations exhibit quite similar scalability, but PDHSEQR is much faster, about $23 - 36$ times for a moderate number of cores. The corresponding profile is displayed in Figure 12.

With respect to the residuals, PDHSEQR returns $R_r \approx 10^{-15}$ while PDLAHQR returns $R_r \approx 10^{-14}$. The average numbers of shifts for one eigenvalue to converge in PDHSEQR and PDLAHQR are 0.26 and 2.50, respectively.

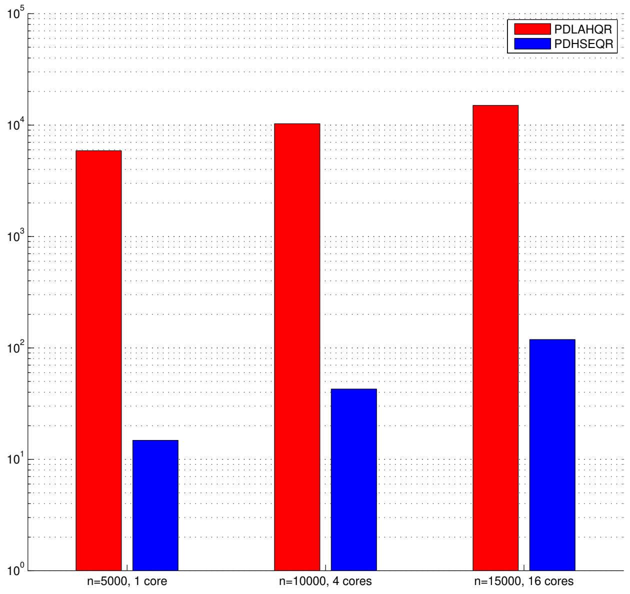


Figure 8: T_p for PDLAHQR and PDHSEQR applied to BBMSN on *akka*.

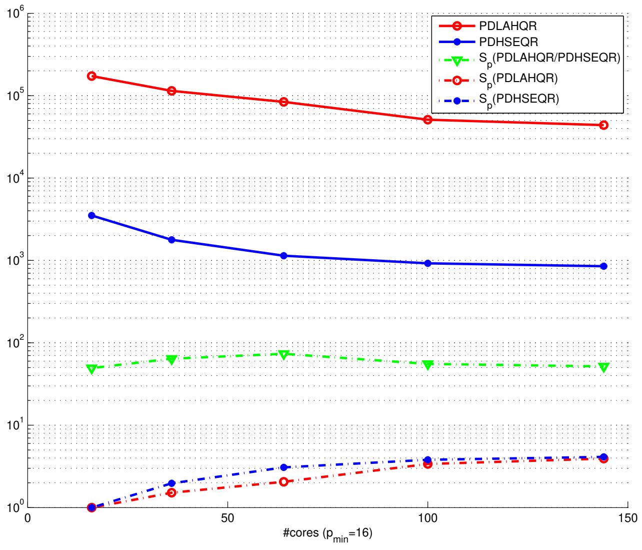


Figure 9: T_p and S_p for PDLAHQR and PDHSEQR applied to AF23560 on *akka*.

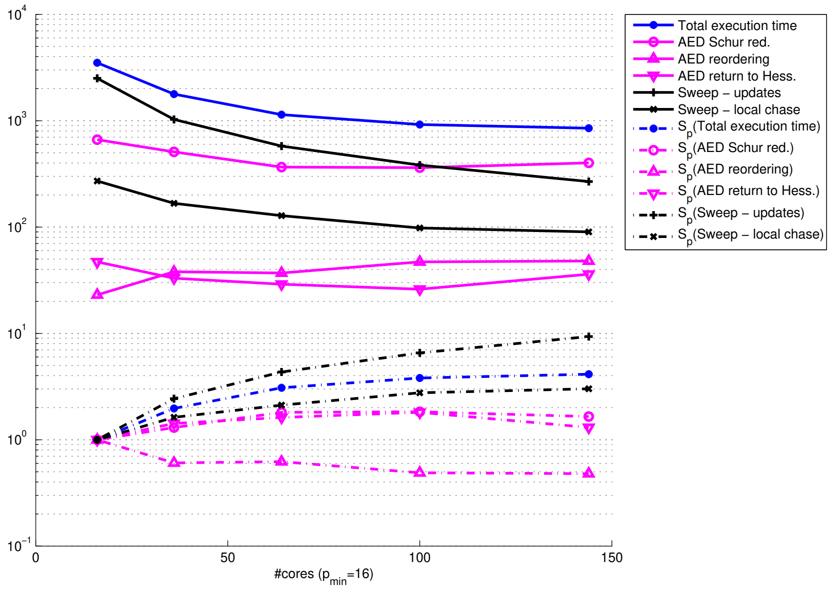


Figure 10: Profile of `PDHSEQR` applied to AF23560 on *akka*.

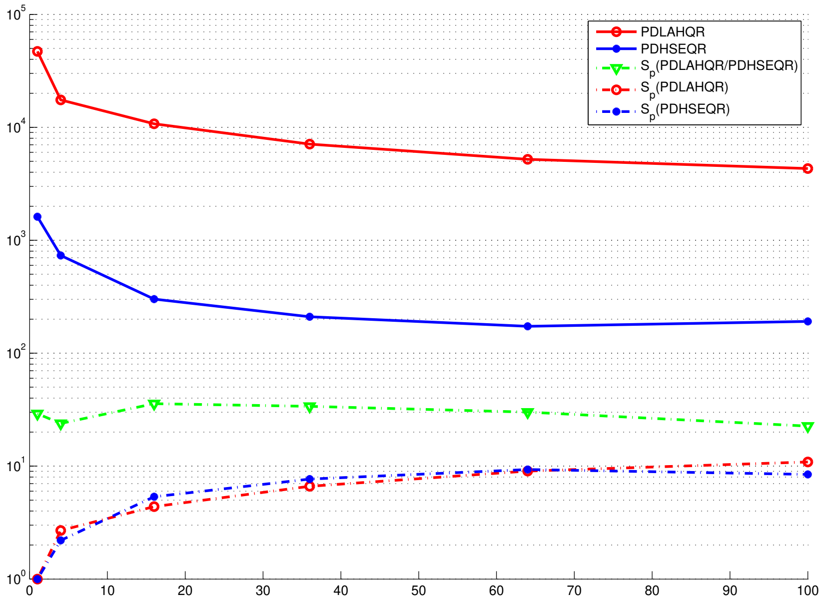


Figure 11: T_p and S_p for `PDLAHQR` and `PDHSEQR` applied to CRYG10000 on *akka*.

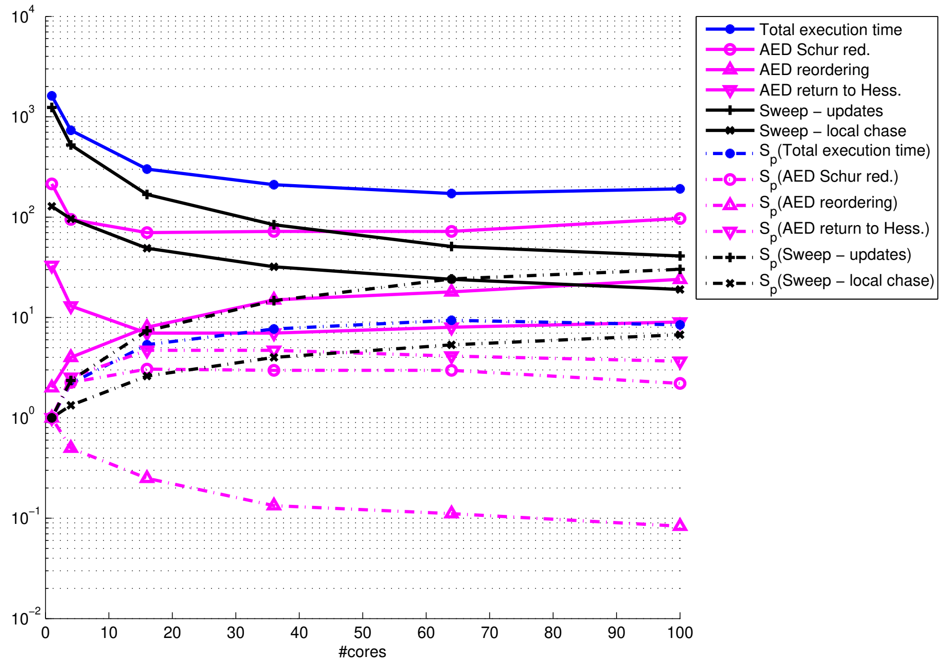


Figure 12: Profile of PDHSEQR applied to CRYG10000 benchmark on *akka*.

A.2.4 OLM5000

The Olmstead model represents the flow of a layer of viscoelastic fluid heated from below. The discretized equations give rise to a matrix of size 5000. Since this is a relatively small eigenvalue problem for a parallel solver, we expect the performance gain of using PDHSEQR instead of PDLAHQR to be less remarkable, especially for larger process meshes. This expectation is certainly met, see Figure 13. Nevertheless, there is a notable speedup even though the scaling of PDHSEQR deteriorates for more than 64 cores.

With respect to the residuals, PDHSEQR returns $R_r \approx 10^{-15}$ while PDLAHQR returns $R_r \approx 10^{-14}$. The average numbers of shifts for one eigenvalue to converge in PDHSEQR and PDLAHQR are 0.72 and 3.46, respectively.

A.2.5 DW8192

A finite difference discretization of the Helmholtz equation governing the magnetic field associated with a dielectric channel waveguide problem, which arises in many integrated circuit applications, leads to a 8192×8192 generalized nonsymmetric eigenvalue problem. A standard nonsymmetric eigenvalue problem is obtained by explicit inversion. Again, this is a relatively small-sized eigenvalue problem for larger process meshes. Figure 14 clearly reflects this. Note, however, that the ratio between PDLAHQR and PDHSEQR is larger than for OLM5000.

With respect to the residuals, PDHSEQR returns $R_r \approx 10^{-15}$ while PDLAHQR returns $R_r \approx 10^{-14}$. The average numbers of shifts for one eigenvalue to converge in PDHSEQR and PDLAHQR are 0.58 and 3.42, respectively.

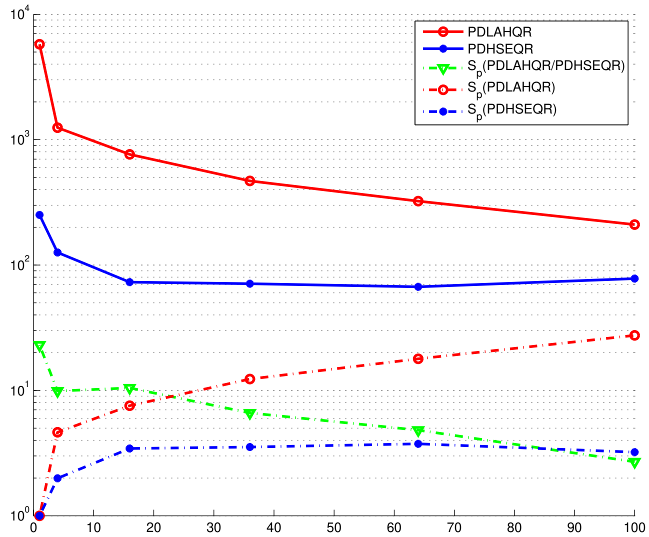


Figure 13: T_p and S_p for `PDLAHQR` and `PDHSEQR` applied to OLM5000 on *akka*. Note that with 4 cores, the time for `PDLAHQR` is measured for multithreading on one node, which turned out to be faster than using 4 MPI processes inside the node.

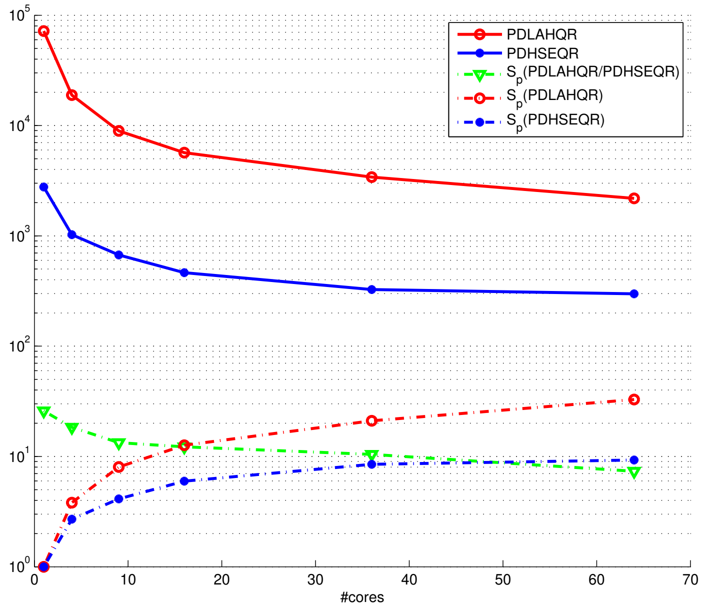


Figure 14: T_p and S_p for `PDLAHQR` and `PDHSEQR` applied to DW8192 on *sarek*.

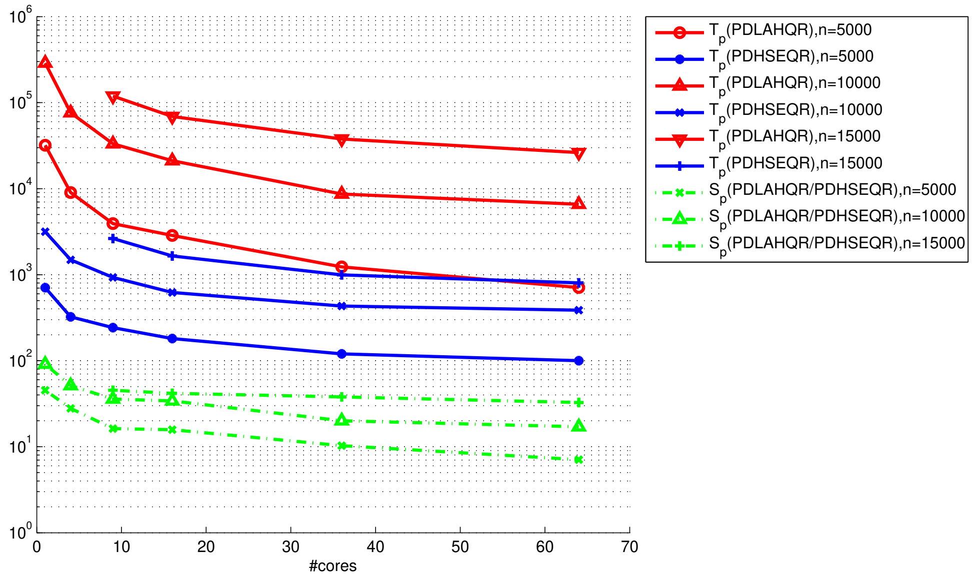


Figure 15: T_p and S_p for `PDLAHQR` and `PDHSEQR` applied to MATRAN on *sarek*.

A.2.6 MATRAN

This benchmark generator produces sparse matrices whose nonzero entries are uniformly distributed on the interval $(-1, 1)$. In our tests, we choose the number of non-zeros per column to be $\max(1, \lfloor n/100 \rfloor)$. We present the performance of `PDLAHQR` and `PDHSEQR` solving this eigenproblem for $n = 5000, 10000, 15000$ in Figure 15. The speedup is significant (always above 30) for $n = 15000$.

With respect to the residuals, both `PDHSEQR` and `PDLAHQR` return $R_r \approx 10^{-14}$. The average numbers of shifts for one eigenvalue to converge in `PDHSEQR` and `PDLAHQR` are 0.65 and 3.85 for $n = 5000$, 0.52 and 4.03 for $n = 10000$, 0.40 and 5.26 for $n = 15000$, respectively.

A.2.7 MATPDE

This example arises from a five-point central finite difference discretization of a 2D variable-coefficient linear elliptic PDE, using $n_x = n_y$ grid points in each spatial dimension. The resulting $n \times n$ matrix with $n = n_x \cdot n_y$ is block tridiagonal. All other parameters of the generator are set to their default values, see [5]. This eigenvalue problem is solved for $n = 10000$ ($n_x = n_y = 100$), $n = 14400$ ($n_x = n_y = 120$) and $n = 19600$ ($n_x = n_y = 140$) on *akka*. The obtained results are presented in Figure 16. Once again, the gain of using `PDHSEQR` instead of `PDLAHQR` is significant – a speedup of $32 - 38$ is obtained for the largest problem ($n = 19600$).

With respect to the residuals, both `PDHSEQR` and `PDLAHQR` return with $R_r \approx 10^{-14}$. The average numbers of shifts for one eigenvalue to converge in `PDHSEQR` and `PDLAHQR` are 0.57 and 3.23 for $n = 10000$, 0.52 and 8.34 for $n = 14400$, 0.54 and 8.83 for $n = 19600$, respectively.

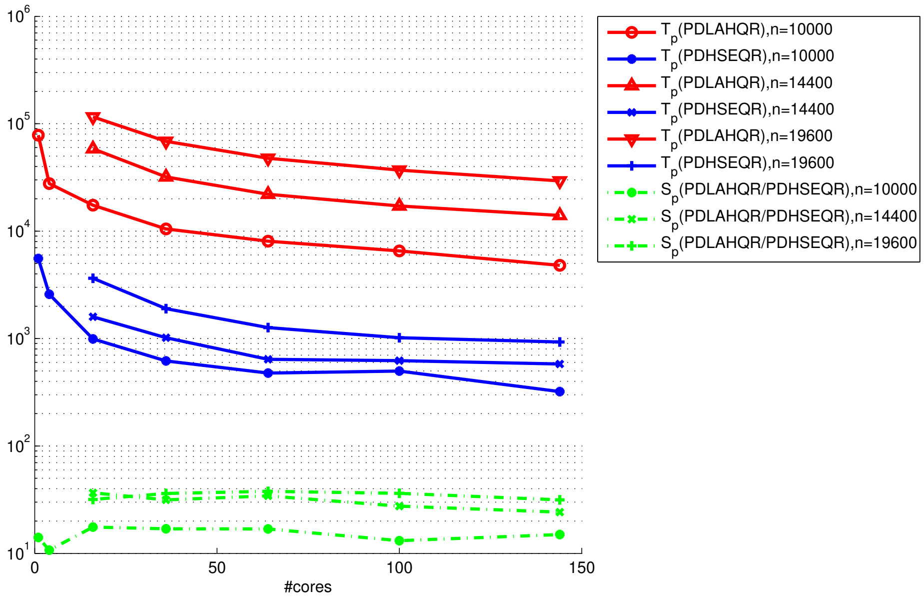


Figure 16: T_p and S_p for `PDLAHQR` and `PDHSEQR` applied to MATPDE benchmark on *akka*.

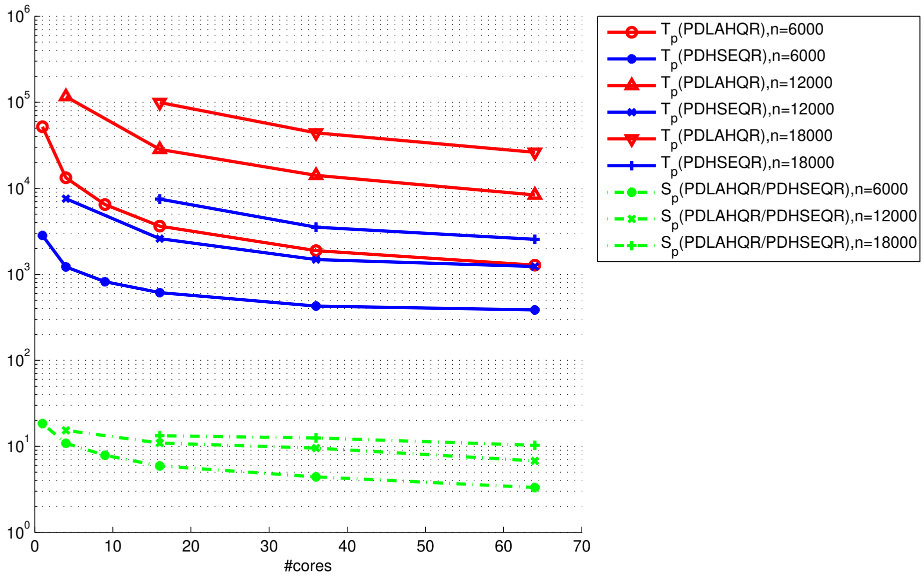


Figure 17: T_p and S_p for `PDLAHQR` and `PDHSEQR` applied to GRCAR on *sarek*.

A.2.8 GRCAR

An $n \times n$ Grcar matrix is a nonsymmetric Toeplitz matrix with very ill-conditioned eigenvalues. For this matrix, PDHSEQR is between 2–20 times faster than PDLAHQR depending on the problem size and the number of cores, see Figure 17.

With respect to the residuals, PDHSEQR returns $R_r \approx 10^{-14}$ while PDLAHQR returns $R_r \in [10^{-13}, 10^{-14}]$. The average numbers of shifts for one eigenvalue to converge in PDHSEQR and PDLAHQR are 1.22 and 3.36 for $n = 6000$, 1.52 and 3.69 for $n = 12000$, 1.29 and 3.83 for $n = 18000$, respectively.