Parallel Library Software for the Multishift QR Algorithm with Aggressive Early Deflation

By

Robert Granat, Bo Kågström, Daniel Kressner, and Meiyue Shao

UMINF-12/06

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
Sweden
Parallel Library Software for the Multishift QR Algorithm with Aggressive Early Deflation

Robert Granat, Bo Kågström, Daniel Kressner, and Meiyue Shao

June 3, 2014

Abstract

Library software implementing a parallel small-bulge multishift QR algorithm with aggressive early deflation (AED) targeting distributed memory high-performance computing systems is presented. Starting from recent developments of the parallel multishift QR algorithm [Granat et al., SIAM J. Sci. Comput. 32(4), 2010], we describe a number of algorithmic and implementation improvements. These include communication avoiding algorithms via data redistribution and a refined strategy for balancing between multishift QR sweeps and AED. Guidelines concerning several important tunable algorithmic parameters are provided. As a result of these improvements, a computational bottleneck within AED has been removed in the parallel multishift QR algorithm. A performance model is established to explain the scalability behavior of the new parallel multishift QR algorithm. Numerous computational experiments confirm that our new implementation significantly outperforms previous parallel implementations of the QR algorithm.

Keywords: Multishift QR algorithm, aggressive early deflation, parallel algorithms, distributed memory architectures

1 Introduction

The QR algorithm is the method of choice for computing all eigenvalues of a nonsymmetric matrix \( A \in \mathbb{R}^{n \times n} \). This paper describes a novel parallel implementation of the multishift QR algorithm for distributed memory architectures. While our implementation is largely based on the algorithms described in [21], a number of additional algorithmic improvements have been made, leading to significantly reduced execution times and higher robustness.

In the following, we give a brief history of serial and parallel implementations of the QR algorithm. The algol procedure hqr by Martin, Petersen, and Wilkinson [32] was among the first computer implementations of the QR algorithm. A Fortran translation of this procedure was included in EISPACK [35] as routine HQR. The initial version of the LAPACK routine DHSEQR was based on work by Bai and Demmel [5]; the most notable difference to HQR was the use of multishift techniques to improve data locality. This routine had only seen a few minor modifications [2] until LAPACK version 3.1, when it was replaced by an implementation incorporating pipelined bulges and aggressive early deflation techniques from the works by Braman, Byers, and Mathias [12, 13]. This implementation is described in more detail in [14]. While there has been a lot of early work on parallelizing the QR algorithm, for example in [23, 33, 37, 38, 39, 40], the first publicly available parallel implementation was released only 1997 in ScaLAPACK [11] version 1.5 as routine PDLAHQR, based on work by Henry, Watkins, and Dongarra [24]. A complex version PZLAHQR of this routine was included later on [16]. In this work, we describe...
a new parallel implementation of the QR algorithm that aims to replace PDLAHQR. It might be interesting to note that all recently released high-performance linear algebra packages, such as MAGMA and PLASMA [1], ELPA [3], FLAME [10] lack adapted implementations of the QR algorithm or other nonsymmetric eigenvalue solvers.

Given a nonsymmetric matrix \( A \), the parallel implementation of the eigenvalue solver in ScaLAPACK consists of the following steps. In the first optional step, the matrix is balanced, that is, an invertible diagonal matrix \( D \) is computed to make the rows and columns of \( A \leftarrow D^{-1} A D \) as close as possible. In the second step, \( A \) is reduced to Hessenberg form: \( H = Q_0^T A Q_0 \) with an orthogonal matrix \( Q_0 \) and \( h_{ij} = 0 \) for \( i \geq j + 2 \). In the third step, the QR algorithm iteratively reduces \( H \) further to real Schur form, eventually resulting in an orthogonal matrix \( Q \) such that

\[
T = Q^T H Q \tag{1}
\]

is quasi-upper triangular. This means that \( T \) is block upper triangular with \( 1 \times 1 \) blocks (corresponding to real eigenvalues) and \( 2 \times 2 \) blocks (corresponding to complex conjugate eigenvalue pairs) on the diagonal. Therefore, the Schur decomposition of \( A \) is \( A = ZTZ^T \), where \( Z = Q_0 Q \). The last optional step consists of computing the eigenvectors of \( T \) and performing a back transformation to obtain the eigenvectors of the original matrix \( A \).

This paper is only concerned with the reduction to real Schur form (1). In particular, we will not discuss the implementation of Hessenberg reduction, see [31, 29, 36] for recent developments in this direction.

The rest of this paper is organized as follows. Section 2 provides a summary of our implementation and the underlying algorithm, emphasizing improvements over [21]. In Section 3, we present a performance model that provides insights into the cost of computations and communication. The derivation of this model is given in the electronic appendix. The performance model is then used to guide the choice of the parameters in Section 4. Finally, in Section 5, we evaluate the performance of our parallel library software by a large set of numerical experiments.

### 2 Algorithms and Implementation

Modern variants of the QR algorithm usually consist of two major components—multishift QR sweeps and aggressive early deflation (AED). A typical structure of the modern QR algorithm in a sequential or parallel setting is provided in Algorithm 1. Although our parallel multishift QR algorithm also builds on Algorithm 1, there are several issues to be considered for reaching high performance. Figure 1 shows the software hierarchy of our implementation of the parallel multishift QR algorithm. Details of the algorithm and some implementation issues are discussed in the successive subsections.

**Algorithm 1 Multishift QR algorithm with AED**

**Input:** \( H \in \mathbb{R}^{n \times n} \), \( H \) is upper Hessenberg.

**Output:** A real Schur form of \( H \).

1. while not converged do
2. Perform AED on the \( n_{aed} \times n_{aed} \) trailing principle submatrix.
3. Apply the accumulated orthogonal transformation to the corresponding off-diagonal blocks.
4. if a large fraction of eigenvalues has been deflated in Step 2 then
5. goto Step 2.
6. end if
7. Perform a small-bulge multishift QR sweep with \( n_{shift} \) undeflatable eigenvalues obtained from Step 2 as shifts.
8. Check for negligible subdiagonal elements.
9. end while
2.1 Data layout convention in ScaLAPACK

In ScaLAPACK, the \( p = p_r p_c \) processors are usually arranged into a \( p_r \times p_c \) grid. Matrices are distributed over the rectangular processor grid in a 2D block-cyclic layout with block size \( m_b \times n_b \) (see an example in Figure 2). The information regarding the data layout is stored in an array descriptor so that the mapping between entries of the global matrix and their corresponding locations in the memory hierarchy can be established. We adopt ScaLAPACK's data layout convention and require that the \( n \times n \) input matrices \( H \) and \( Z \) have identical data layout with square data blocks (i.e., \( m_b = n_b \)). However, the processor grid need not to be square unless explicitly specified.

2.2 Multishift QR sweep

The multishift QR sweep is a bulge chasing process that involves several shifts. The QR sweep applied to a Hessenberg matrix \( H \) with \( k \) shifts \( \sigma_1, \sigma_2, \ldots, \sigma_k \) yields another Hessenberg matrix \( Q^T H Q \) where \( Q \) is determined by the QR decomposition of the shift polynomial:

\[
(H - \sigma_1 I)(H - \sigma_2 I) \cdots (H - \sigma_k I) = QR.
\]
Figure 3: Loosely coupled shifts v.s. tightly coupled shifts. The dashed lines represent borders of the processor grid. Only parts of the matrix are displayed. Intrablock bulge chasings can be performed independently and in parallel.

Thanks to the Implicit Q theorem (e.g., see [17, 18]), there is a lot of freedom regarding how to perform the QR sweep. In ScALAPACK v1.8.0 and earlier versions, PDLAHQR uses a pipelined approach that chases a chain of loosely coupled bulges, see Figure 3(a). While this approach clearly gives potential for parallelism, it comes with the disadvantage that its computational intensity is concentrated at level 1 and level 2 BLAS. In addition, frequent communication between processors is required. To avoid these shortcomings of the pipelined QR algorithm, we use several chains of tightly coupled bulges, see Figure 3(b), to improve node performance and reduce communication. For the new bulge chasing routine PDLAQR5, the total number of shifts ($n_{\text{shift}}$) used in a single QR sweep is shown in Table 1 and is usually much larger compared to the pipelined approach. These shifts are divided into several chains of tightly coupled bulges with up to $[n_b/3]$ shifts per chain so that the length of each chain does not exceed $n_b/2$. The chains are placed on different diagonal blocks, such that $\Theta(\sqrt{p})$ chains can be chased locally and simultaneously. The corresponding off-diagonal blocks are updated by explicit multiplication with the orthogonal matrix accumulated in the diagonal chasing step. This delay-and-accumulate technique leads to level 3 computational intensity. When the chains are passing through the processor border, they are chased in an odd-even manner (see Figure 4) to avoid conflicts between different tightly coupled chains. We refer to [21] for a detailed description of this technique.

2.3 Aggressive Early Deflation (AED)

The AED technique has been proposed by Braman, Byers, and Mathias [13] and proceeds by partitioning the current upper Hessenberg matrix $H \in \mathbb{R}^{n \times n}$ as

$$H = \begin{pmatrix}
n^{-n_{\text{AED}}-1} & 1 & n_{\text{AED}} \\
1 & \begin{pmatrix} H_{11} & H_{12} & H_{13} \\
H_{21} & H_{22} & H_{23} \\
0 & H_{32} & H_{33} \end{pmatrix} \end{pmatrix},$$
First round  

Second round

Figure 4: Interblock bulge chasing. Odd-numbered chains (left) and even-numbered chains (right) are chased separately in two rounds.

Table 1: Recommended values for $n_{\text{shift}}$ and $n_{\text{AED}}$. Values taken from Table 2.1 in [21] for $n \leq 96K$, values for $n > 96K$ are extrapolations. These values can be tuned by the user. However, such a tuning would not only need to take the computer architecture into account but also the balance between multishift QR iterations and AED, which depends on the particular matrix under consideration.

<table>
<thead>
<tr>
<th>$n$ (matrix size)</th>
<th>$n_{\text{shift}}$</th>
<th>$n_{\text{AED}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;6K$</td>
<td>see [14]</td>
<td></td>
</tr>
<tr>
<td>6K–12K</td>
<td>256</td>
<td>384</td>
</tr>
<tr>
<td>12K–24K</td>
<td>512</td>
<td>768</td>
</tr>
<tr>
<td>24K–48K</td>
<td>1024</td>
<td>1536</td>
</tr>
<tr>
<td>48K–96K</td>
<td>2048</td>
<td>3072</td>
</tr>
<tr>
<td>96K–192K</td>
<td>4096</td>
<td>6144</td>
</tr>
<tr>
<td>192K–384K</td>
<td>8192</td>
<td>12288</td>
</tr>
<tr>
<td>384K–768K</td>
<td>16384</td>
<td>24576</td>
</tr>
<tr>
<td>768K–1000K</td>
<td>32768</td>
<td>49152</td>
</tr>
<tr>
<td>$&gt;1M$</td>
<td>[n/25]</td>
<td>$3n_{\text{shift}}/2$</td>
</tr>
</tbody>
</table>
where $H_{33} \in \mathbb{R}^{n_{\text{AED}} \times n_{\text{AED}}}$ is the so-called AED window. By computing the (real) Schur decomposition $H_{33} = VTV^T$, and applying the corresponding similarity transformation to $H$, we obtain

$$U^T H U = \begin{pmatrix} H_{11} & H_{12} & H_{13}V \\ H_{21} & H_{22} & H_{23}V \\ 0 & s & T \end{pmatrix},$$

where

$$U = \begin{pmatrix} 1 & 1 \\ I & I \\ n_{\text{AED}} - n_{\text{AED}} - 1 & V \end{pmatrix}.$$  

The vector $s \in \mathbb{R}^{n_{\text{AED}}}$ is the so-called spike, created from the first entry of the vector $H_{12}$. The last diagonal entry (or $2 \times 2$ diagonal block) of $T$ can be deflated if the magnitude of the last component (or the last two components) of the spike is negligible. Undeletable eigenvalues are moved to the top left corner of $T$ by a swapping algorithm [6, 20]. The orthogonal transformations for reordering eigenvalues in the Schur form of the AED window are accumulated in an $n_{\text{AED}} \times n_{\text{AED}}$ orthogonal matrix. By repeating the same procedure to all diagonal entries (or $2 \times 2$ blocks) of $T$, the eigenvalues of $T$ are checked subsequently and possibly deflated. Then the entire matrix is reduced back to upper Hessenberg form and the off-diagonal blocks $H_{13}$ and $H_{23}$ are multiplied by $V$, the product of all involved orthogonal transformations. Typically, the size of the AED window is recommended to be somewhat larger, e.g., by 50%, than the number of shifts in the multishift QR sweeps [12, 14], see also Table 1.

In principle, aggressive early deflation can be incorporated into any variant of the QR algorithm. Therefore both the new multishift QR algorithm and the pipelined QR algorithm benefit from performing AED. As the efficient implementation of AED requires some care, we will now discuss these two settings in more detail.

### 2.3.1 AED within the new multishift QR algorithm

As this setting will be used for targeting large-scale problems, the AED window can be expected to become quite large. It is therefore not reasonable to expect that executing AED locally and sequentially yields good performance. Hence, the corresponding routine PDLAQR3 for performing AED requires a parallel approach.

The first and most costly step of AED is to calculate the Schur decomposition of $H_{33}$, the $n_{\text{AED}} \times n_{\text{AED}}$ trailing principal submatrix of $H$. This eigenvalue problem can be solved by either recursively using the new multishift QR algorithm (PDLAQR0) or using the pipelined QR algorithm (PDLAQR1). The choice of the solver is determined by the size of the AED window as well as the number of processors used. Since $n_{\text{AED}}$ is relatively small compared to $n$, the number of available processors may be too large to facilitate all of them without causing significant communication overhead. In this case, we only use a subset of the processors to reduce the overhead and minimizing the execution time. See Section 2.6 for a more detailed discussion.

In the deflation checking phase, the reordering algorithm is arranged in a blocked manner, to reduce memory transfers and communication. Unlike the procedure described in [12], undeletable eigenvalues are not moved immediately and individually towards the top left corner of the AED window. Instead, they are first reordered within an $n_b \times n_b$ computational window. Only after all eigenvalues in this $n_b \times n_b$ window are checked, the group of undeletable eigenvalues is moved simultaneously to the top left corner of the AED window. This blocked approach increases the computational intensity and avoids the frequent communication needed when reordering each eigenvalue individually. The procedure is repeated until all eigenvalues in the AED window are checked.

The last step is to eliminate the spike and reduce the AED window back to the upper Hessenberg form. This task is performed with the ScaLAPACK routine PDGEHRD. The corresponding off-diagonal blocks are updated by explicitly multiplying the accumulated orthogonal matrix using the PBLAS routine PDGEMM.

### 2.3.2 AED within the pipelined QR algorithm

Within the AED stage in the new parallel multishift QR algorithm, the pipelined QR algorithm is often used as the eigensolver since the AED window is relatively small compared to the whole matrix. However, the original
ScaLAPACK v1.8.0 implementation PDLAHQR of the pipelined QR algorithm is not equipped with the AED strategy and is hence not efficient. When adding AED, we have taken into account that we will only use this routine for small- to medium-size (sub)matrices. In particular, we can expect the AED window to be sufficiently small such that AED can be performed on one processor efficiently, by using the LAPACK implementation of AED.

Apart from AED, our new routine PDLAQR1 incorporates further modifications to PDLAHQR, making it both faster and more robust. The following list summarizes the most important aspects; additional details can be found in [30, 34].

- **Aggressive early deflation**: AED is implemented in an auxiliary routine PDLAQR2 which copies the AED window to local memory and calls the LAPACK routine DLAQR3 to solve the problem sequentially. To determine the parameters of AED, we use the settings of the LAPACK installation determined by ILAENV. However, a notable difference is that we do not use the undeflatable eigenvalues from the AED step as shifts in the subsequent pipelined QR sweep. Instead, we recompute the eigenvalues of the trailing submatrix to improve the quality of the shifts. We have observed that this shifting strategy accelerates the convergence of the pipelined QR algorithm.

- **Conventional deflation**: In PDLAHQR, pipelined QR sweeps are performed until the very end, that is, the remaining diagonal blocks are all of size 1 × 1 or 2 × 2. In PDLAQR1, we use a different strategy: Once the active block is sufficiently small (say, not larger than 385 × 385), we copy this block to local memory and call the LAPACK routines DLAQR3/DLAQR4 to solve the problem sequentially. This strategy significantly reduces communication overhead in the latter stages and is implemented in an auxiliary routine PDLAQR4.

- **Avoidance of anomalies**: The original ScaLAPACK routine PDLAHQR suffered from two anomalies, which have been removed. First, the routine sometimes returned 2 × 2 diagonal blocks containing real eigenvalues, which is not in accordance with the specification of the interface. In PDLAQR1, each 2 × 2 diagonal block contains a pair of complex conjugate eigenvalues. The second issue is concerned with a strategy already proposed by Francis [17] to potentially benefit from two consecutive small sub-diagonal entries. In the rare case when it is successful, this strategy allows to introduce bulges below the top left corner of the active submatrix. However, this turns out to be difficult to implement in a safe manner in pipelined or multishift QR sweeps. Indeed, when using the ScaLAPACK routine PDLACONSB which implements this strategy, we have observed large relative residuals of norm up to 10^{−5}, indicating numerical instabilities. As the performance improvements gained from this strategy are usually negligible, we have decided to remove it. In return, the numerical stability is improved.

### 2.4 Switching between Multishift QR and AED

In the LAPACK implementation of the QR algorithm, there are rules for balancing the cost between multishift QR sweeps and AED, see Step 4 in Algorithm 1. The precise meaning of this step is characterized by a threshold called NIBBLE. If we let \( n_{\text{undflt}} \) denote the number of undeflatable shifts in an AED step, the multishift QR sweep is skipped if

\[
\frac{n_{\text{AED}} - n_{\text{undflt}}}{n_{\text{AED}}} \geq \frac{\text{NIBBLE}}{100}.
\]

Since AED behaves differently for different matrices, this strategy automatically adjusts the choice between AED and QR sweeps based on the properties of the matrix.

The default value of NIBBLE in the sequential LAPACK implementation is 14, which provides a good balance between multishift QR sweeps and AED. The same default value is used in the pipelined QR algorithm. However, in the new parallel multishift QR algorithm, the parallel AED process becomes substantially more expensive than the sequential AED process due to communication. As explained above, the AED process only involves a smaller trailing submatrix, leading to decreased parallel efficiency. To account for this, NIBBLE should be set larger to avoid performing AED too frequently. A good choice of this threshold depends both on the size of the matrix \( H \) and the number of processors involved. We use the model NIBBLE = \( a \cdot n^b \cdot p^c \) for this purpose, where \( a, b, \) and \( c \) are...
are machine-dependent constants. An appropriate choice of these constants can be gained from repeated runs of the program with different thresholds. It turns out that the right choice of NIBBLE becomes rather sensitive when communication is slow. (In our numerical experiments, see Section 5, the default values on our computing architectures are chosen as \((a, b, c) = (335, -0.44, 0.5)\).)

A complication arises when using NIBBLE \(\geq 33\). Such a choice may lead to situations where the number of undeflatable eigenvalues is less than the desired number of shifts, that is \(n_{\text{undflt}} < n_{\text{shift}}\), due to the fact that \(n_{\text{undflt}} = 3n_{\text{shift}}/2\). The solution in the software is that as long as \(n_{\text{undflt}} \geq n_{\text{shift}}/2\), we only use these \(n_{\text{undflt}}\) undeflatable eigenvalues as shifts in the next QR sweep. However, the condition \(n_{\text{undflt}} \geq n_{\text{shift}}/2\) may also fail when NIBBLE \(\geq 67\). In this case we calculate the eigenvalues of the \(n_{\text{shift}} \times n_{\text{shift}}\) trailing principal submatrix of \(H\) and use them as shifts. The calculation can be performed by either PDLAQR0 or PDLAQR1, just like computing the Schur decomposition in the AED step.

### 2.5 Task duplication—efficient but hazardous

Task duplication is a common technique in parallel computing to reduce communication and potentially improve performance, see, e.g., [19]. This technique has already been employed in previous implementations of the parallel QR algorithm, such as PDLAHQR and software developed in [21, 30]. However, a crucial assumption is made when applying this technique: all involved processors need to produce identical outputs for identical tasks. Due to the effect of roundoff error, this assumption is not always satisfied, especially on heterogeneous architectures.

This lack of numerical reproducibility is potentially harmful to the robustness of the parallel QR algorithm. As discussed in Section 2.3.2, all computations within the AED window are performed sequentially for the pipelined QR algorithm. In [30], we have proposed to duplicate these sequential parts on all involved processors. The parallel update of the off-diagonal blocks can then be performed with the local copy of the orthogonal transformation matrix resulting from AED, without any extra communication. However, even the slightest change in finite-precision arithmetic may lead to very different outputs produced by AED. In particular, the ordering of the eigenvalues in the Schur decomposition computed within AED is very sensitive to such changes. In turn, the off-diagonal blocks are updated using completely different local copies of the orthogonal transformation matrices, leading to meaningless results. We have observed similar problems in crossborder bulge chasing and eigenvalue reordering. To avoid this, we use explicit communication rather than task duplication in the new implementation. For a moderate number of processors (e.g., \(p \leq 100\)), the change in performance is negligible; while for a large number of processors, the performance can drop. For example, for computing the Schur decomposition of a \(100,000 \times 100,000\) matrix using \(40 \times 40\) processors, up to 25% performance drop has been observed by replacing task duplication with explicit communication.

### 2.6 Avoiding communication via data redistribution

As observed in [30], the parallel QR algorithm is not efficient at solving relatively small eigenvalue problem on many processors, due to excessive communication, to the point that the execution time actually increases when increasing the number of processors. Such a situation is regularly encountered when calculating the Schur decomposition of the AED window. Therefore, an efficient parallel solver for this relatively small eigenvalue problem is of great interest. One possible approach to this problem is to use alternative algorithms, e.g., the spectral divide-and-conquer algorithm [7, 8]. In the following we propose a solution within the framework of the QR algorithm—a data redistribution strategy that reduces the communication overhead.

Recent work on communication avoiding algorithms [9, 8, 25, 28] usually focuses on the design of algorithms that can attain the theoretical lower bounds of the communication cost. A basic assumption in these theoretical analyses is that the data are nearly evenly distributed over the processors. Here we propose an alternative approach, which does not rely on this assumption and is especially useful for operations involving smaller submatrices.

We first consider a simple and extreme case. Suppose there is one processor which has a large amount of local memory and very high clock speed. Then by gathering all data to this processor, the problem can be solved
without further communication. Once the computation is completed, the data are scattered to their original owners. The total amount of communication does not exceed the cost of scattering and gathering regardless of the complexity of computational work. Although this simple idea does not work for large problems that cannot be stored on a single processor, it is still useful for smaller problems. For example, the AED process in the pipelined QR algorithm is implemented in such a manner since we know in advance that the AED window is always sufficiently small, such that the associated Schur decomposition can be efficiently solved sequentially. By introducing the overhead of data redistribution, the total amount of communication as well as the execution time are reduced.

For larger problems, it is not feasible to solve them sequentially via data redistribution. Specifically, the AED window within the new parallel multishift QR algorithm usually becomes quite large, although much smaller compared to the whole matrix. In this case, we choose a subset of processors instead of a single processor to perform AED. The data redistribution is performed using the routine PDGEMR2D in ScaLAPACK; its overhead has been observed to be negligible relative to the AED process as a whole.

The tunable parameter $p_{\text{min}} = \text{PILAENVX}(\text{ISPEC} = 23)$ determines our heuristic strategy for choosing the number of processors for the redistribution. If $\min(p_r, p_c) > p_{\text{min}} + 1$, we redistribute the AED window to a $p_{\text{min}} \times p_{\text{min}}$ processor grid and perform the calculations on this subset of processors. The same strategy is also applied if we need to compute shifts after an AED step. The default value for this parameter is $p_{\text{min}} = \lceil n_{\text{AED}}/(n_b[384/n_b]) \rceil$, implying that each processor needs to own at least 384 columns of the AED window. The constant 384 has been obtained via extensive numerical experiments on one of our target architectures. It certainly needs adjustment for optimal performance on other architectures.

2.7 Summary

Finally, we present pseudocode to summarize the discussion on our new parallel multishift QR algorithm. Algorithm 2 represents the parallel AED procedure used within the new multishift QR algorithm. Algorithm 3 provides pseudocode for the parallel multishift QR sweeps. For simplicity, the start-up and ending stages, i.e., bulge introduction and bulge annihilation, are not considered in the pseudocode. These algorithms are to be used within Algorithm 1 on distributed memory architectures.

3 Performance model

In this section, we briefly discuss the cost of computation and communication of the new parallel multishift QR algorithm for reducing a Hessenberg matrix to Schur form. The cost of accumulating orthogonal transformations are taken into account. The derivation of the results presented here will be given in the electronic appendix. For simplicity, we consider a square processor grid, that is, $p_r = p_c = \sqrt{p}$. In addition, we assume that each processor contains reasonably many data blocks of the matrices, i.e., $\sqrt{p} n_b \ll n$, so that the work load is balanced. The parallel execution time consists of two main components:

$$T_p = T_a + T_c,$$

where $T_a$ and $T_c$ are the times for arithmetic operations and communication, respectively. The possibility of overlapping communication with computations is not taken into account. By neglecting the communication between memory and cache lines inside one core, the serial runtime can be approximated by

$$T_a = \frac{\text{#(flops)}}{f(p)} \gamma,$$

where $\gamma$ is the average time for performing one floating point operation and $f(p)$ is the degree of concurrency. For the communication between two processors, we define $\alpha$ and $\beta$ as the start-up time (or communication latency) and the time for transferring one word without latency (or reciprocal of bandwidth), respectively. The time for a single point-to-point communication is modelled as $\alpha + L \beta$ where $L$ is the message size in words. A one-to-all broadcast or an all-to-one reduction within a scope of $p$ processors is assumed to take $\Theta(\log p)$ steps.
Algorithm 2 Parallel aggressive early deflation within the new multishift QR algorithm

**Input:** \( H \in \mathbb{R}^{n \times n} \) is upper Hessenberg, with the AED window contained in \( H(i : j, i : j) \); \( Q \in \mathbb{R}^{n \times n} \) is orthogonal.

**Output:** Updated Hessenberg matrix \( H \leftarrow U^T H U \) obtained after performing AED, \( Q \leftarrow QU \), with \( U \) orthogonal.

1. Estimate the optimal process grid size, \( p_{\text{AED}} = p_{\text{min}}^2 \), based on \( n_{\text{AED}} = j - i + 1 \) and \( n_b \).
2. if \( \min(p_r, p_c) \leq p_{\text{min}} + 1 \) then
   3. \( \hat{p}_r \leftarrow p_{\text{min}}, \hat{p}_c \leftarrow p_{\text{min}}. \)
4. else
   5. \( \hat{p}_r \leftarrow p_r, \hat{p}_c \leftarrow p_c. \)
6. end if
7. Distribute \( H_0 \leftarrow H(i : j, i : j) \) to a \( \hat{p}_r \times \hat{p}_c \) process subgrid if necessary.
8. Compute the Schur decomposition \( H_0 = V_0 H_0 V_0^T \) on the \( \hat{p}_r \times \hat{p}_c \) process subgrid.
9. Distribute \( T \leftarrow T_0, V \leftarrow V_0 \) back to the original \( p_r \times p_c \) process grid.
10. \( s \leftarrow H(i, i - 1) V(1:)^T. \)
11. repeat
12.   Check deflation for the bottommost \( n_b \) unchecked eigenvalues; undeflatable eigenvalues are moved to the top-left corner of this \( n_b \times n_b \) block.
13.   Move all undeflatable eigenvalues within this group of \( n_b \) eigenvalues to the top-left corner of the AED window.
14.   Update \( s \leftarrow V_1^T s, T \leftarrow V_1^T T V_1 \) in parallel (on the original \( p_r \times p_c \) process grid), where \( V_1 \) is the accumulated orthogonal matrix from Steps 12 and 13.
15. until all eigenvalues of \( T \) are tested
16. Eliminate the spike: \( V_2^T s = \eta e_1 \); Update \( T \leftarrow V_2^T T V_2 \) in parallel.
17. Reduce \( T \) to an upper Hessenberg matrix \( H_1 \leftarrow V_2^T T V_3 \) in parallel.
18. Set \( H(i : i - 1) \leftarrow \eta, H(i : j, i : j) \leftarrow T \); Update \( V \leftarrow V V_3 \) in parallel.
19. Update \( H(i : j, j + 1 : n) \leftarrow V^T H(i : j, j + 1 : n) \) in parallel.
20. Update \( H(1 : i - 1, i : j) \leftarrow H(1 : i - 1, i : j) V \) in parallel.
21. Update \( Q(1 : n, i : j) \leftarrow Q(1 : n, i : j) V \) in parallel.

Let \( k_{\text{AED}} \) and \( k_{\text{Sweep}} \) denote the number of AED steps and QR sweeps, respectively, performed by the new parallel multishift QR algorithm. We have \( k_{\text{Sweep}} \leq k_{\text{AED}} \), since some QR sweeps are skipped when the percentage of deflated eigenvalues in the AED step is larger than the threshold (NIBBLE). When the number of undeflatable eigenvalues from AED is not sufficient for performing the next QR sweep, we need to calculate shifts from the trailing submatrix. The number of extra calls to the parallel Schur decomposition solver in this case is denoted by \( k_{\text{Shift}} \), which of course satisfies \( k_{\text{Shift}} \leq k_{\text{Sweep}} \). Given the constants \( k_{\text{AED}}, k_{\text{Sweep}}, \) and \( k_{\text{Shift}} \), the execution time of the new multishift QR algorithm is modelled as the sum of the corresponding phases:

\[
T_{\text{new}}(n, p) = k_{\text{AED}} T_{\text{AED}}(n, n_{\text{AED}}, p) + k_{\text{Sweep}} T_{\text{Sweep}}(n, n_{\text{Shift}}, p) + k_{\text{Shift}} T_{\text{Shift}}(n, n_{\text{Shift}}, p),
\]

where \( T_{\text{AED}}, T_{\text{Sweep}}, \) and \( T_{\text{Shift}} \) are the runtimes for performing each phase once. For simplicity, it is assumed that \( n_{\text{AED}} \) and \( n_{\text{Shift}} \) remain constant throughout the entire QR algorithm, and all QR sweeps act on the entire matrix. We further assume that AED is always performed on a \( \sqrt{p_{\text{AED}}} \times \sqrt{p_{\text{AED}}} \) processor grid, so that the property \( \sqrt{p_{\text{AED}}}, n_b \leq n_{\text{AED}} \) is also valid inside the AED window. The same assumption is made for the shift calculation phase.

Typically, we have

\[
n_{\text{Shift}} \approx \frac{2}{3} n_{\text{AED}} \approx \frac{1}{C_1} n \quad \text{and} \quad \frac{n_{\text{AED}}}{\sqrt{p_{\text{AED}}}} \approx \frac{n_{\text{Shift}}}{\sqrt{p_{\text{Shift}}}} \geq C_2,
\]

where \( C_1 \) and \( C_2 \) are constants (e.g., \( C_1 = 24, C_2 = 384 \)). In practice, \( k_{\text{AED}}, k_{\text{Sweep}}, k_{\text{Shift}} \) can vary a lot for different matrices. We assume \( k_{\text{AED}} = \Theta(n / n_{\text{AED}}) = \Theta(C_1) \), which appears to be reasonable.

In the following we present performance models based on the assumptions above. Tiny terms, especially lower order terms with reasonably sized constants, are omitted. The derivation of these models can be found in
Algorithm 3 Parallel multishift QR sweep (bulge chasing process)

Input: $H \in \mathbb{R}^{n \times n}$ is upper Hessenberg except for several tightly coupled bulge chains in the top left corner; $Q \in \mathbb{R}^{n \times n}$ is orthogonal.

Output: Updated matrix $H \leftarrow U^T H U$ has the tightly coupled bulge chains moved to the bottom right corner, $Q \leftarrow QU$, where $U$ is orthogonal and the new $H$ is upper Hessenberg.

1: for each window $w = (i : i + n_b - 1)$ in parallel do
2: if (myrow, mycol) owns parts of $(\cdot, w)$ or $(w, \cdot)$ then
3: if (myrow, mycol) owns $(w, w)$ then
4: Chase the bulge chain inside $w$ down $[n_b/2]$ rows.
5: Broadcast the local orthogonal matrix $V$ in process row myrow.
6: Broadcast the local orthogonal matrix $V$ in process column mycol.
7: else
8: Receive $V$.
9: end if
10: Update $H(w, i + n_b : n) \leftarrow V^T H(w, i + n_b : n)$ in parallel.
11: Update $H(1 : i - 1, w) \leftarrow H(1 : i - 1, w)V$ in parallel.
12: Update $Q(1 : n, w) \leftarrow Q(1 : n, w)V$ in parallel.
13: end if
14: end for
15: for each odd-numbered window $w = (j : j + n_b - 1)$ in parallel do
16: if (myrow, mycol) owns parts of $(\cdot, w)$ or $(w, \cdot)$ then
17: if (myrow, mycol) owns parts of $(w, w)$ then
18: Form a process subgrid $G_w = \{(0, 0)_w, (0, 1)_w, (1, 0)_w, (1, 1)_w\}$. 
19: Exchange data in $G_w$ to build $H(w, w)$ at $(0, 0)_w$.
20: if (myrow, mycol) = $(0, 0)_w$ then
21: Chase the bulge chain inside $w$ down $[n_b/2]$ rows.
22: Send $H(w, w)$ and the local orthogonal matrix $V$ to other processes in $G_w$.
23: else
24: Receive $H(w, w)$ and $V$ from $(0, 0)_w$.
25: end if
26: if (myrow, mycol) = $(0, 0)_w$ or (myrow, mycol) = $(1, 1)_w$ then
27: Broadcast $V$ in process row myrow.
28: Broadcast $V$ in process column mycol.
29: end if
30: else
31: Receive $V$.
32: end if
33: Exchange local parts of $H(w, j + n_b : n), H(1 : j - 1, w)$, and $Q(1 : n, w)$ with neighboring processes in parallel.
34: Update $H(w, j + n_b : n) \leftarrow V^T H(w, j + n_b : n)$ in parallel.
35: Update $H(1 : j - 1, w) \leftarrow H(1 : j - 1, w)V$ in parallel.
36: Update $Q(1 : n, w) \leftarrow Q(1 : n, w)V$ in parallel.
37: end if
38: end for
39: for each even-numbered window $w = (j : j + n_b - 1)$ in parallel do
40: % Analogous procedure as described for the odd case above.
41: end for

the electronic appendix. By [24], the execution time of the pipelined QR algorithm is

$$T_{pipe}(n, p) = \Theta\left(\frac{n^3}{p}\right)^\gamma + \Theta\left(\frac{n^2 \log p}{\sqrt{p} n_b}\right)^\alpha + \Theta\left(\frac{n^3}{p n_b}\right)^\beta,$$

(2)
provided that the average number of shifts required for deflating each eigenvalue is Θ(1). One QR sweep in our new parallel multishift QR algorithm roughly requires

\[
T_{\text{sweep}}(n, n_{\text{shift}}, p) \approx \frac{36n^2 n_{\text{shift}}}{p} \gamma + \frac{9n n_{\text{shift}}}{\sqrt{p} n_b^2} (\log_2 p + 4) \alpha + \frac{9n^2 n_{\text{shift}}}{pn_b} \beta
\]

evaluation time. Therefore, under the same assumption of convergence rate (i.e., \(k_{\text{sweep}} n_{\text{shift}} = \Theta(n)\)), it can be shown that the execution time of the new parallel multishift QR algorithm without AED is

\[
T_{\text{new}}(n, p) = k_{\text{sweep}} T_{\text{sweep}}(n, n_{\text{shift}}, p) = \Theta\left(\frac{n^3}{p}\right) \gamma + \Theta\left(\frac{n^2 \log p}{\sqrt{p} n_b^2}\right) \alpha + \Theta\left(\frac{n^3}{pn_b}\right) \beta.
\]

It is interesting to make a comparison between (2) and (3). Both solvers have an ideal degree of concurrency. However, the tightly coupled shift strategy is superior to loosely coupled shifts, because it yields less frequent communication. The number of messages is reduced by a factor of \(\Theta(n_b)\); in return the average message length increases correspondingly. Another important observation is that the serial term in \(T_{\text{pipe}}\) assumes level 3 performance, which is actually not the case. This already explains why the pipelined QR algorithm is usually much slower than the new parallel multishift QR algorithm for larger matrices, even when neglecting the effects of AED.

Taking AED into account makes the model significantly more complicated. The execution times for AED and computing shifts are estimated by

\[
T_{\text{AED}}(n, n_{\text{AED}}, p) \approx \left[ \frac{30C_2^2 n}{C_1} + \frac{9n n_{\text{AED}}}{C_1 \sqrt{p}} + \frac{9(C_1 + 6)n^3}{2C_1^2 p} \right] \gamma
\]

\[
+ \left[ \frac{9C_2^2 n}{C_1 n_b} \log_2 \frac{3n}{2C_1 C_2} + \frac{3n}{2C_1} \log_2 p \right] \alpha
\]

\[
+ \left[ \frac{9C_2^2 n}{C_1 n_b} \log_2 \frac{3n}{2C_1 C_2} + \frac{12C_2^2 n}{C_1 n_b} + \frac{3n^2 (18 + C_1 + 51 \log_2 p)}{16C_1^2 \sqrt{p}} \right] \beta,
\]

and

\[
T_{\text{shift}}(n, n_{\text{shift}}, p) \approx \frac{10C_2^2 n}{C_1} \gamma + \frac{6C_2 n}{C_1 n_b} \log_2 \frac{n}{C_1 C_2} \alpha + \left( \frac{6C_2 n}{C_1} \log_2 \frac{n}{C_1 C_2} + \frac{8C_2^2 n}{C_1 n_b} \right) \beta.
\]

respectively. To be able to provide some intuition, we assign concrete values to most parameters. For example, let us set \(C_1 = 24\), \(C_2 = 384\), and assume \(k_{\text{AED}} = 2k_{\text{sweep}} = 16k_{\text{shift}} = 64\). Then

\[
T_{\text{sweep}}(n, n_{\text{shift}}, p) \approx \frac{3n^3}{2p} \gamma + \frac{3n^2}{8 \sqrt{p} n_b^2} (\log_2 p + 4) \alpha + \frac{3n^3}{8pn_b} \beta,
\]

\[
T_{\text{AED}}(n, n_{\text{AED}}, p) \approx \left[ 184320n + \frac{n^2 n_b}{64 \sqrt{p}} + \frac{5n^3}{256p} \right] \gamma
\]

\[
+ \left[ \frac{144n}{n_b} (\log_2 n - 14) + \frac{n \log_2 p}{8} \right] \alpha
\]

\[
+ \left[ \frac{144n}{n_b} (\log_2 n - 14 + \frac{512}{n_b}) + \frac{(51 \log_2 p + 42)n^2}{3072 \sqrt{p}} \right] \beta,
\]

\[
T_{\text{shift}}(n, n_{\text{shift}}, p) \approx 61440n \gamma + \frac{96n}{n_b} (\log_2 n - 13) \alpha + 96n (\log_2 n - 13 + \frac{512}{n_b}) \beta.
\]

This yields the following overall estimate for the new parallel QR algorithm with AED:

\[
T_{\text{new}}(n, p) \approx \Theta\left(\frac{n^3}{p}\right) \gamma + \Theta\left(\frac{n^2 \log p}{\sqrt{p} n_b^2}\right) \alpha + \Theta\left(\frac{n^3}{pn_b}\right) \beta,
\]

\[
= \Theta\left(\frac{n^3}{p}\right) \gamma + \Theta\left(\frac{n^2 \log p}{\sqrt{p} n_b^2}\right) \alpha + \Theta\left(\frac{n^3}{pn_b}\right) \beta.
\]
where most small-order terms are neglected. It turns out that both QR sweeps and AED have significant serial runtime when $n$ is not very large. However, QR sweeps usually dominate the communication cost. As a consequence, the models (2), (3), and (6) nearly have the same asymptotic behavior. AED is asymptotically not more expensive compared to QR sweeps, and hence it does not represent a computational bottleneck for larger matrices. Combined with the convergence acceleration often observed when using AED (and not fully attributed in the model above), this contributes to the superior performance of the new parallel multishift QR algorithm.

4 Other Implementation Issues

4.1 Calling sequence

The calling sequence of the newly developed routine PDHSEQR is nearly identical with the LAPACK routine DHSEQR, see Figure 5. Apart from the need of a descriptor for each globally distributed matrix and the leading dimension for each local matrix, the only difference is that PDHSEQR requires an extra integer workspace.

The calling sequence of the ScaLAPACK routine PDLAHQR is also similar, hopefully allowing to easily switch from PDLAHQR and DHSEQR in existing software making use of ScaLAPACK. In practice, it is advisable to call PDHSEQR twice—one call for the workspace query (by setting LWORK=-1) and another call for actually doing the computation. This follows the convention of many LAPACK/ScaLAPACK routines that make use of workspace.

4.2 Tuning parameters

In the new software for the parallel multishift QR algorithm, tunable parameters are defined in the routine PILAENLV. They are available via the function call PILAENLV(ICTXT, ISPEC, ...) with $12 \leq ISPEC \leq 23$. A complete list of these parameters is provided in Table 2. Some of them require fine tuning to attain nearly optimal performance across different architectures.

Although a reasonable choice of $n_b$, the data layout block size, is important, we have observed the performance to be not overly sensitive to this choice. On the one hand, $n_b$ should be large enough so that the local computations can achieve level 3 performance. On the other hand, it is advisable to avoid $n_b$ being too large. A large value of $n_b$ harms load balance and increases the overhead in the start-up and ending stages of the bulge chasing process, especially when computing the Schur decomposition of the AED window. In our performance
Table 2: List of tunable parameters

<table>
<thead>
<tr>
<th>ISPEC</th>
<th>Name</th>
<th>Description</th>
<th>Recommended value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>n_{min}</td>
<td>Crossover point between PDLAQR0 and PDLAQR1</td>
<td>220 \min(p_r, p_c)</td>
</tr>
<tr>
<td>13</td>
<td>n_{AED}</td>
<td>Size of the AED window</td>
<td>See Table 1</td>
</tr>
<tr>
<td>14</td>
<td>NIBBLE</td>
<td>Threshold for skipping a multishift QR sweep</td>
<td>See Section 2.4</td>
</tr>
<tr>
<td>15</td>
<td>n_{shift}</td>
<td>Number of simultaneous shifts</td>
<td>See Table 1</td>
</tr>
<tr>
<td>16</td>
<td>KACC22</td>
<td>Specification of how to update off-diagonal blocks in the multishift QR sweep</td>
<td>Use GEMM/TRMM</td>
</tr>
<tr>
<td>17</td>
<td>NUMWIN</td>
<td>Maximum number of concurrent computational windows (for both QR sweep and eigenvalue reordering)</td>
<td>\min(p_r, p_c, \lceil n/n_b \rceil)</td>
</tr>
<tr>
<td>18</td>
<td>WINEIG</td>
<td>Number of eigenvalues in each window (for eigenvalue reordering)</td>
<td>\min(n_b/2, 40)</td>
</tr>
<tr>
<td>19</td>
<td>WINSIZE</td>
<td>Computational window size (for both bulge-chasing and eigenvalue reordering)</td>
<td>\min(n_b, 80)</td>
</tr>
<tr>
<td>20</td>
<td>MMULT</td>
<td>Minimal percentage of flops for performing GEMM instead of pipelined Householder reflections when updating the off-diagonal blocks in the eigenvalue reordering routine</td>
<td>50</td>
</tr>
<tr>
<td>21</td>
<td>NCB</td>
<td>Width of block column slabs for rowwise update of Householder reflections in factorized form</td>
<td>\min(n_b, 32)</td>
</tr>
<tr>
<td>22</td>
<td>WNEICR</td>
<td>Maximum number of eigenvalues to move over a block border in the eigenvalue reordering routine</td>
<td>Identical to WINEIG</td>
</tr>
<tr>
<td>23</td>
<td>p_{min}</td>
<td>Size of processor grid involving AED</td>
<td>See Section 2.6</td>
</tr>
</tbody>
</table>

We refer to the model, we always assume n_b \ll n/\sqrt{p} to avoid such kind of overhead. For many architectures, n_b \in [32, 128] will offer a good choice.

We expect that most of the recommended values in Table 2 yield reasonable performance on existing architectures. However, the parameters n_{min}, p_{min} and NIBBLE require some extra care, as the performance of AED crucially relies on them. The values of these parameters need to be determined by performing a bunch of test runs: To determine n_{min} and p_{min}, it is advisable to use the typical sizes of the AED windows (see Table 1) and run tests on different processor grids. Then the optimal values for both n_{min} and p_{min} can be chosen via examining the number of columns of H owned by each processor. NIBBLE should be tuned lastly, once all other parameters are fixed. Auto-tuning tools have been provided in the software for tuning these three parameters. We refer to the User’s Guide [22] for details of the tuning procedure. Tuning NIBBLE is time-consuming but highly recommended, especially on older architectures with relatively slow communication. As discussed in Section 2.4, NIBBLE = a \cdot n^b p^c is a reasonably good model that takes into account both n and p. We have observed reasonable performance behavior of this model for large problems up to size n = 100,000 and p = 40 \times 40.

5 Computational Experiments

We have performed a large set of computational experiments on Akka [27] and Abisko [26] hosted by the High Performance Computing Center North (HPC2N), and on Bellatrix [15] hosted by École Polytechnique Fédérale de Lausanne. In this section, we present a subset from these computational experiments to confirm and demonstrate the improvements we have made in the parallel QR algorithm. The computational environments are summarized in Table 3.

We compare the following implementations:
Table 3: Computational environments.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Description</th>
</tr>
</thead>
</table>
| Akka        | 64-bit Intel Xeon (Harpertown) Linux cluster  
|             | 672 dual socket nodes with L5420 quad-core 2.5GHz processors and 16GB RAM per node  
|             | Cisco Infiniband and Gigabit Ethernet, 10Gbps bandwidth  
|             | PathScale compiler version 4.0.13  
|             | OpenMPI 1.4.4, LAPACK 3.4.0, GotoBLAS2 1.13 |
| Abisko      | 64-bit AMD Opteron (Interlagos) Linux cluster  
|             | 322 nodes with four Opteron 6238 12-core 2.6GHz processors and 128GB RAM per node  
|             | Mellanox 4X QSFP Infiniband connectivity, 40Gbps bandwidth  
|             | PathScale compiler version 4.0.13  
|             | OpenMPI 1.6.4, LAPACK 3.4.0, OpenBLAS 0.1 alpha 2.4 |
| Bellatrix   | 64-bit Intel Xeon (Sandy Bridge) Linux cluster  
|             | 424 dual socket nodes with E5-2660 octa-core 2.2GHz processors and 32GB RAM per node  
|             | Qlogic Infiniband QDR 2:1 connectivity, 40Gbps bandwidth  
|             | Intel compiler version 13.0.1  
|             | Intel MPI version 4.1.0, Intel Math Kernel Library version 11.0 |

The implementation NEW improves upon the implementation SISC in terms of robustness and performance, in particular because of the proposed modifications to the AED step.

The data layout block size $n_b = 50$ is used for all experiments. No multithreaded features (such as OpenMP or threaded BLAS) are used. Therefore the number of processors ($p = p_r \cdot p_c$) means the number of cores involved in the computation.

5.1 Random matrices

First, we consider two types of random matrices—fullrand and hessrand [21].

Matrices of the type fullrand are dense square matrices with all entries randomly generated from a uniform distribution in $[0, 1]$. We call the ScaLAPACK routine PDGEHRD to reduce them to upper Hessenberg form before applying the QR algorithm. Only the time for the QR algorithm (i.e., reducing the upper Hessenberg matrix to Schur form) is measured. These matrices usually have well-conditioned eigenvalues and exhibit “regular” convergence behavior.

Matrices of the type hessrand are upper Hessenberg matrices whose nonzero entries are randomly generated from a uniform distribution in $[0, 1]$. The eigenvalues of these matrices are extremely ill-conditioned for larger $n$, affecting the convergence behavior of the QR sweeps [21]. On the other hand, AED often deflates a high fraction of eigenvalues in the AED window for such matrices. These properties sometimes cause erratic convergence rates.

Tables 4 and 5 show the parallel execution times of the three solvers on Akka. Both the real Schur form $T$ and the orthogonal transformation matrix $Z$ are calculated. We limit the total execution time (including the Hessenberg reduction) by 10 hours for each individual problem, to avoid excessive use of the computational resources. An entry $\infty$ corresponds to an execution time larger than 10 hours. These tables reveal that our new version of PDHSEQR (i.e., NEW) always improves the performance compared to the SISC version. On average, the improvement is 31% for matrices of type fullrand and 14 times for matrices of type hessrand. Not surprisingly, the improvements compared to PDLAHQR in ScaLAPACK v1.8.0 are even more significant.

The convergence rates for fullrand are sufficiently regular, so that we can analyze the scalability of the
Table 4: Execution time (in seconds) on Akka for fullrand matrices.

<table>
<thead>
<tr>
<th>p = p_r \times p_c</th>
<th>n = 4000</th>
<th>n = 8000</th>
<th>n = 16000</th>
<th>n = 32000</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-v180</td>
<td>SISC</td>
<td>NEW</td>
<td>S-v180</td>
<td>SISC</td>
</tr>
<tr>
<td>1 \times 1</td>
<td>834</td>
<td>178</td>
<td>115</td>
<td>10730</td>
</tr>
<tr>
<td>2 \times 2</td>
<td>317</td>
<td>87</td>
<td>56</td>
<td>2780</td>
</tr>
<tr>
<td>4 \times 4</td>
<td>136</td>
<td>50</td>
<td>35</td>
<td>764</td>
</tr>
<tr>
<td>6 \times 6</td>
<td>112</td>
<td>50</td>
<td>43</td>
<td>576</td>
</tr>
<tr>
<td>8 \times 8</td>
<td>100</td>
<td>45</td>
<td>37</td>
<td>464</td>
</tr>
<tr>
<td>10 \times 10</td>
<td>97</td>
<td>50</td>
<td>36</td>
<td>417</td>
</tr>
</tbody>
</table>

Table 5: Execution time (in seconds) on Akka for hessrand matrices.

<table>
<thead>
<tr>
<th>p = p_r \times p_c</th>
<th>n = 4000</th>
<th>n = 8000</th>
<th>n = 16000</th>
<th>n = 32000</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-v180</td>
<td>SISC</td>
<td>NEW</td>
<td>S-v180</td>
<td>SISC</td>
</tr>
<tr>
<td>1 \times 1</td>
<td>685</td>
<td>317</td>
<td>14</td>
<td>6981</td>
</tr>
<tr>
<td>2 \times 2</td>
<td>322</td>
<td>200</td>
<td>8</td>
<td>2464</td>
</tr>
<tr>
<td>4 \times 4</td>
<td>163</td>
<td>112</td>
<td>36</td>
<td>1066</td>
</tr>
<tr>
<td>6 \times 6</td>
<td>137</td>
<td>84</td>
<td>31</td>
<td>768</td>
</tr>
<tr>
<td>8 \times 8</td>
<td>121</td>
<td>68</td>
<td>25</td>
<td>634</td>
</tr>
<tr>
<td>10 \times 10</td>
<td>131</td>
<td>83</td>
<td>23</td>
<td>559</td>
</tr>
</tbody>
</table>

parallel multishift QR algorithm. If we fix the memory load per core to n/\sqrt{p} = 4000, the execution times in Table 4 satisfy

\[ 2T(n, p) \leq T(2n, 4p) < 4T(n, p), \]

indicating that the parallel multishift QR algorithm scales reasonably well but not perfectly. To verify the performance models we have derived in Section 3, we use (4) together with the measured values of \( k_{\text{AED}}, k_{\text{sweep}}, \) and \( k_{\text{shift}} \) to predict the execution time. Since \( k_{\text{shift}} = 0 \) is observed for all these examples, there are only two components in the total execution time (i.e., \( T = k_{\text{sweep}}T_{\text{sweep}} + k_{\text{AED}}T_{\text{AED}} \)). Figure 6(a) illustrates that the predicted execution times underestimate the measured ones (especially, for AED), mainly due to too optimistic choices of the parameters \( (\alpha, \beta, \gamma) \). If we assume that the program executes at 40% and 7.5% of the peak core performance for level 3 and level 1–2 BLAS operations, respectively, the calibrated model fits the actual execution time quite well for large matrices (see Figure 6(b)). Since the model(s) are asymptotic, the results are very satisfactory.

For hessrand, it is observed that most eigenvalues are deflated with very few (or even no) QR sweeps. Considering that the main difference of PDHESEQR between versions NEW and SISC is in the AED process, it is not surprising to see the great convergence acceleration for hessrand, where AED dominates the calculation. In Table 5, sometimes the execution time for the new parallel multishift QR algorithm does not change too much when increasing the number of processors. This is mainly because the Schur decomposition of the AED window, which is the most expensive part of the algorithm, is performed by a constant number of processors \( \left( p_{\min} \cdot p_{\min} \leq p \right) \) after data redistribution.

In Tables 6–9 we list the execution times received from Abisko and Bellatrix. The observations are similar to those obtained from Akka. Therefore, in the rest of this section we only present experiments on Akka for economical consideration.

5.2 100,000 × 100,000 matrices

The modifications proposed and presented result into dramatic improvements for settings with very large matrices and many processors. To demonstrate this, we present the obtained execution times for 100,000 × 100,000 matrices in Table 10. Although the QR algorithm does not scale as well as Hessenberg reduction when fixing the problem size and increasing the number of processors, the execution times of these two reduction steps are still on the same order of magnitude. With the help of the dynamic NIBBLE strategy, the fraction of the execution time spent on AED for fullrand matrices is under control. In contrast to our earlier implementation, AED is not a
Figure 6: Comparison between the measured execution times and the predicted times using (4) (fullrand, \(n/\sqrt{p} = 4000\)). The original model (left) uses theoretical values of \((\alpha, \beta, \gamma)\) according to the hardware information, while the calibrated one (right) adjusts \(\gamma\) according to different computational intensities (level 1, 2, and 3).

Table 6: Execution time (in seconds) on Abisko for fullrand matrices.

<table>
<thead>
<tr>
<th>(p = p_r \times p_c)</th>
<th>(n = 4000)</th>
<th>(n = 8000)</th>
<th>(n = 16000)</th>
<th>(n = 32000)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S-v180 SISC NEW</td>
<td>S-v180 SISC NEW</td>
<td>S-v180 SISC NEW</td>
<td>S-v180 SISC NEW</td>
</tr>
<tr>
<td>1 \times 1</td>
<td>764 139 97</td>
<td>7373 694 471</td>
<td>5507 1040 548</td>
<td>(\infty) 2641 1706</td>
</tr>
<tr>
<td>2 \times 2</td>
<td>302 77 49</td>
<td>2479 417 240</td>
<td>2799 591 374</td>
<td>(\infty) 2506 1245</td>
</tr>
<tr>
<td>4 \times 4</td>
<td>91 40 31</td>
<td>781 156 132</td>
<td>1881 383 294</td>
<td>(\infty) 1909 1118</td>
</tr>
<tr>
<td>6 \times 6</td>
<td>76 36 28</td>
<td>541 101 91</td>
<td>2799 591 374</td>
<td>(\infty) 2641 1706</td>
</tr>
<tr>
<td>8 \times 8</td>
<td>52 34 29</td>
<td>276 88 98</td>
<td>1881 383 294</td>
<td>(\infty) 2506 1245</td>
</tr>
<tr>
<td>10 \times 10</td>
<td>52 30 18</td>
<td>234 99 92</td>
<td>1455 317 257</td>
<td>(\infty) 1909 1118</td>
</tr>
</tbody>
</table>

Table 7: Execution time (in seconds) on Abisko for hessrand matrices.

<table>
<thead>
<tr>
<th>(p = p_r \times p_c)</th>
<th>(n = 4000)</th>
<th>(n = 8000)</th>
<th>(n = 16000)</th>
<th>(n = 32000)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S-v180 SISC NEW</td>
<td>S-v180 SISC NEW</td>
<td>S-v180 SISC NEW</td>
<td>S-v180 SISC NEW</td>
</tr>
<tr>
<td>1 \times 1</td>
<td>611 307 12</td>
<td>5021 2064 63</td>
<td>6671 1603 53</td>
<td>(\infty) 231 176</td>
</tr>
<tr>
<td>2 \times 2</td>
<td>302 202 7</td>
<td>1966 1458 29</td>
<td>4006 1034 52</td>
<td>(\infty) 231 176</td>
</tr>
<tr>
<td>4 \times 4</td>
<td>110 77 18</td>
<td>881 516 48</td>
<td>6671 1603 53</td>
<td>(\infty) 231 176</td>
</tr>
<tr>
<td>6 \times 6</td>
<td>96 61 21</td>
<td>578 339 70</td>
<td>6671 1603 53</td>
<td>(\infty) 231 176</td>
</tr>
<tr>
<td>8 \times 8</td>
<td>84 53 18</td>
<td>423 249 98</td>
<td>2822 605 53</td>
<td>(\infty) 737 166</td>
</tr>
<tr>
<td>10 \times 10</td>
<td>73 58 17</td>
<td>360 214 79</td>
<td>2456 553 56</td>
<td>(\infty) 670 166</td>
</tr>
</tbody>
</table>

Table 8: Execution time (in seconds) on Bellatrix for fullrand matrices.

<table>
<thead>
<tr>
<th>(p = p_r \times p_c)</th>
<th>(n = 4000)</th>
<th>(n = 8000)</th>
<th>(n = 16000)</th>
<th>(n = 32000)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S-v180 SISC NEW</td>
<td>S-v180 SISC NEW</td>
<td>S-v180 SISC NEW</td>
<td>S-v180 SISC NEW</td>
</tr>
<tr>
<td>1 \times 1</td>
<td>637 73 50</td>
<td>5377 441 252</td>
<td>4505 552 271</td>
<td>(\infty) 1267 901</td>
</tr>
<tr>
<td>2 \times 2</td>
<td>192 24 21</td>
<td>1594 137 91</td>
<td>1886 247 165</td>
<td>(\infty) 1267 901</td>
</tr>
<tr>
<td>4 \times 4</td>
<td>68 16 13</td>
<td>498 79 63</td>
<td>1347 184 129</td>
<td>(\infty) 1362 714</td>
</tr>
<tr>
<td>6 \times 6</td>
<td>47 12 11</td>
<td>294 44 39</td>
<td>1347 184 129</td>
<td>(\infty) 1362 714</td>
</tr>
<tr>
<td>8 \times 8</td>
<td>36 16 12</td>
<td>204 42 37</td>
<td>1347 184 129</td>
<td>(\infty) 1362 714</td>
</tr>
<tr>
<td>10 \times 10</td>
<td>37 14 9</td>
<td>181 39 40</td>
<td>961 140 110</td>
<td>(\infty) 726 525</td>
</tr>
</tbody>
</table>
Table 9: Execution time (in seconds) on *Bellatrix* for *hessrand* matrices.

<table>
<thead>
<tr>
<th>p = p_r × p_c</th>
<th>n = 4000</th>
<th>n = 8000</th>
<th>n = 16000</th>
<th>n = 32000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S-v180</td>
<td>SISC</td>
<td>NEW</td>
<td>S-v180</td>
</tr>
<tr>
<td>1 × 1</td>
<td>510</td>
<td>174</td>
<td>6</td>
<td>4353</td>
</tr>
<tr>
<td>2 × 2</td>
<td>205</td>
<td>66</td>
<td>4</td>
<td>1590</td>
</tr>
<tr>
<td>4 × 4</td>
<td>87</td>
<td>42</td>
<td>7</td>
<td>657</td>
</tr>
<tr>
<td>6 × 6</td>
<td>57</td>
<td>23</td>
<td>9</td>
<td>428</td>
</tr>
<tr>
<td>8 × 8</td>
<td>54</td>
<td>37</td>
<td>9</td>
<td>340</td>
</tr>
<tr>
<td>10 × 10</td>
<td>46</td>
<td>22</td>
<td>8</td>
<td>280</td>
</tr>
</tbody>
</table>

Table 10: Execution time (in seconds) on *Akka* of the new parallel multishift QR algorithm (NEW) for 100,000 × 100,000 matrices.

<table>
<thead>
<tr>
<th></th>
<th>p = 16 × 16</th>
<th>p = 24 × 24</th>
<th>p = 32 × 32</th>
<th>p = 40 × 40</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fullrand</td>
<td>hessrand</td>
<td>fullrand</td>
<td>hessrand</td>
</tr>
<tr>
<td>Balancing</td>
<td>876</td>
<td>–</td>
<td>881</td>
<td>–</td>
</tr>
<tr>
<td>Hess. reduction</td>
<td>10084</td>
<td>–</td>
<td>6441</td>
<td>–</td>
</tr>
<tr>
<td>QR algorithm</td>
<td>13797</td>
<td>922</td>
<td>8055</td>
<td>1268</td>
</tr>
<tr>
<td><em>k_sweep</em></td>
<td>35</td>
<td>19</td>
<td>31</td>
<td>19</td>
</tr>
<tr>
<td>#(shifts)/n</td>
<td>0.20</td>
<td>0.23</td>
<td>0.35</td>
<td>0.49</td>
</tr>
<tr>
<td>AED% in the QR alg.</td>
<td>48%</td>
<td>100%</td>
<td>43%</td>
<td>100%</td>
</tr>
</tbody>
</table>

bottleneck of the whole QR algorithm now. As reported in [21], it took 7 hours for our earlier implementation PDHSEQR to solve the 100,000 × 100,000 *fullrand* problem with 32 × 32 processors; 80% execution time of the QR algorithm was spent on AED. Our new version of PDHSEQR is able to solve the same problem in roughly 1.85 hours, which is about four times faster. The time fraction spent on AED is reduced to 39%.

5.3 Benchmark examples

Besides random matrices, we also report performance results for some commonly used benchmark matrices. For comparison, we have tested the same matrices as in [21], see Table 11. The execution times for the three solvers are listed in Tables 12–19. The conclusions are similar to those we have made for random matrices: our earlier version of PDHSEQR outperforms the ScaLAPACK 1.8.0 routine PDLAHQR by a large extent; the new PDHSEQR is usually even faster, especially for BBMSN and GRCAR.

In [21], it was observed that the accuracy for AF23560 is not fully satisfactory; the relative residuals _R_r_ = ∥Q^T AQ – T∥_F/∥A∥_F were large for both PDLAHQR and PDHSEQR. It turns out that these large residuals are caused by an anomaly in PDLAHQR, which has been fixed by avoiding the use of PDLACONSB, see Section 2.3.2. As a result, the new PDHSEQR always produce _R_r_ ∈ [10^{-15}, 10^{-13}] for all test matrices.

6 Conclusions and Future Work

We have presented a new parallel implementation of the multishift QR algorithm with aggressive early deflation. The new routine PDHSEQR combines a number of techniques to improve serial performance and reduce communication. These include performing multiple levels of AED, reducing communication overhead by data redistribution, and refining the strategy for balancing between multishift QR sweeps and AED. Our numerical experiments provide compelling evidence that PDHSEQR significantly outperforms not only the original ScaLAPACK routine PDLAHQR but also an earlier version of PDHSEQR presented in [21]. In particular, our new implementation removes a bottleneck in the aggressive early deflation strategy by reducing communication and tuning algorithmic parameters. As a result, our new version is both faster and more robust. An intermediate version of the software presented in this paper is available in ScaLAPACK version 2.0.
### Table 11: Benchmark matrices.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Dimension (n)</th>
<th>Type/Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BBMSN [13]</td>
<td>n</td>
<td>$S_n = \begin{bmatrix} n &amp; n-1 &amp; n-2 &amp; \cdots &amp; 2 &amp; 1 \ 10^{-3} &amp; 1 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 10^{-3} &amp; 2 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ \vdots &amp; \vdots &amp; \vdots &amp; \vdots &amp; \vdots &amp; \vdots \ 10^{-3} &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 10^{-3} &amp; n-2 &amp; 0 &amp; \cdots &amp; \cdots &amp; 10^{-3} \ n-1 &amp; 0 &amp; \cdots &amp; \cdots &amp; \cdots &amp; 0 \ 10^{-3} &amp; 0 &amp; \cdots &amp; \cdots &amp; \cdots &amp; 0 \ \end{bmatrix}$</td>
</tr>
<tr>
<td>2</td>
<td>AF23560 [4]</td>
<td>23560</td>
<td>Computational fluid dynamics</td>
</tr>
<tr>
<td>3</td>
<td>CRY10000 [4]</td>
<td>10000</td>
<td>Material science</td>
</tr>
<tr>
<td>4</td>
<td>OLM5000 [4]</td>
<td>5000</td>
<td>Computational fluid dynamics</td>
</tr>
<tr>
<td>5</td>
<td>DW8192 [4]</td>
<td>8192</td>
<td>Electrical engineering</td>
</tr>
<tr>
<td>6</td>
<td>MATTRAN [4]</td>
<td>n</td>
<td>Sparse random matrix</td>
</tr>
<tr>
<td>8</td>
<td>GRCAR [4]</td>
<td>n</td>
<td>$G_n = \begin{bmatrix} 1 &amp; 1 &amp; 1 &amp; 1 \ -1 &amp; 1 &amp; 1 &amp; 1 \ -1 &amp; 1 &amp; 1 &amp; 1 \ \vdots &amp; \vdots &amp; \vdots &amp; \vdots \ -1 &amp; 1 &amp; 1 &amp; 1 \ -1 &amp; 1 &amp; 1 \ -1 &amp; 1 \ -1 \ \end{bmatrix}$</td>
</tr>
</tbody>
</table>

### Table 12: Execution time (in seconds) for BBMSN.

<table>
<thead>
<tr>
<th>n</th>
<th>$p = p_r \times p_c$</th>
<th>S-v180</th>
<th>SISC</th>
<th>NEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>$1 \times 1$</td>
<td>523</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>10000</td>
<td>$2 \times 2$</td>
<td>1401</td>
<td>30</td>
<td>9</td>
</tr>
<tr>
<td>15000</td>
<td>$3 \times 3$</td>
<td>1489</td>
<td>62</td>
<td>13</td>
</tr>
</tbody>
</table>

### Table 13: Execution time (in seconds) for AF23560.

<table>
<thead>
<tr>
<th>$p = p_r \times p_c$</th>
<th>S-v180</th>
<th>SISC</th>
<th>NEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>4×4</td>
<td>15486</td>
<td>2651</td>
<td>1375</td>
</tr>
<tr>
<td>6×6</td>
<td>9088</td>
<td>1279</td>
<td>826</td>
</tr>
<tr>
<td>8×8</td>
<td>6808</td>
<td>793</td>
<td>563</td>
</tr>
<tr>
<td>10×10</td>
<td>5694</td>
<td>662</td>
<td>475</td>
</tr>
<tr>
<td>12×12</td>
<td>5422</td>
<td>578</td>
<td>404</td>
</tr>
</tbody>
</table>

### Table 14: Execution time (in seconds) for CRY10000.

<table>
<thead>
<tr>
<th>$p = p_r \times p_c$</th>
<th>S-v180</th>
<th>SISC</th>
<th>NEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>4×4</td>
<td>15486</td>
<td>2651</td>
<td>1375</td>
</tr>
<tr>
<td>6×6</td>
<td>9088</td>
<td>1279</td>
<td>826</td>
</tr>
<tr>
<td>8×8</td>
<td>6808</td>
<td>793</td>
<td>563</td>
</tr>
<tr>
<td>10×10</td>
<td>5694</td>
<td>662</td>
<td>475</td>
</tr>
<tr>
<td>12×12</td>
<td>5422</td>
<td>578</td>
<td>404</td>
</tr>
</tbody>
</table>

### Table 15: Execution time (in seconds) for OLM5000.

<table>
<thead>
<tr>
<th>$p = p_r \times p_c$</th>
<th>S-v180</th>
<th>SISC</th>
<th>NEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>4×4</td>
<td>15486</td>
<td>2651</td>
<td>1375</td>
</tr>
<tr>
<td>6×6</td>
<td>9088</td>
<td>1279</td>
<td>826</td>
</tr>
<tr>
<td>8×8</td>
<td>6808</td>
<td>793</td>
<td>563</td>
</tr>
<tr>
<td>10×10</td>
<td>5694</td>
<td>662</td>
<td>475</td>
</tr>
<tr>
<td>12×12</td>
<td>5422</td>
<td>578</td>
<td>404</td>
</tr>
</tbody>
</table>

### Table 16: Execution time (in seconds) for DW8192.

<table>
<thead>
<tr>
<th>$p = p_r \times p_c$</th>
<th>S-v180</th>
<th>SISC</th>
<th>NEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>4×4</td>
<td>15486</td>
<td>2651</td>
<td>1375</td>
</tr>
<tr>
<td>6×6</td>
<td>9088</td>
<td>1279</td>
<td>826</td>
</tr>
<tr>
<td>8×8</td>
<td>6808</td>
<td>793</td>
<td>563</td>
</tr>
<tr>
<td>10×10</td>
<td>5694</td>
<td>662</td>
<td>475</td>
</tr>
<tr>
<td>12×12</td>
<td>5422</td>
<td>578</td>
<td>404</td>
</tr>
</tbody>
</table>
Table 17: Execution time (in seconds) for MATRAN.

<table>
<thead>
<tr>
<th>$p \times p_c$</th>
<th>$n = 5000$</th>
<th>$n = 10000$</th>
<th>$n = 15000$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S-v180</td>
<td>SISC NEW</td>
<td>S-v180</td>
</tr>
<tr>
<td>1 x 1</td>
<td>1617</td>
<td>332</td>
<td>218</td>
</tr>
<tr>
<td>2 x 2</td>
<td>579</td>
<td>152</td>
<td>122</td>
</tr>
<tr>
<td>4 x 4</td>
<td>247</td>
<td>74</td>
<td>60</td>
</tr>
<tr>
<td>6 x 6</td>
<td>178</td>
<td>64</td>
<td>57</td>
</tr>
<tr>
<td>8 x 8</td>
<td>147</td>
<td>58</td>
<td>50</td>
</tr>
<tr>
<td>10 x 10</td>
<td>149</td>
<td>59</td>
<td>43</td>
</tr>
</tbody>
</table>

Table 18: Execution time (in seconds) for MATPDE.

<table>
<thead>
<tr>
<th>$p \times p_c$</th>
<th>$n = 10000$</th>
<th>$n = 14400$</th>
<th>$n = 19600$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S-v180</td>
<td>SISC NEW</td>
<td>S-v180</td>
</tr>
<tr>
<td>1 x 1</td>
<td>12429</td>
<td>2600</td>
<td>1699</td>
</tr>
<tr>
<td>2 x 2</td>
<td>3531</td>
<td>1081</td>
<td>966</td>
</tr>
<tr>
<td>4 x 4</td>
<td>1565</td>
<td>415</td>
<td>361</td>
</tr>
<tr>
<td>6 x 6</td>
<td>1118</td>
<td>256</td>
<td>225</td>
</tr>
<tr>
<td>8 x 8</td>
<td>871</td>
<td>189</td>
<td>156</td>
</tr>
<tr>
<td>10 x 10</td>
<td>789</td>
<td>189</td>
<td>137</td>
</tr>
<tr>
<td>12 x 12</td>
<td>719</td>
<td>194</td>
<td>126</td>
</tr>
</tbody>
</table>

Table 19: Execution time (in seconds) for GRCAR.

<table>
<thead>
<tr>
<th>$p \times p_c$</th>
<th>$n = 6000$</th>
<th>$n = 12000$</th>
<th>$n = 18000$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S-v180</td>
<td>SISC NEW</td>
<td>S-v180</td>
</tr>
<tr>
<td>1 x 1</td>
<td>2738</td>
<td>1340</td>
<td>69</td>
</tr>
<tr>
<td>2 x 2</td>
<td>850</td>
<td>645</td>
<td>40</td>
</tr>
<tr>
<td>4 x 4</td>
<td>363</td>
<td>258</td>
<td>226</td>
</tr>
<tr>
<td>6 x 6</td>
<td>244</td>
<td>190</td>
<td>173</td>
</tr>
<tr>
<td>8 x 8</td>
<td>217</td>
<td>150</td>
<td>145</td>
</tr>
<tr>
<td>10 x 10</td>
<td>207</td>
<td>161</td>
<td>126</td>
</tr>
</tbody>
</table>

20
Concerning future work, we believe to have come to a point, where it will be difficult to attain further dramatic performance improvements for parallel nonsymmetric eigensolvers, without leaving the classical framework of QR algorithms. Considering the fact that the execution times spent on Hessenberg reduction and on QR iterations are now nearly on the same level, any further improvement of the iterative part will only have a limited impact on the total execution time. The situation is quite different when shared memory many-core processors with accelerators, such as GPUs, are considered. Although efficient implementations of the Hessenberg reduction on such architectures have recently been proposed [31, 29, 36], the iterative part remains to be done. Another future challenge is to combine the message passing paradigm used in this new implementation of the multishift QR algorithm and dynamic and static scheduling on many-core nodes using multithreading.

**Acknowledgements**

We are grateful to Björn Adlerborn and Lars Karlsson for constructive discussions and comments on the subject, and to Åke Sandgren for support at HPC2N. We also thank David Guerrero, Rodney James, Julien Langou, Jack Poulson, Jose Roman, as well as anonymous users from IBM for helpful feedback.

**References**


Estimating $T_{\text{sweep}}$

The QR sweep is relatively simple because the computation and communication cost is well-determined by $n$, $n_{\text{shift}}$, and $p$. Usually there are up to $\sqrt{p}$ simultaneous computational windows, one at each diagonal processor in the grid, with at most $n_b/3$ shifts in each window. If $n_{\text{shift}} > \sqrt{p} n_b/3$, these shifts are chased in several rounds. So we use a rough approximation $n_{\text{shift}}^* = \sqrt{p} n_b/3$ to represent the total amount of shifts which can be chased simultaneously in the QR sweep. Based on the assumption $\sqrt{p} n_b \ll n$, the overhead for the start-up and ending phases of the bulge chasing are not important. Therefore the cost of one QR sweep is roughly

$$T_{\text{sweep}}(n, n_{\text{shift}}, p) = \frac{n_{\text{shift}}}{n_{\text{shift}}^*} n_b (T_{\text{local}} + T_{\text{cross}}),$$

where $T_{\text{local}}$ and $T_{\text{cross}}$ represent the runtime for local and crossborder bulge chasing, respectively. Both parts require chasing the chain of bulges with $n_b/2$ steps inside the computational window, as well as updating the corresponding off-diagonal blocks. Hence the runtime for arithmetic operations is $(4n_b^3 + 4n n_b^2/\sqrt{p}) \gamma$, half of which is for accumulating the orthogonal matrix $Q$. The only communication cost in the local chasing phase is broadcasting the accumulated orthogonal matrix rowwise and columnwise in the processor grid, which requires $\log_2 p (\alpha + n_b^2 \beta)$ runtime, i.e.,

$$T_{\text{local}} = \left(4n_b^3 + \frac{4n n_b^2}{\sqrt{p}}\right) \gamma + \log_2 p (\alpha + n_b^2 \beta) \approx 4n_b^3 \frac{n_b^2}{\sqrt{p}} \gamma + \log_2 p (\alpha + n_b^2 \beta).$$

One round crossborder chasing requires at least the same amount of communication as in one local chasing step, with some extra cost for explicitly forming the $n_b \times n_b$ computational window and exchanging data with processor neighbours for updating the off-diagonal blocks. Notice that usually there are two rounds for a crossborder chasing step, therefore we have

$$T_{\text{cross}} = 2 \left[ T_{\text{local}} + 3 \left( \alpha + \frac{n_b^2}{4} \beta \right) + 3 \left( \alpha + \frac{n_b^2}{2 \sqrt{p}} \beta \right) \right],$$

and then

$$T_{\text{sweep}}(n, n_{\text{shift}}, p) \approx \frac{12n^2 n_{\text{shift}} n_b}{\sqrt{p} n_{\text{shift}}^*} \gamma + \frac{3n n_{\text{shift}}}{n_b} (\log_2 p + 4) \alpha + \frac{3n^2 n_{\text{shift}}}{\sqrt{p} n_{\text{shift}}^*} \beta$$

$$= \frac{36n^2 n_{\text{shift}}}{p} \gamma + \frac{9n n_{\text{shift}}}{\sqrt{p} n_b} (\log_2 p + 4) \alpha + \frac{9n^2 n_{\text{shift}}}{pn_b} \beta.$$
From this model, we can see that the cost for updating the off-diagonal blocks dominates in both the computation and communication parts, under the assumption that $\sqrt{p}n_b \ll n$ (or equivalently $n_{\text{shift}}^2 \ll n$). As a byproduct, the performance model of a plain multishift QR algorithm without AED can also be obtained. By assuming the convergence rate as $\Theta(1)$ shifts per eigenvalue, i.e., $k_{\text{sweep}} = \Theta(n/n_{\text{shift}})$ and neglecting the cost for generating shifts, the total execution time of a plain multishift QR algorithm is

$$T_{\text{new}}(n, p) = \Theta\left(\left(\frac{n^3}{p}\right)\gamma + \Theta\left(\frac{n^2 \log p}{\sqrt{p} n_b}\right)\alpha + \Theta\left(\frac{n^3}{p n_b}\right)\beta\right).$$

Fixing the memory load per processor (i.e., $n/\sqrt{p}$ = constant) yields

$$T_{\text{new}}(n, p) = \Theta(n) \gamma + \Theta(n \log n) \alpha + \Theta(n) \beta.$$  

## 2 Estimating $T_{\text{AED}}$ and $T_{\text{shift}}$

The execution time for one step AED is modelled as

$$T_{\text{AED}}(n, n_{\text{AED}}, p) = T_{\text{redist}}(n_{\text{AED}}, p, p_{\text{AED}}) + T_{\text{pipe}}(n_{\text{AED}}, p_{\text{AED}}) + T_{\text{reorder}}(n_{\text{AED}}, p) + T_{\text{Hess}}(n_{\text{AED}}, p) + T_{\text{update}}(n, n_{\text{AED}}, p),$$

where the terms in the right-hand-side represent the runtime for data redistribution, Schur decomposition of the AED window, deflation checking and reordering of eigenvalues, Hessenberg reduction, and updating the off-diagonal blocks corresponding to the AED window, respectively. We estimate these terms one by one using the hierarchical approach in [3].

- $T_{\text{redist}}$: The general-purpose data redistribution routine $\text{PDGEMR2D}$ in ScaLAPACK uses the algorithm described in [7]. Since the scheduling part is tiny compared to the communication part, the complexity of data redistribution is provided [7] as

$$T_{\text{redist}}(n_{\text{AED}}, p, p_{\text{AED}}) = \Theta(p) \alpha + \Theta\left(\frac{n_{\text{AED}}^2}{\sqrt{p} p_{\text{AED}}}\right)\beta.$$  

- $T_{\text{pipe}}$: The complexity of the Schur decomposition performed by $\text{PDLAQR1}$ largely depends on the property of the matrix, since AED affects the convergence rate significantly. To obtain an estimate of the complexity, we assume that AED roughly reduces the number of pipelined QR sweeps by half. According to the experimental results presented in [6], this assumption usually provides a reasonable upper bound of the runtime, although it can be overestimated. Using the model in [5], we obtain an approximate execution time

$$T_{\text{pipe}}(n, p) = \frac{20n^3}{p} \gamma + \frac{3n^2}{\sqrt{p} n_b} \left(\log_2 p + 2\right)\alpha + \left(\frac{3n^2 \log_2 p}{\sqrt{p}} + \frac{8n^3}{p n_b}\right)\beta$$

$$= \Theta\left(\frac{n^3}{p}\right) \gamma + \Theta\left(\frac{n^2 \log p}{\sqrt{p} n_b}\right)\alpha + \Theta\left(\frac{n^3}{p n_b}\right)\beta. \quad (1)$$

If the orthogonal matrix $Q$ is not accumulated in the calculation, the arithmetic operations are roughly halved, i.e.,

$$\tilde{T}_{\text{pipe}}(n, p) = \frac{10n^3}{p} \gamma + \frac{3n^2}{\sqrt{p} n_b} \left(\log_2 p + 2\right)\alpha + \left(\frac{3n^2 \log_2 p}{\sqrt{p}} + \frac{8n^3}{p n_b}\right)\beta.$$  

The model provided in [1] is similar, but with slightly different coefficients.
• $T_{\text{reorder}}$: Obviously, the cost for eigenvalue reordering depends on the deflation ratio. However, we can evaluate an upper bound for the cost—all eigenvalues are involved in the reordering. Then the performance model is almost the same as that of QR sweeps, since updating the off-diagonal blocks is the dominant operation. Notice that each eigenvalue needs to move $n_{\text{AED}}/2$ steps in average, so the overall cost for eigenvalue reordering inside the AED window is bounded by

$$T_{\text{reorder}}(n_{\text{AED}}, p) \approx \frac{4n_{\text{AED}}^2 n_b}{\sqrt{p}} \gamma + \frac{2n_{\text{AED}}}{n_b} (\log_2 p + 3) \alpha + \frac{3n_{\text{AED}}^2}{2\sqrt{p}} \beta.$$  

As a different feature compared to QR sweeps or the performance model in [4] for parallel eigenvalue reordering, the degree of concurrency here is $\Theta(\sqrt{p})$ instead of $\Theta(p)$ since usually there are at most two computational windows for the reordering phase inside the AED window.

• $T_{\text{Hess}}$: The Hessenberg reduction routine PDGEHRD uses the parallel algorithm described in [2]. Almost all computations and communication are performed on matrix-vector and matrix-matrix multiplications. Therefore we need to model these PBLAS operations first. The level 2 operations GEMV and GER require

$$T_{\text{GEMV}}(m, n, p) \approx T_{\text{GEMV}}(m, m, n, p) \approx \frac{2mn}{p} \gamma + \log_2 p (\alpha + \frac{m+n}{2}\beta).$$

where $m \times n$ is the size of the matrix. This model can be directly generalized to multiplying two $m \times k$ and $k \times n$ matrices as long as $\min(m, n, k) \leq n_b$ since it is merely a “fat” level 2 operation. In the Hessenberg reduction algorithm, all level 3 operations are “fat” level 2 operations, so the cost for one GEMM operation can be modelled as

$$T_{\text{GEMM}}(m, n, b, p) \approx T_{\text{GEMM}}(m, n_b, n, p) \approx \frac{2mn_b}{p} \gamma + \log_2 p (\alpha + \frac{(m+n)n_b}{2}\beta).$$

(2)

Using these simple models of PBLAS operations, we are able to establish a model for $T_{\text{Hess}}$. The level 2 part consists roughly of $n$ matrix-vector multiplications of dimension $n \times (n-j)$ (for $j = 1, 2, \ldots, n$). Therefore the cost is

$$T_{\text{level2}} \approx \sum_{j=1}^{n} \left[ \frac{2n(n-j)}{p} \gamma + \log_2 p (\alpha + \frac{n-j}{2}\beta) \right] \approx \frac{n^3}{p} \gamma + \log_2 p (n\alpha + \frac{3n^2}{2}\beta).$$

The level 3 part contains roughly $n/n_b$ iterations with one PDGEMM and one PDLARFB per iteration. Within the $j$th iteration ($j = 1, 2, \ldots, n/n_b$), PDGEMM involves matrices of dimension $n \times n_b$ and $n_b \times (n-j)n_b-n_b$; PDLARFB mainly performs two parallel GEMM operations, with $\{n_b \times (n-j)n_b, (n-j)n_b \times (n-j)n_b\}$ and $\{(n-j)n_b \times n_b, n_b \times (n-j)n_b\}$ matrices involved. Another sequential TRMM operation in PDLARFB is neglected since it only contributes lower order terms in both arithmetic and communication costs. So the cost for level 3 part is

$$T_{\text{level3}} \approx \frac{n(n_b)}{p} \left[ \frac{2jn_b + 6(n-j)n_b}{n_b^2}n_b(n-j)n_b \gamma + \log_2 p (3\alpha + \frac{6n-5jn_b}{2}\beta) \right]$$

$$\approx \frac{n^3}{3p} \gamma + \frac{3n \log_2 p}{n_b} \alpha + \frac{7n^2 \log_2 p}{4\sqrt{p}} \beta,$$

and hence the execution time for Hessenberg reduction (without explicitly forming the orthogonal matrix) is

$$\tilde{T}_{\text{Hess}}(n, p) = T_{\text{level2}} + T_{\text{level3}} \approx \frac{10n^3}{3p} \gamma + n \log_2 p \alpha + \frac{5n^2 \log_2 p}{2\sqrt{p}} \beta.$$  

(3)

Even if the proportion of level 3 operations is improved to 80% as suggested in [8] but not implemented in the current PDGEHRD yet, the estimate in (3) would not change too much since the number of messages in the level 2 part is not reduced.
Since the Householder reflections are stored in a compact form in the lower triangular part of the upper Hessenberg matrix, formulating the orthogonal matrix after Hessenberg reduction is another necessary step. This step is done by the ScalAPACK routine \textsc{PDORMHR}, which is mainly a series of calls to \textsc{PDLARFB}. Similar to the discussion above, we obtain

\[ T_{\text{ORMHR}} \approx \frac{2n^3}{p} + \frac{3n \log_2 p}{n_b} + \frac{7n^2}{4\sqrt{p}}. \]

Therefore the total runtime for the Hessenberg reduction process including formulating the orthogonal matrix is

\[ T_{\text{Hess}}(n, p) = T_{\text{Hess}} + T_{\text{ORMHR}} \approx \frac{16n^3}{3p} + \frac{n \log_2 p}{\sqrt{p}} + \frac{17n^2 \log_2 p}{4\sqrt{p}}. \tag{4} \]

- \( T_{\text{update}} \): The cost for updating the off-diagonal blocks with respect to the AED window is simple to analyze since it merely contains three GEMM operations. Since these GEMM operations are not “fat” level 2 operations, we need to use a model different to (2). According to [9], the execution time for a GEMM operation on a \( \sqrt{p} \times \sqrt{p} \) processor grid with \( m \times k \) and \( k \times n \) matrices involved is

\[ T_{\text{GEMM}}(m, n, k, p) \approx \frac{2mnk}{p} + \left(\frac{k}{n_b} + 2\sqrt{p}\right)\left(2\alpha + \frac{(m + n)n_b}{\sqrt{p}}\beta\right) \]

if \( \min(m, n, k) = k \gg n_b \). Then we conclude that

\[ T_{\text{update}}(n, n_{\text{AED}}, p) \approx \frac{2n^2_{\text{AED}}}{p} + \frac{n_{\text{AED}}}{n_b} \left(6\alpha + \frac{2n_b}{\sqrt{p}}\beta\right). \]

Now we are ready to estimate the overall runtime \( T_{\text{AED}} \) by substituting \( n \) with \( n_{\text{AED}} \) in (1) and (4). We can see that \( T_{\text{redist}} \) is always negligible compared to other components. Reordering contributes with only marginal communication costs also. By merging all these estimates together, we eventually obtain

\[ T_{\text{AED}}(n, n_{\text{AED}}, p) \approx T_{\text{pipe}}(n_{\text{AED}}, p_{\text{AED}}) + T_{\text{reorder}}(n_{\text{AED}}, p) + T_{\text{Hess}}(n_{\text{AED}}, p) + T_{\text{update}}(n, n_{\text{AED}}, p) \]

\[ \approx \left(\frac{20n_{\text{AED}}}{p_{\text{AED}}} + 4\sqrt{p}n_b + 16n_{\text{AED}} + 2n\right) n_{\text{AED}}^2 \gamma 
+ \frac{n_{\text{AED}}^2}{n_b} \left(3\log_2 p_{\text{AED}} + 2\right) n_{\text{AED}} \alpha 
+ \frac{n_{\text{AED}}^2}{n_b} \left(3n_{\text{AED}} \log_2 p_{\text{AED}} + 8n_{\text{AED}} n_{\text{AED}} \log_2 p_{\text{AED}} + 3n_{\text{AED}} + 17n_{\text{AED}} \log_2 p_{\text{AED}} + 2n_{\text{AED}} \log_2 p_{\text{AED}} \beta \right) \]

\[ \approx \left[ \frac{30C_2^2 C_1 n}{C_1^2 \sqrt{p}} + \frac{9n^2 n_b}{C_1^2 \sqrt{p}} + \frac{9(C_1 + 6)n^3}{2C_1 p} \right] \gamma 
+ \left[ \frac{9C_2^2 C_1 n}{C_1^2 n_b \log_2 \log_2 p_{\text{AED}}} + \frac{3n}{2C_1 \log_2 p_{\text{AED}}} \right] \alpha 
+ \left[ \frac{9C_2^2 n}{C_1 \log_2 \log_2 \log_2 p_{\text{AED}}} + \frac{12C_2^2 n}{C_1 n_b} + \frac{3n^2 (18 + C_1 + 51 \log_2 p_{\text{AED}})}{16C_1^2 \sqrt{p}} \right] \beta. \]

When \( n \) is extremely large (i.e., \( C_1, C_2 \) and \( n_b \) are all tiny enough compared to \( n \)) and \( n/\sqrt{p} = \text{constant} \), we have

\[ T_{\text{AED}} = \Theta\left(n + \frac{n^2}{\sqrt{p}} + \frac{n^3}{p}\right) \gamma + \Theta(n \log n + n \log p) \alpha + \Theta(n \log n + \frac{n^2}{\sqrt{p}} \log p) \beta \]

\[ = \Theta(n) \gamma + \Theta(n \log n) \alpha + \Theta(n \log n) \beta. \]
Asymptotically AED only has slightly larger message sizes by a $\Theta(\log n)$ factor compared to QR sweeps and is hence not much more expensive. However, in practice we still need to handle AED very carefully since large leading factors in lower order terms have significant impact on the performance if the matrix is not large enough. Similar to the analysis for $T_{AED}$, the cost for computing shifts can be estimated by

$$T_{\text{shift}}(n, n_{\text{shift}}, p) \approx \tilde{T}_{\text{pipe}}(n_{\text{shift}}, p_{\text{shift}}) \approx \frac{10n_{\text{shift}}^3}{p_{\text{shift}}} \gamma + \frac{3n_{\text{shift}}^2}{\sqrt{p_{\text{shift}} n_b}} (\log_2 p_{\text{shift}} + 2) \alpha + \left( \frac{3n_{\text{shift}}^2 \log_2 p_{\text{shift}}}{\sqrt{p_{\text{shift}}}} + \frac{8n_{\text{shift}}^3}{p_{\text{shift}} n_b} \right) \beta \approx \frac{10C_2^2 n}{C_1} \gamma + \frac{6C_2 n}{C_1 n_b} \log_2 \frac{n}{C_1 C_2} \alpha + \left( \frac{6C_2 n}{C_1} \log_2 \frac{n}{C_1 C_2} + \frac{8C_2^2 n}{C_1 n_b} \right) \beta.$$

Asymptotically $T_{\text{shift}}$ is not so important in the scalability analysis since it can never be larger than $T_{AED}$.

References


