Algorithm 953: Parallel Library Software for the Multishift QR Algorithm with Aggressive Early Deflation

ROBERT GRANAT and BO KÅGSTRÖM, Umeå University
DANIEL KRESSNER, EPF Lausanne
MEIYUE SHAO, Umeå University and EPF Lausanne

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.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: *Algorithm design and analysis*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Multishift QR algorithm, aggressive early deflation, parallel algorithms, distributed memory architectures

ACM Reference Format:
Robert Granat, Bo Kågström, Daniel Kressner, and Meiyue Shao. 2015. Algorithm 953: Parallel library software for the multishift QR algorithm with aggressive early deflation. ACM Trans. Math. Softw. 41, 4, Article 29 (October 2015), 23 pages.
DOI: http://dx.doi.org/10.1145/2699471

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 article 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 Granat et al. [2010], a number of additional algorithmic improvements have been made, leading to significantly reduced execution times and higher robustness.

This work is supported by the Swedish Research Council under grant A0581501, UMIT Research Lab via an EU Mål 2 project, and eSSENCE, a strategic collaborative e-Science programme funded by the Swedish Research Council.
Authors' addresses: R. Granat and B. Kågström, Department of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden; emails: {granat, bokg}@cs.umu.se; D. Kressner, MATHICSE, EPF Lausanne, CH-1015 Lausanne, Switzerland; email: daniel.kressner@epfl.ch; M. Shao, Department of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden; MATHICSE, EPF Lausanne, CH-1015 Lausanne, Switzerland; email: meiyue.shao@epfl.ch.

DOI: http://dx.doi.org/10.1145/2699471

In the following, we give a brief history of serial and parallel implementations of the QR algorithm. The Algol procedure `hqr` by Martin et al. [1970] was among the first computer implementations of the QR algorithm. A Fortran translation of this procedure was included in EISPACK [Smith et al. 1976] as routine `HQR`. The initial version of the LAPACK routine `DHSEQR` was based on work by Bai and Demmel [1989]; 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 [Ahues and Tisseur 1997] 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 et al. [2002a, 2002b]. This implementation is described in more detail in Byers [2007]. While there has been a lot of early work on parallelizing the QR algorithm, for example in Henry and van de Geijn [1996], Schreiber et al. [1994], van de Geijn [1988, 1993], van de Geijn and Hudson [1989], and Watkins [1994], the first publicly available parallel implementation was only released in 1997 in ScaLAPACK [Blackford et al. 1997] version 1.5 as routine `PDLAHQR`, based on work by Henry et al. [2002]. A complex version of this routine, `PZLAHQR`, was included later [Fahey 2003]. 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 [Agullo et al. 2009], ELPA [Auckenthaler et al. 2011] and FLAME [Bientinesi et al. 2005] lack adapted implementations of either 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 $A \leftarrow D^{-1}AD$ 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 1×1 blocks (corresponding to real eigenvalues) and 2×2 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 article is only concerned with the reduction to real Schur form (1). In particular, we will not discuss the implementation of Hessenberg reduction (see Karlsson and Kågström [2011], Kågström et al. [2008], and Tomov et al. [2010] for recent developments in this direction).

The rest of this article is organized as follows. Section 2 provides a summary of our implementation and the underlying algorithm, emphasizing improvements over Granat et al. [2010]. 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 [Granat et al. 2014a]. 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 using 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

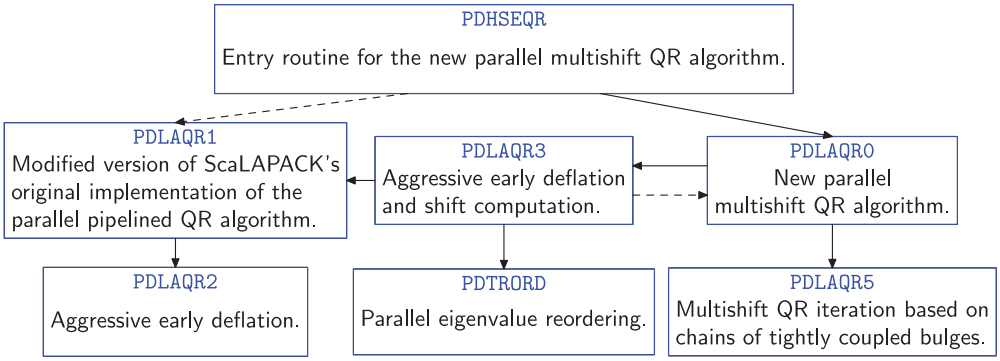

Fig. 1. Software hierarchy of the parallel multishift QR algorithm with AED.

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, several issues need to be considered to attain 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 $n_{\text{AED}} \times n_{\text{AED}}$ 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 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

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

Fig. 2. The 2D block-cyclic data layout across a 2×3 processor grid. For example, processor $(0, 0)$ owns all highlighted blocks.

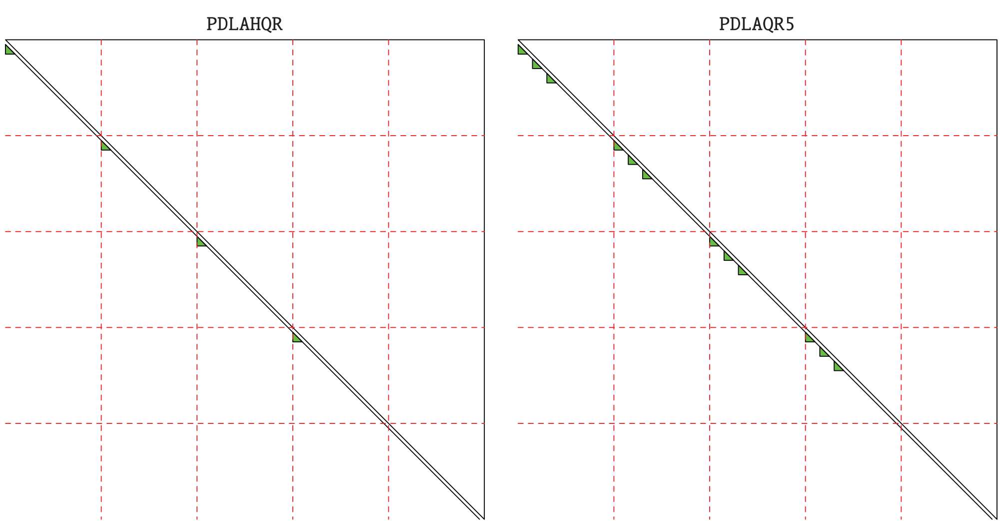

Fig. 3. Loosely coupled shifts vs. tightly coupled shifts. The dashed lines represent borders of the processor grid. Only parts of the matrix are displayed. Intrablock bulge chasings can be performed independently and in parallel.

shift polynomial:

$$(H - \sigma_1 I)(H - \sigma_2 I) \cdots (H - \sigma_k I) = QR.$$

Thanks to the *Implicit Q theorem* (for example, see Francis [1962] and Golub and Van Loan [1996]), we have a large amount 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))

Table I. Recommended Values for n_{shift} and n_{AED} Values taken from Table 2.1 in Granat et al. [2010] for $n \leq 96$K; values for $n > 96$K 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.

Matrix size (n)	n_{shift}	n_{AED}
<6K	see Byers [2007]	
6K–12K	256	384
12K–24K	512	768
24K–48K	1024	1,536
48K–96K	2048	3,072
96K–192K	4096	6,144
192K–384K	8192	12,288
384K–768K	16384	24,576
768K–1,000K	32768	49,152
> 1M	$\lceil n/25 \rceil$	$3n_{\text{shift}}/2$

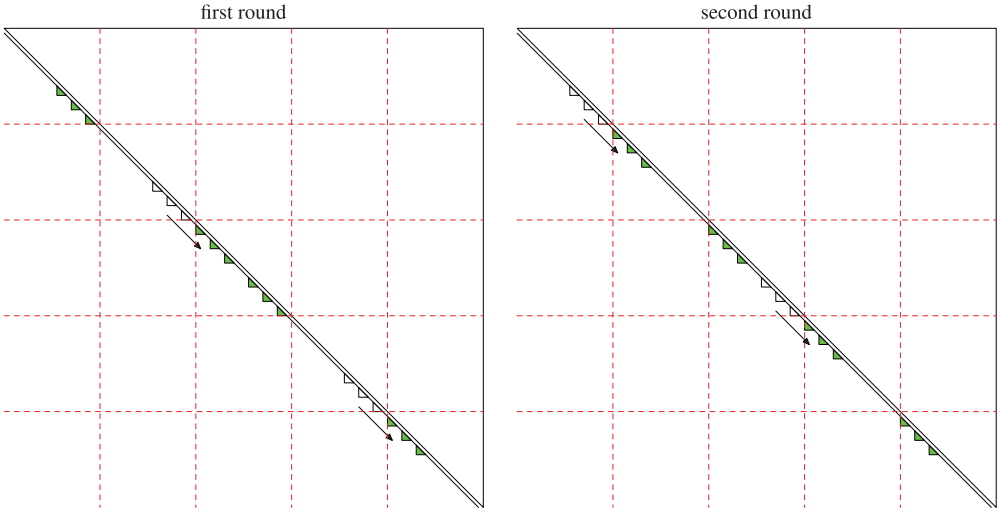

Fig. 4. Interblock bulge chasing. Odd-numbered chains (left) and even-numbered chains (right) are chased separately in two rounds.

to improve node performance and reduce communication. For the new bulge chasing routine, PDLAQR5, the total number of shifts, n_{shift}, used in a single QR sweep is shown in Table I and is usually much larger than that employed with the pipelined approach. These shifts are divided into several chains of tightly coupled bulges with up to $\lfloor n_b/3 \rfloor$ 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 BLAS 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 Granat et al. [2010] for a detailed description of this technique.

2.3. Aggressive Early Deflation (AED)

The AED technique has been proposed by Braman et al. [2002b] and proceeds by partitioning the current upper Hessenberg matrix $H \in \mathbb{R}^{n\times n}$ as

$$H = \begin{array}{c} \\ n-n_{\mathrm{AED}}-1 \\ 1 \\ n_{\mathrm{AED}} \end{array} \begin{array}{c} \begin{array}{ccc} n-n_{\mathrm{AED}}-1 & 1 & n_{\mathrm{AED}} \end{array} \\ \begin{pmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ 0 & H_{32} & H_{33} \end{pmatrix} \end{array},$$

where $H_{33} \in \mathbb{R}^{n_{\mathrm{AED}}\times n_{\mathrm{AED}}}$ 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

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

where

$$U = \begin{array}{c} \\ n-n_{\mathrm{AED}}-1 \\ 1 \\ n_{\mathrm{AED}} \end{array} \begin{array}{c} \begin{array}{ccc} n-n_{\mathrm{AED}}-1 & 1 & n_{\mathrm{AED}} \end{array} \\ \begin{pmatrix} I & & \\ & 1 & \\ & & V \end{pmatrix} \end{array}.$$

The vector $s \in \mathbb{R}^{n_{\mathrm{AED}}}$ is the so-called *spike*, created from the first entry of the vector H_{32}. The last diagonal entry (or 2×2 diagonal block) of T can be deflated if the magnitude of the last component (or the last two components) of the spike is negligible. Undeflatable eigenvalues are moved to the top left corner of T by a swapping algorithm [Bai and Demmel 1993; Granat et al. 2009]. The orthogonal transformations for reordering eigenvalues in the Schur form of the AED window are accumulated in an $n_{\mathrm{AED}} \times n_{\mathrm{AED}}$ orthogonal matrix. By repeating the same procedure on all diagonal entries (or 2×2 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 $\widetilde{V}$, the product of all involved orthogonal transformations. Typically, the size of the AED window is recommended to be somewhat larger, for example, by 50%, than the number of shifts in the multishift QR sweeps [Braman et al. 2002a; Byers 2007] (see also Table I).

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 for performing AED, `PDLAQR3`, requires a parallel approach.

The first and most costly step of AED is to calculate the Schur decomposition of H_{33}, the $n_{\mathrm{AED}} \times n_{\mathrm{AED}}$ 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_{AED} is relatively small compared to n, the number of available processors may be too large to use all of them without causing significant communication overhead. In this case, we only use

a subset of the processors to reduce the overhead and to minimize 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 Braman et al. [2002a], undeflatable eigenvalues are not moved immediately and individually toward 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, is the group of undeflatable eigenvalues 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 of the pipelined QR algorithm, `PDLAHQR`, 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 efficiently on a single processor, 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 Kågström et al. [2012] and Shao [2011].

—Aggressive early deflation: AED is implemented in an auxiliary routine, `PDLAQR2`, which copies the AED window to local memory and calls a 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 LAPACK routines, `DLAHQR`/`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 [1962] to benefit potentially from two consecutive small

subdiagonal entries. In the rare case when it is successful, this strategy allows bulges to be introduced 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, that 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_{undflt} denote the number of undeflatable shifts in an AED step, the multishift QR sweep is skipped if

$$\frac{n_{\text{AED}} - n_{\text{undflt}}}{n_{\text{AED}}} \geq \frac{\texttt{NIBBLE}}{100}.$$

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 previously, the AED process only involves a smaller trailing submatrix, which leads to decreased parallel efficiency. To counter this, NIBBLE needs to be set larger to avoid performing AED too frequently. A good choice for this threshold depends both on the size of the matrix, H, and the number of processors involved. We use the model $\texttt{NIBBLE} = a \cdot n^b p^c$ for this purpose, where a, b, and c are machine-dependent constants, and an appropriate choice for 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 $\texttt{NIBBLE} > 33$. 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 $n_{\text{AED}} = 3n_{\text{shift}}/2$. The solution in the software is that as long as $n_{\text{undflt}} \geq n_{\text{shift}}/2$, we only use these n_{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 $\texttt{NIBBLE} \geq 67$. 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 may 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, for example, Grama et al. [2003]). This technique has already been employed in previous implementations of the parallel QR algorithm, such as PDLAHQR and software developed in Granat et al. [2010] and Kågström et al. [2012]. 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 Kågström et al. [2012], we proposed to duplicate these sequential parts on all involved processors. The parallel update of the off-diagonal blocks may 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 being 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 cross-border 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 (say, $p \leq 100$), the change in performance is negligible; while for a large number of processors, the performance can drop. For example, when computing the Schur decomposition of a $100{,}000 \times 100{,}000$ matrix using 40×40 processors, up to 25% performance drop has been observed when replacing task duplication with explicit communication.

2.6. Avoiding Communication via Data Redistribution

As observed in Kågström et al. [2012], the parallel QR algorithm may be inefficient at solving a relatively small eigenvalue problem on many processors, due to excessive communication, to the point that the execution time actually increases as the number of processors increases. 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, for example, the spectral divide-and-conquer algorithm [Bai et al. 1997b; Ballard et al. 2010]. 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 [Ballard et al. 2010, 2011; Hoemmen 2010; Irony et al. 2004] 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 almost evenly distributed over the processors. Here we propose an alternative approach that 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 that 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 this way since we know in advance that the AED window is always sufficiently small for the associated Schur decomposition to be efficiently solved sequentially. By introducing the overhead of data redistribution, the total amount of communication as well as the execution time is 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, but still small 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 from ScaLAPACK; its overhead has been observed to be negligible relative to the AED process as a whole.

The tunable parameter $p_{\min} = \texttt{PILAENVX}(\texttt{ISPEC} = 23)$ determines our heuristic strategy for choosing the number of processors for the redistribution. If $\min(p_r, p_c) > p_{\min}+1$, we redistribute the AED window to a $p_{\min} \times p_{\min}$ 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 $p_{\min} = \lceil n_{\text{AED}}/(n_b\lceil 384/n_b\rceil)\rceil$, 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, that is, bulge introduction and bulge annihilation, are not considered in the pseudocode. These algorithms are used within Algorithm 1 on distributed memory architectures.

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 $H \leftarrow U^T HU$ obtained after performing AED, $Q \leftarrow QU$, with U orthogonal.

1: Estimate the optimal process grid size, $p_{\text{AED}} = p_{\min}^2$, based on $n_{\text{AED}} = j - i + 1$ and n_b.
2: **if** $\min(p_r, p_c) \leq p_{\min} + 1$ **then**
3: $\quad \hat{p}_r \leftarrow p_{\min}$, $\hat{p}_c \leftarrow p_{\min}$.
4: **else**
5: $\quad \hat{p}_r \leftarrow p_r$, $\hat{p}_c \leftarrow p_c$.
6: **end if**
7: Redistribute $H_0 \leftarrow H(i:j, i:j)$ to a $\hat{p}_r \times \hat{p}_c$ process subgrid if necessary.
8: Compute the Schur decomposition $H_0 = V_0 T_0 V_0^T$ on the $\hat{p}_r \times \hat{p}_c$ process subgrid.
9: Redistribute $T \leftarrow T_0$, $V \leftarrow V_0$ back to the original $p_r \times p_c$ process grid.
10: $s \leftarrow H(i, i-1)V(1,:)^T$.
11: **repeat**
12: $\quad$ 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: $\quad$ Move all undeflatable eigenvalues within this group of n_b eigenvalues to the top-left corner of the AED window.
14: $\quad$ 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 $H_1 \leftarrow V_3^T T V_3$ 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.

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 $H \leftarrow U^T HU$ 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 $(:, w)$ or $(w, :)$ **then**
3: **if** $(myrow, mycol)$ owns (w, w) **then**
4: Chase the bulge chain inside w down $\lfloor n_b/2 \rfloor$ 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 $(:, w)$ or $(w, :)$ **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 $\lceil n_b/2 \rceil$ 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**

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 costs 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 enough data blocks of the matrices, that is, $\sqrt{p}\,n_b \ll n$, 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 γ 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 α and β 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 modeled 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.

Let k_{AED} and k_{sweep} denote, respectively, the number of AED steps and QR sweeps 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_{shift}, which of course satisfies $k_{\text{shift}} \leq k_{\text{sweep}}$. Given the constants k_{AED}, k_{sweep}, and k_{shift}, the execution time of the new parallel multishift QR algorithm is modeled 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_{AED}, T_{sweep}, and T_{shift} are the runtimes for performing each phase once. For simplicity, it is assumed that n_{AED} and n_{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 $\sqrt{p_{\text{AED}}}\,n_b \ll n_{\text{AED}}$ 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 (for example, $C_1 = 24, C_2 = 384$). In practice $k_{\text{AED}}, k_{\text{sweep}}$, k_{shift} can differ widely 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 previous assumptions. Tiny terms, especially lower order terms with reasonably sized constants, are omitted. The derivation of these models can be found in the electronic appendix [Granat et al. 2014a]. By Henry et al. [2002], the execution time of the pipelined QR algorithm is

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

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

execution time of

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

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

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

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_{pipe} assumes level 3 BLAS 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

$$\begin{aligned} T_{\text{AED}}(n, n_{\text{AED}}, p) \approx & \left[\frac{30C_2^2 n}{C_1} + \frac{9n^2 n_b}{C_1^2\sqrt{p}} + \frac{9(C_1+6)n^3}{2C_1^3 p}\right]\gamma \\ & + \left(\frac{9C_2 n}{C_1 n_b}\log_2\frac{3n}{2C_1C_2} + \frac{3n}{2C_1}\log_2 p\right)\alpha \\ & + \left[\frac{9C_2 n}{C_1}\log_2\frac{3n}{2C_1C_2} + \frac{12C_2^2 n}{C_1 n_b} + \frac{3n^2(18 + C_1 + 51\log_2 p)}{16C_1^2\sqrt{p}}\right]\beta \end{aligned}$$

and

$$T_{\text{shift}}(n, n_{\text{shift}}, p) \approx \frac{10C_2^2 n}{C_1}\gamma + \frac{6C_2 n}{C_1 n_b}\log_2\frac{n}{C_1C_2}\alpha + \left(\frac{6C_2 n}{C_1}\log_2\frac{n}{C_1C_2} + \frac{8C_2^2 n}{C_1 n_b}\right)\beta,$$

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

$$\begin{aligned} T_{\text{sweep}}(n, n_{\text{shift}}, p) \approx & \frac{3n^3}{2p}\gamma + \frac{3n^2}{8\sqrt{p}\, n_b^2}(\log_2 p + 4)\alpha + \frac{3n^3}{8pn_b}\beta, \\ T_{\text{AED}}(n, n_{\text{AED}}, p) \approx & \left(184320n + \frac{n^2 n_b}{64\sqrt{p}} + \frac{5n^3}{256p}\right)\gamma \\ & + \left[\frac{144n}{n_b}(\log_2 n - 14) + \frac{n\log_2 p}{8}\right]\alpha \\ & + \left[144n\left(\log_2 n - 14 + \frac{512}{n_b}\right) + \frac{(51\log_2 p + 42)n^2}{3072\sqrt{p}}\right]\beta, \\ T_{\text{shift}}(n, n_{\text{shift}}, p) \approx & 61440n\gamma + \frac{96n}{n_b}(\log_2 n - 13)\alpha + 96n\left(\log_2 n - 13 + \frac{512}{n_b}\right)\beta. \end{aligned}$$

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

$$T_{\text{new}}(n, p) \approx \left(\frac{48n^3}{p} + 1.2 \times 10^7 n\right)\gamma + \frac{12n^2 \log_2 p}{\sqrt{p}\, n_b^2}\alpha + \left(\frac{12n^3}{pn_b} + \frac{17n^2 \log_2 p}{16\sqrt{p}}\right)\beta, \quad (5)$$

$$= \Theta\left(\frac{n^3}{p}\right)\gamma + \Theta\left(\frac{n^2 \log p}{\sqrt{p}\, n_b^2}\right)\alpha + \Theta\left(\frac{n^3}{pn_b}\right)\beta, \quad (6)$$

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) all have nearly 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 preceding model), 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 the user manual that accompanies the software [Granat et al. 2014b] for details). 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 `PILAENVX`. They are available via the function call `PILAENVX(ICTXT, ISPEC, ...)` with $12 \leq$ `ISPEC` ≤ 23. A complete list of these parameters is provided in Table II. 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 that performance is not overly sensitive to this choice. On the one hand, n_b should be large enough to ensure that the local computations achieve level 3 BLAS performance. On the other hand, it is advisable to avoid n_b being too large as this 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 model, we always assume $n_b \ll n/\sqrt{p}$ to avoid this 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 II will 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 series of test runs: To determine $n_{\min}$ and $p_{\min}$, it is advisable to use the typical sizes of the AED windows (see Table I) and run tests on different processor grids. The optimal values for both $n_{\min}$ and $p_{\min}$ may be chosen by examining the number of columns of H owned by each processor. `NIBBLE` should be tuned lastly, once all other parameters are fixed. Autotuning tools

Table II. List of Tunable Parameters

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

have been provided in the software for tuning these three parameters. We refer to the user manual [Granat et al. 2014b] 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, $\text{NIBBLE} = a \cdot n^b p^c$ 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* [HPC2N 2013b] and *Abisko* [HPC2N 2013a] hosted by the High Performance Computing Center North (HPC2N), and on *Bellatrix* [EPFL 2013] 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 III.

We compare the following implementations:

S-v180	Pipelined QR algorithm in ScaLAPACK v1.8.0
SISC	Previous implementation of parallel multishift QR algorithm with AED, as described in Granat et al. [2010]
NEW	New implementation of parallel multishift QR algorithm with AED, as described in this article

The implementation NEW improves upon 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.

Table III. Computational Environments

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

Table IV. Execution Time (in seconds) on *Akka* for `fullrand` Matrices

$p =$	$n = 4{,}000$			$n = 8{,}000$			$n = 16{,}000$			$n = 32{,}000$		
$p_r \times p_c$	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW
1×1	834	178	115	10,730	939	628						
2×2	317	87	56	2,780	533	292						
4×4	136	50	35	764	205	170	6,671	1,220	710			
6×6	112	50	43	576	142	116	3,508	754	446	∞	3,163	2,200
8×8	100	45	37	464	127	104	2,536	506	339	∞	2,979	1,470
10×10	97	50	36	417	159	119	2,142	457	320	∞	2,401	1,321

5.1. Random Matrices

First, we consider two types of random matrices: `fullrand` and `hessrand` [Granat et al. 2010].

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 [Granat et al. 2010]. 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 IV and V 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) to 10 hours for each individual problem, to avoid excessive use of the computational resources. An entry ∞ corresponds to an execution time that exceeds 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.

Table V. Execution Time (in seconds) on *Akka* for `hessrand` Matrices

$p =$	$n = 4{,}000$			$n = 8{,}000$			$n = 16{,}000$			$n = 32{,}000$		
$p_r \times p_c$	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW
1×1	685	317	14	6,981	2,050	78						
2×2	322	200	8	2,464	1,904	27						
4×4	163	112	36	1,066	679	71	8,653	2,439	65			
6×6	137	84	31	768	412	113	4,475	1,254	71	∞	373	252
8×8	121	68	25	634	321	107	3,613	719	71	∞	919	228
10×10	131	83	23	559	313	111	3,549	667	76	∞	943	267

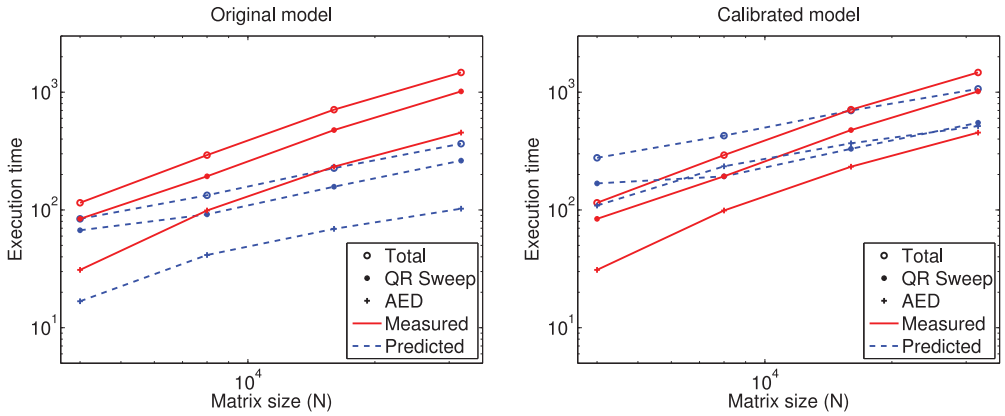

Fig. 5. Comparison between the measured execution times and the predicted times using (4) (`fullrand`, $n/\sqrt{p} = 4{,}000$). The original model (left) uses theoretical values of (α, β, γ) according to the hardware information, while the calibrated one (right) adjusts γ according to different computational intensities (levels 1, 2, and 3).

The convergence rates for `fullrand` are sufficiently regular to allow us to analyze the scalability of the parallel multishift QR algorithm. If we fix the memory load per core to $n/\sqrt{p} = 4{,}000$, the execution times in Table IV 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_{AED}, k_{sweep}, and k_{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 5(a) illustrates that the predicted execution times underestimate the measured ones (especially, for AED) mainly due to overoptimistic choices of the parameters (α, β, γ). If we assume that the program executes at 40% and 7.5% of the peak core performance for level 3 and levels 1–2 BLAS operations, respectively, the calibrated model fits the actual execution time quite well for large matrices (see Figure 5(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 `PDHSEQR` is between versions NEW and SISC in the AED process, it is not surprising to see the great convergence acceleration for `hessrand`, where AED dominates the calculation. We note in Table V that, in some instances, increasing the number of processes has little effect on the execution time for the new parallel multishift QR algorithm. This is mainly because the Schur decomposition of the AED window, which is the most expensive part of the

Table VI. Execution Time (in seconds) on *Abisko* for `fullrand` Matrices

$p =$	$n = 4{,}000$			$n = 8{,}000$			$n = 16{,}000$			$n = 32{,}000$		
$p_r \times p_c$	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW
1×1	764	139	97	7,373	694	471						
2×2	302	77	49	2,479	417	240						
4×4	91	40	31	781	156	132	5,507	1,040	548			
6×6	76	36	28	541	101	91	2,799	591	374	∞	2,641	1,706
8×8	52	34	29	276	88	98	1,881	383	294	∞	2,506	1,245
10×10	52	30	18	234	99	92	1,455	317	257	∞	1,909	1,118

Table VII. Execution Time (in seconds) on *Abisko* for `hessrand` Matrices

$p =$	$n = 4{,}000$			$n = 8{,}000$			$n = 16{,}000$			$n = 32{,}000$		
$p_r \times p_c$	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW
1×1	611	307	12	5,021	2,064	63						
2×2	302	202	7	1,966	1,458	29						
4×4	110	77	18	881	516	48	6,671	1,603	53			
6×6	96	61	21	578	339	70	4,006	1,034	52	∞	231	176
8×8	84	53	18	423	249	98	2,822	605	53	∞	737	166
10×10	73	58	17	360	214	79	2,456	553	56	∞	670	166

Table VIII. Execution Time (in seconds) on *Bellatrix* for `fullrand` Matrices

$p =$	$n = 4{,}000$			$n = 8{,}000$			$n = 16{,}000$			$n = 32{,}000$		
$p_r \times p_c$	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW
1×1	637	73	50	5,377	441	252						
2×2	192	24	21	1,594	137	91						
4×4	68	16	13	498	79	63	4,505	552	271			
6×6	47	12	11	294	44	39	1,886	247	165	∞	1,267	901
8×8	36	16	12	204	42	37	1,347	184	129	∞	1,362	714
10×10	37	14	9	181	39	40	961	140	110	∞	726	525

algorithm, is performed by a constant number of processors ($p_{\min} \cdot p_{\min} \leq p$) after data redistribution.

In Tables VI–IX we give the execution times obtained using the *Abisko* and *Bellatrix* systems. We observe that these times are qualitatively similar to those generated using *Akka* and we, therefore, restrict ourselves to just presenting the results from the *Akka* system in the rest of this section.

5.2. 100,000 x 100,000 Matrices

The modifications proposed and presented result in dramatic improvements when used with very large matrices and many processors. To demonstrate this, we present the execution times obtained for 100,000 $\times$ 100,000 matrices in Table X. 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 of 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 well controlled. In contrast to our earlier implementation, AED is no longer a bottleneck within the complete QR algorithm. As reported in Granat et al. [2010], it took 7 hours for our earlier implementation `PDHSEQR` to solve the 100,000$\times$100,000 `fullrand` problem with 32 $\times$ 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 Granat et al. [2010] (see Table XI). The execution times for the three solvers are listed in

Table IX. Execution Time (in seconds) on *Bellatrix* for `hessrand` Matrices

$p =$	$n = 4{,}000$			$n = 8{,}000$			$n = 16{,}000$			$n = 32{,}000$		
$p_r \times p_c$	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW
1×1	510	174	6	4,353	1,102	26						
2×2	205	66	4	1,590	530	11						
4×4	87	42	7	657	259	19	5,416	808	22			
6×6	57	23	9	428	134	37	3,054	380	22	∞	102	77
8×8	54	37	9	340	125	32	2,227	365	22	∞	342	72
10×10	46	22	8	280	92	27	1,846	214	24	∞	214	115

Table X. Execution Time (in seconds) on *Akka* of the New Parallel Multishift QR Algorithm (NEW) for 100,000 × 100,000 Matrices

	$p = 16 \times 16$		$p = 24 \times 24$	
	`fullrand`	`hessrand`	`fullrand`	`hessrand`
Balancing	876	–	881	–
Hess. reduction	10,084	–	6,441	–
QR algorithm	13,797	922	8,055	1,268
k_{AED}	35	19	31	19
k_{sweep}	5	0	6	0
#(shifts)/n	0.20	0	0.23	0
AED% in the QR alg.	48%	100%	43%	100%
	$p = 32 \times 32$		$p = 40 \times 40$	
	`fullrand`	`hessrand`	`fullrand`	`hessrand`
Balancing	886	–	912	–
Hess. reduction	3,868	–	2,751	–
QR algorithm	6,646	5,799	8,631	1,091
k_{AED}	27	18	23	18
k_{sweep}	13	0	12	0
#(shifts)/n	0.35	0	0.49	0
AED% in the QR alg.	39%	100%	54%	100%

Table XI. Benchmark Matrices

ID	Name	Dimension (n)	Type/Structure
1	BBMSN [Braman et al. 2002b]	n	$S_n = \begin{bmatrix} n & n-1 & n-2 & \cdots & 2 & 1 \\ 10^{-3} & 1 & 0 & & 0 & 0 \\ & 10^{-3} & 2 & & 0 & 0 \\ & & 10^{-3} & & 0 & 0 \\ & & & \ddots & n-2 & 0 \\ & & & & 10^{-3} & n-1 \end{bmatrix}$
2	AF23560 [Bai et al. 1997a]	23,560	Computational fluid dynamics
3	CRY10000 [Bai et al. 1997a]	10,000	Material science
4	OLM5000 [Bai et al. 1997a]	5,000	Computational fluid dynamics
5	DW8192 [Bai et al. 1997a]	8,192	Electrical engineering
6	MATRAN [Bai et al. 1997a]	n	Sparse random matrix
7	MATPDE [Bai et al. 1997a]	n	Partial differential equations
8	GRCAR [Bai et al. 1997a]	n	$G_n = \begin{bmatrix} 1 & 1 & 1 & 1 & & & \\ -1 & 1 & 1 & 1 & 1 & & \\ & -1 & 1 & 1 & 1 & 1 & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & & -1 & 1 & 1 & 1 & 1 \\ & & & & -1 & 1 & 1 & 1 \\ & & & & & -1 & 1 & 1 \\ & & & & & & -1 & 1 \end{bmatrix}$

Table XII. Execution Time (in seconds) for BBMSN

n	$p = p_r \times p_c$	S-v180	SISC	NEW
5,000	1×1	523	6	3
10,000	2×2	1,401	30	9
15,000	3×3	1,489	62	13

Table XIII. Execution Time (in seconds) for AF23560

$p = p_r \times p_c$	S-v180	SISC	NEW
4×4	15,486	2,651	1,375
6×6	9,088	1,279	826
8×8	6,808	793	563
10×10	5,694	662	475
12×12	5,422	578	404

Table XIV. Execution Time (in seconds) for CRY10000

$p = p_r \times p_c$	S-v180	SISC	NEW
1×1	6,394	1,331	1,251
2×2	2,123	580	495
4×4	979	236	209
6×6	731	161	132
8×8	545	128	95
10×10	496	144	90

Table XV. Execution Time (in seconds) for OLM5000

$p = p_r \times p_c$	S-v180	SISC	NEW
1×1	426	206	189
2×2	167	98	108
4×4	76	54	53
6×6	58	52	42
8×8	46	49	29
10×10	48	51	47

Table XVI. Execution Time (in seconds) for DW8192

$p = p_r \times p_c$	S-v180	SISC	NEW
1×1	10,307	1,329	1,297
2×2	1,187	572	524
4×4	635	225	236
6×6	357	152	138
8×8	302	129	104
10×10	275	121	93

Tables XII–XIX. The conclusions are similar to those we have made for random matrices: our earlier version of PDHSEQR easily outperforms the ScaLAPACK 1.8.0 routine PDLAHQR; the new PDHSEQR is usually even faster, especially for BBMSN and GRCAR.

In Granat et al. [2010], it was observed that the accuracy for AF23560 is not completely 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 are caused by an anomaly in PDLAHQR, which is now fixed by avoiding the use of PDLACONSB (see Section 2.3.2). As a result, the new PDHSEQR always produces $R_r \in [10^{-15}, 10^{-13}]$ 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 Granat et al. [2010]. 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 article is available in ScaLAPACK version 2.0.

Concerning future work, we believe we 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. The execution times spent on Hessenberg reduction and on QR iterations are almost comparable, so any further improvement in the iterative part will only have a limited impact on the

Table XVII. Execution Time (in seconds) for MATRAN

$p =$	$n = 5{,}000$			$n = 10{,}000$			$n = 15{,}000$		
$p_r \times p_c$	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW
1×1	1,617	332	218	∞	1,756	1,137			
2×2	579	152	122	4,495	931	471			
4×4	247	74	60	1,555	321	268	5,122	937	575
6×6	178	64	57	1,035	207	170	3,046	535	382
8×8	147	58	50	746	153	157	2,166	390	315
10×10	149	59	43	615	169	140	1,669	362	264

Table XVIII. Execution Time (in seconds) for MATPDE

$p =$	$n = 10{,}000$			$n = 14{,}400$			$n = 19{,}600$		
$p_r \times p_c$	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW
1×1	12,429	2,600	1,699						
2×2	3,531	1,081	966						
4×4	1,565	415	361	4,446	1,207	1,027	9,915	2,844	2,406
6×6	1,118	256	225	3,069	654	573	6,130	1,426	1,284
8×8	871	189	156	2,259	449	384	4,615	912	802
10×10	789	189	137	1,955	431	313	4,046	743	628
12×12	719	194	126	1,736	367	260	3,483	648	504

Table XIX. Execution Time (in seconds) for GRCAR

$p =$	$n = 6{,}000$			$n = 12{,}000$			$n = 18{,}000$		
$p_r \times p_c$	S-v180	SISC	NEW	S-v180	SISC	NEW	S-v180	SISC	NEW
1×1	2,738	1,340	69						
2×2	850	645	40	8,199	2,734	132			
4×4	363	258	226	2,499	1,336	114	8,171	4,037	182
6×6	244	190	173	1,471	849	112	4,385	2,172	187
8×8	217	150	145	1,107	515	142	3,342	1,345	175
10×10	207	161	126	923	538	338	2,675	1,104	276

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 [Karlsson and Kågström 2011; Kågström et al. 2008; Tomov et al. 2010], 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.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

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. Finally, we are grateful for several constructive comments from the referees and the Algorithm Editor, Tim Hopkins.

REFERENCES

E. Agullo, J. W. Demmel, J. J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.* 180, 1, 012037.

M. Ahues and F. Tisseur. 1997. *A New Deflation Criterion for the QR Algorithm*. LAPACK Working Note 122.

T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Kraemer, B. Lang, H. Lederer, and P. R. Willems. 2011. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Comput.* 37, 12, 783–794.

Z. Bai, D. Day, J. W. Demmel, and J. J. Dongarra. 1997a. *A Test Matrix Collection for Non-Hermitian Eigenvalue Problems (Release 1.0)*. Technical Report CS-97-355. Department of Computer Science, University of Tennessee. Also available online from http://math.nist.gov/MatrixMarket.

Z. Bai and J. W. Demmel. 1989. On a block implementation of Hessenberg multishift QR iteration. *Intl. J. High Speed Comput.* 1, 97–112.

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

Z. Bai, J. W. Demmel, J. J. Dongarra, A. Petitet, H. Robison, and K. Stanley. 1997b. The spectral decomposition of nonsymmetric matrices on distributed memory parallel computers. *SIAM J. Sci. Comput.* 18, 5, 1446–1461.

G. Ballard, J. W. Demmel, and I. Dumitriu. 2010. *Minimizing Communication for Eigenproblems and the Singular Value Decomposition*. Technical Report UCB/EECS-2010-136. EECS Department, University of California, Berkeley. Also as LAPACK Working Note 237.

G. Ballard, J. W. Demmel, O. Holtz, and O. Schwartz. 2011. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Anal. Appl.* 32, 3, 866–901.

P. Bientinesi, E. S. Quintana-Ortí, and R. A. van de Geijn. 2005. Representing linear algebra algorithms in code: The FLAME application program interfaces. *ACM Trans. Math. Software* 31, 1, 27–59.

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. 1997. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

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

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

R. Byers. 2007. *LAPACK 3.1 xHSEQR: Tuning and Implementation Notes on the Small Bulge Multi-Shift QR Algorithm with Aggressive Early Deflation*. LAPACK Working Note 187.

EPFL. 2013. Bellatrix. URL: http://hpc.epfl.ch/clusters/bellatrix/.

M. R. Fahey. 2003. Algorithm 826: A parallel eigenvalue routine for complex Hessenberg matrices. *ACM Trans. Math. Software* 29, 3, 326–336.

J. G. F. Francis. 1962. The QR transformation: A unitary analogue to the LR transformation—Part 2. *Comput. J.* 4, 4, 332–345.

G. H. Golub and C. F. Van Loan. 1996. *Matrix Computations* (3rd ed.). Johns Hopkins University Press, Baltimore, MD.

A. Grama, A. Gupta, G. Karypis, and V. Kumar. 2003. *Introduction to Parallel Computing* (2nd ed.). Addison-Wesley, Boston, MA.

R. Granat, B. Kågström, and D. Kressner. 2009. Parallel eigenvalue reordering in real Schur forms. *Concurrency and Computat.: Pract. Exper.* 21, 9, 1225–1250.

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

R. Granat, B. Kågström, D. Kressner, and M. Shao. 2014a. Algorithm 953: Parallel Library Software for the Multishift QR Algorithm with Aggressive Early Deflation—Electronic Appendix: Derivation of the Performance Model.

R. Granat, B. Kågström, D. Kressner, and M. Shao. 2014b. *PDHSEQR User's Guide*. Technical Report UMINF-14.24. Department of Computing Science, Umeå University. Available from http://www8.cs.umu.se/research/uminf/.

G. Henry and Robert A. van de Geijn. 1996. Parallelizing the QR algorithm for the unsymmetric algebraic eigenvalue problem: Myths and reality. *SIAM J. Sci. Comput.* 17, 4, 870–883.

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

M. Hoemmen. 2010. *Communication-Avoiding Krylov Subspace Methods*. Ph.D. Dissertation. University of California, Berkeley.

HPC2N. 2013a. Abisko. URL: http://www.hpc2n.umu.se/resources/abisko/.

HPC2N. 2013b. Akka. URL: http://www.hpc2n.umu.se/resources/akka/.

D. Irony, S. Toledo, and A. Tiskin. 2004. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distr. Comput.* 64, 9, 1017–1026.

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

B. Kågström, D. Kressner, and M. Shao. 2012. On aggressive early deflation in parallel variants of the QR algorithm. In *Applied Parallel and Scientific Computing (PARA 2010)* (Lecture Notes in Computer Science), Kristján Jónasson (Ed.), Vol. 7133. Springer-Verlag, Berlin, Germany, 1–10.

L. Karlsson and B. Kågström. 2011. Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures. *Parallel Comput.* 37, 12, 771–782.

R. S. Martin, G. Peters, and J. H. Wilkinson. 1970. Handbook series linear algebra: The QR algorithm for real Hessenberg matrices. *Numer. Math.* 14, 3, 219–231.

T. Schreiber, P. Otto, and F. Hofmann. 1994. A new efficient parallelization strategy for the QR algorithm. *Parallel Comput.* 20, 1, 63–75.

M. Shao. 2011. PDLAQR1*: An improved version of the ScaLAPACK routine* PDLAHQR. Technical Report UMINF-11.22. Department of Computing Science, Umeå University. Available from http://www8.cs.umu.se/research/uminf/.

B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. 1976. *Matrix Eigensystem Routines—EISPACK Guide* (2nd ed.). Lecture Notes in Computer Science, Vol. 6. Springer-Verlag, New York.

S. Tomov, R. Nath, and J. J. Dongarra. 2010. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Comput.* 36, 12, 645–654.

R. A. van de Geijn. 1988. Storage schemes for parallel eigenvalue algorithms. In *Numerical Linear Algebra Digital Signal Processing and Parallel Algorithms*, Gene H. Golub and Paul Van Dooren (Eds.). Springer-Verlag, 639–648.

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

R. A. van de Geijn and D. G. Hudson. 1989. An efficient parallel implementation of the nonsymmetric QR algorithm. In *4th Conference on Hypercube Concurrent Computers and Applications*. 697–700.

D. S. Watkins. 1994. Shifting strategies for the parallel QR algorithm. *SIAM J. Sci. Comput.* 15, 4, 953–958.

Received October 2013; revised June 2014; accepted October 2014