Parallel eigenvalue reordering in real Schur forms

LAPACK Working Note #192

R. Granat[1], B. Kågström[1], D. Kressner[1,2]

[1]Department of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden.
[2]Seminar für Angewandte Mathematik, ETH Zürich, Switzerland.
{granat,bokg}@cs.umu.se, kressner@sam.math.ethz.ch

Abstract. A parallel variant of the standard eigenvalue reordering method for the real Schur form is presented and discussed. The novel parallel algorithm adopts computational windows and delays multiple outside-window updates until each window has been completely reordered locally. By using multiple concurrent windows the parallel algorithm has a high level of concurrency, and most work is level 3 BLAS operations. The presented algorithm is also extended to the generalized real Schur form. Experimental results for ScaLAPACK-style Fortran 77 implementations on a Linux cluster confirm the efficiency and scalability of our algorithms in terms of more than 16 times of parallel speedup using 64 processor for large scale problems. Even on a single processor our implementation is demonstrated to perform significantly better compared to the state-of-the-art serial implementation.

Keywords: Parallel algorithms, eigenvalue problems, invariant subspaces, direct reordering, Sylvester matrix equations, condition number estimates

1 Introduction

The solution of large-scale matrix eigenvalue problems represents a frequent task in scientific computing. For example, the asymptotic behavior of a linear or linearized dynamical system is determined by the right-most eigenvalue of the system matrix. Despite the advance of iterative methods – such as Arnoldi and Jacobi-Davidson algorithms [3] – there are problems where a direct method – usually the QR algorithm [14] – is preferred, even in a large-scale setting. In the example quoted above, an iterative method may fail to detect the right-most eigenvalue and, in the worst case, misleadingly predict stability even though the system is unstable [29]. The QR algorithm avoids this problem, simply because *all* and not only selected eigenvalues are computed. Also, iterative methods are usually not well suited for simultaneously computing a large portion of eigenvalues along with the associated invariant subspace. For example, an invariant subspace belonging to typically half of the eigenvalues needs to be computed in problems arising from linear-quadratic optimal control [31], see also Section 7.

Parallelization of the QR algorithm is indispensable for large matrices. So far, only its two most important steps have been addressed in the literature: Hessenberg reduction and QR iterations, see [5, 10, 19], and the resulting software is implemented in ScaLAPACK [6, 32]. The (optional) third step of reordering the eigenvalues, needed for computing eigenvectors and invariant subspaces, has not undergone parallelization yet. The purpose of this paper is to fill this gap, aiming at a complete and highly performing parallel implementation of the QR algorithm.

1.1 Mathematical problem description

Given a general square matrix $A \in \mathbb{R}^{n\times n}$, computing the *Schur decomposition* (see, e.g., [14])

$$Q^T AQ = T \tag{1}$$

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

For any decomposition of (1) of the form

$$Q^T AQ = T \equiv \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix} \tag{2}$$

with $T_{11} \in \mathbb{R}^{p\times p}$ for some integer p and provided that $T(p+1, p)$ is zero, the first p columns of the matrix Q span an *invariant subspace* of A corresponding to the p eigenvalues of T_{11} (see, e.g, [13]). Invariant subspaces are important since they often eliminate the need of explicit eigenvector calculations in applications. However, notice that if $T(2, 1) = 0$, $v_1 = Q(1 : n, 1)$ is an (orthonormal) eigenvector of A corresponding to the eigenvalue $\lambda_1 = T(1, 1)$. If $T(2, 1) \neq 0$, complex arithmetics is required for extracting explicit eigenvectors to the eigenvalues in the leading 2×2 block in T.

Computing the *ordered* real Schur form (ORSF) of a general $n \times n$ matrix is vital in many applications, for example, in *stable-unstable separation* for solving Riccati matrix equations [34]. It can also be used for computing explicit eigenvectors of A by reordering each eigenvalue (block) of interest to the leading position in T and reading off the corresponding first (two) column(s) in Q, as illustrated above.

In [2], a direct algorithm for reordering adjacent eigenvalues in the real Schur form (2) is proposed. For the special case of a tiny matrix T where $p, n - p \in \{1, 2\}$, the method is as follows:

– Solve the continuous-time Sylvester (SYCT) equation

$$T_{11}X - XT_{22} = \gamma T_{12}, \tag{3}$$

where γ is a scaling factor to avoid overflow in the right hand side.

– Compute the QR factorization

$$\begin{bmatrix} -X \\ \gamma I \end{bmatrix} = QR \tag{4}$$

using Householder transformations (elementary reflectors).

– Apply Q in the similarity transformation of T:

$$\tilde{T} = Q^T T Q \tag{5}$$

– Standardize 2×2 block(s) if any exists.

In the method above, each swap is performed tentatively to guarantee backward stability by rejecting each swap that appears unstable with respect to a stability criterion. By applying a bubble-sort procedure based on the adjacent swap method, where all selected eigenvalues (usually pointed to by a *select* vector) are moved step by step towards the top-left corner of T by swapping adjacent blocks, ordered Schur forms can be computed, see Algorithm 1. We remark that for $n = 2$ the adjacent swaps are performed using Givens rotations.

Algorithm 1 Reordering a real Schur form (LAPACK's `DTRSEN`)

Input: A matrix $T \in \mathbb{R}^{n \times n}$ in real Schur form with m diagonal blocks of size 1×1 and 2×2, an orthogonal matrix $Q \in \mathbb{R}^{n \times n}$ and a subset of eigenvalues Λ_s, closed under complex conjugation.

Output: A matrix $\tilde{T} \in \mathbb{R}^{n \times n}$ in ordered real Schur form and an orthogonal matrix $\tilde{Q} \in \mathbb{R}^{n \times n}$ such that $\tilde{T} = \tilde{Q}^T T \tilde{Q}$. For some integer j, the set Λ_s is the union of eigenvalues belonging to the j upper-left-most diagonal blocks of $\tilde{T}$. The matrices T and Q are overwritten by $\tilde{T}$ and $Q\tilde{Q}$, respectively.

$j \leftarrow 0$
for $i \leftarrow 1, \ldots, m$ **do**
   if $\lambda(T_{ii}) \subset \Lambda_s$ **then**
      $j \leftarrow j + 1, \ \mathrm{select}(j) \leftarrow i$
   end if
end for
$\mathrm{top} \leftarrow 0$
for $l \leftarrow 1, \ldots, j$ **do**
   for $i \leftarrow \mathrm{select}(l), \mathrm{select}(l) - 1, \ldots, \mathrm{top} + 1$ **do**
      Swap $T_{i-1,i-1}$ and T_{ii} by an orthogonal similarity transformation and apply this transformation to the rest of the columns and rows of T, and the columns of Q.
   end for
   $\mathrm{top} \leftarrow \mathrm{top} + 1$
end for

This paper presents parallel algorithms for blocked variants [27] of the method in [2]. Furthermore, we apply our techniques to parallel eigenvalue reordering in the generalized real Schur form [21].

2 Serial blocked algorithms for eigenvalue reordering

The algorithm in [2] is implemented in LAPACK [1, 28] in the following hierarchy of routines: `DTRSEN`, which computes an ordered real Schur form and (optionally) associated condition numbers for the selected cluster of eigenvalues and the resulting associated invariant subspace; `DTREXC`, which moves a specified eigenvalue from a specific start position `IFST` to a specific end position `ILST`; `DLAEXC`, which performs direct swaps of adjacent eigenvalues and updates the matrices T and Q with respect to each swap (see Section 1).

An analogous software hierarchy exists for reordering in the generalized real Schur form [21, 25].

2.1 Working with computational windows and update regions

The LAPACK-style reordering [2] leads to matrix multiply updates in T and Q by small (2×2, 3×3, 4×4) orthogonal transformations. Essentially, the performance is in the range of level 1 and 2 BLAS [12, 7].

Following the ideas from [26, 27], we may *delay* and *accumulate* all updates from inside a *computational window* until the reordering is completed locally. Then the *update regions* of T (and the corresponding regions of Q) are updated, preferably in level 3 operations, see Figure 1. The window is then moved towards the top-left part of the matrix, such that the cluster now ends up in the bottom-right part of the current window, and the reordering process is repeated.

In general, this strategy leads to a significant performance gain by improving the memory access pattern and diminishing the number of cache misses. Notice that a similar technique was employed for the variants of the QR algorithm presented in [8, 9]. The idea of delaying updates is however not new (see, e.g., [11, 26] and the references therein).

The performance of the block algorithm is controlled by a few parameters, e.g., the size of the computational window n_{win}, and the number of eigenvalues to move inside each window, n_{eig}. A recommended choice is $n_{\mathrm{eig}} = n_{\mathrm{win}}/2$ [27]. Other parameters to tune are r_{mmult}, which defines the threshold for when to use the orthogonal transformations in their factorized form (Householder transformations and Givens rotations) or their accumulated form by matrix multiplication, and n_{slab}, which is used to divide the updates of the rows of T into blocks of columns for improved memory access pattern in case the orthogonal transformations are applied in their factorized form.

If only 1×1 blocks are reordered in the current position of the window and matrix multiplication is used, the orthogonal transformation matrix has the

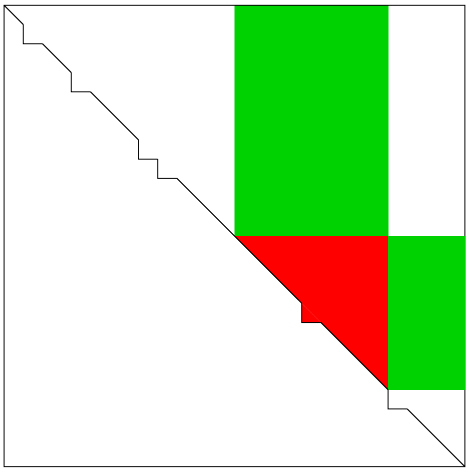


Fig. 1. Working with computational windows (red) and delaying the updates for level 3 BLAS operations on *update regions* (green) in the matrix T.

structure

$$U = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix} = \qquad , \tag{6}$$

i.e., the submatrices $U_{12} \in \mathbb{R}^{(n_{win}-k)\times(n_{win}-k)}$ and $U_{21} \in \mathbb{R}^{k\times k}$ are lower and upper triangular, respectively, and $k \leq n_{\text{eig}}$ is the number of eigenvalues moved from the bottom to top inside the window. This structure can be exploited by replacing a single call to GEMM by two calls to GEMM and TRMM which sometimes can lead to reduced execution times (see [27] for details). Here, GEMM is an acronym for the general matrix multiply and add operation $C \leftarrow \beta C + \alpha \cdot \text{op}(A) \cdot \text{op}(B)$, where $\text{op}(\cdot)$ denotes a matrix or its transpose. TRMM is an acronym for the triangular matrix multiply operation in level 3 BLAS [1].

We present the overall block reordering method in Algorithm 2.

3 Parallel blocked algorithms for eigenvalue reordering

Going in parallel, we adopt to the ScaLAPACK (see, e.g., [6, 32]) conventions of the parallel distributed memory (DM) environment, as follows:

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

Algorithm 2 Blocked algorithm for reordering a real Schur form

Input and Output: See Algorithm 1. Additional input: block parameters n_{eig} (max #eigenvalues in each window) and n_{win} (window size).

$i_{\mathrm{ord}} \leftarrow 0$ % $i_{\mathrm{ord}} = \#$*number of eigenvalues already in order*
while $i_{\mathrm{ord}} < \#\Lambda_s$ **do**
  % *Find first* $n_{ev} \leq n_{\mathrm{eig}}$ *disordered eigenvalues from top.*
  $n_{\mathrm{ev}} \leftarrow 0, \quad i_{\mathrm{hi}} \leftarrow i_{\mathrm{ord}} + 1$
  while $n_{\mathrm{ev}} \leq n_{\mathrm{eig}}$ **and** $i_{\mathrm{hi}} \leq n$ **do**
    if $T_{ii} \in \Lambda_s$ **then** $n_{\mathrm{ev}} \leftarrow n_{\mathrm{ev}} + 1$ **end if**
    $i_{\mathrm{hi}} \leftarrow i_{\mathrm{hi}} + 1$
  end while
  % *Reorder these eigenvalues window-by-window to top.*
  while $i_{\mathrm{hi}} > i_{\mathrm{ord}} + n_{\mathrm{ev}}$ **do**
    $i_{\mathrm{low}} \leftarrow \max\{i_{\mathrm{ord}} + 1, i_{\mathrm{hi}} - n_{\mathrm{win}} + 1\}$
    Apply Algorithm 1 to the active window $T(i_{\mathrm{low}} : i_{\mathrm{hi}}, i_{\mathrm{low}} : i_{\mathrm{hi}})$ in order to reorder the $k \leq n_{\mathrm{ev}}$ selected eigenvalues that reside in this window to top of window. Let the corresponding orthogonal transformation matrix be denoted by U.
    Update $T(i_{\mathrm{low}} : i_{\mathrm{hi}}, i_{\mathrm{hi}} + 1 : n) \leftarrow U^T T(i_{\mathrm{low}} : i_{\mathrm{hi}}, i_{\mathrm{hi}} + 1 : n)$.
    Update $T(1 : i_{\mathrm{low}} - 1, i_{\mathrm{low}} : i_{\mathrm{hi}}) \leftarrow T(1 : i_{\mathrm{low}} - 1, i_{\mathrm{low}} : i_{\mathrm{hi}})U$.
    Update $Q(1 : n, i_{\mathrm{low}} : i_{\mathrm{hi}}) \leftarrow Q(1 : n, i_{\mathrm{low}} : i_{\mathrm{hi}})U$.
    $i_{\mathrm{hi}} \leftarrow i_{\mathrm{low}} + k - 1$
  end while
  $i_{\mathrm{ord}} \leftarrow i_{\mathrm{ord}} + n_{\mathrm{ev}}$
end while

Since the matrices T and Q are square, we assume throughout this paper that $n_b = m_b$, i.e., the matrices are partitioned in square blocks. To simplify the reordering in the presence of 2×2 blocks, we also assume that $n_b \geq 3$ to avoid the situation of having two adjacent 2×2 blocks spanning over three different diagonal blocks in T. Moreover, we require T and Q to be aligned such that blocks T_{ij} and Q_{ij} are held by the same process, for all combinations of i and j, $1 \leq i, j \leq \lceil n/n_b \rceil$. Otherwise, shifting T and/or Q across the process mesh before (and optionally after) the reordering is necessary.

Below, we refer to an *active process* as a process that holds all or some part of a computational window in T.

3.1 The computational windows in the parallel environment

We restrict the size of the computational window to the block size used in the data layout. This means that a computational window can be in two states: either it is completely held by one single block or it is shared by four data layout blocks: two neighboring diagonal blocks, one subdiagonal block (in presence of a 2×2 block residing on the block borders) and one superdiagonal block.

3.2 Moving eigenvalues inside a data layout block

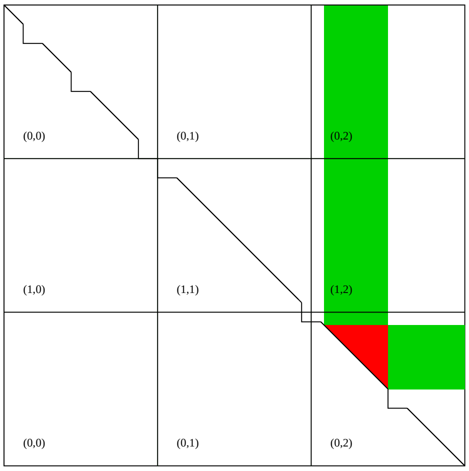
Depending on the values of n_b and n_{win}, each diagonal block of T is locally reordered by a certain number of partly overlapping positions of the window, moving $k \leq n_{\text{eig}}$ selected eigenvalues from the bottom towards the top of the block, see Figure 2. Before the window is moved to its next position inside the block, the resulting orthogonal transformations from the reordering in the current position are broadcasted in the process rows and columns corresponding to the current block row and column in T (and Q), see Figure 3. The subsequent updates are performed independently and in parallel. In principle, no other communication operations are required beside these broadcasts.

Given one active process working alone on one computational window the possible parallel speedup during the update phase is limited by $P_r + P_c$.

3.3 Moving eigenvalues across the process borders

When a computational window reaches the top-left border in the diagonal block, the current eigenvalue cluster must be reordered across the border into the next diagonal block of T. This forces the computational window to be shared by more than one data layout block and (optionally) more than one process. The restrictions in Section 3.1 make sure that a computational window cannot be shared by more than four data layout blocks and by at most four different processes which together form a submesh of maximum size 2×2, see Figure 4.

To be able to maximize the work performed inside each diagonal block before crossing the border and to minimize the required communication for the cross border reordering, it is beneficial to be able to control the size of the shared



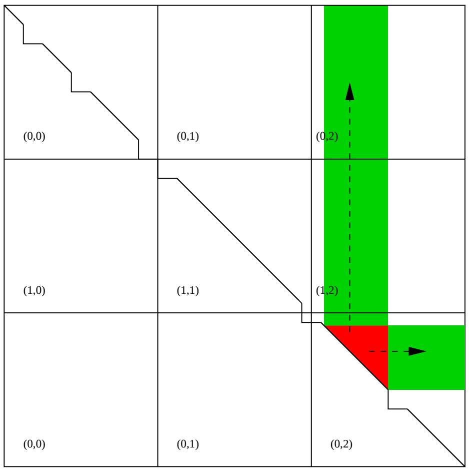


Fig. 2. The Schur form T distributed over a process mesh of 2×3 processors. The computational window (red) is completely local and the update regions (green) are shared by the corresponding process row and column.

Fig. 3. Broadcasting of the orthogonal transformations along the current process row and column of a 2×3 process mesh.

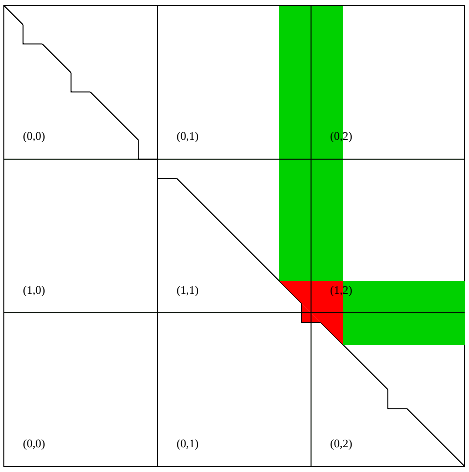


Fig. 4. The Schur form T distributed over a process mesh of 2×3 processors. The computational window (red) is shared by four distinct processes. The update regions (green) is shared by the corresponding two process rows and process columns.

windows by an optional parameter $n_{\text{crb}} \leq n_{\text{win}}$, that can be adjusted to match the properties of the target architecture.

The processes holding the different parts of the shared $n_{\text{crb}} \times n_{\text{crb}}$ window now cooperate to bring the window across the border, as follows:

- The on-diagonal active processes start by exchanging their parts of the window and receiving the off-diagonal parts from the off-diagonal parts from the two other processes. The cross border window causes updates in T and Q that span over parts of two block rows or columns. Therefore, the processes in the corresponding process rows and columns exchange blocks with their neighbors as preparation for the (level 3) updates to come, see Figure 5. The total amount of matrix elements from T and Q exchanged over the border in both directions is $n_{\text{crb}} \cdot (2n - n_{\text{crb}} - 1)$.
- The on-diagonal active processes compute the reordering for the current window. This requires some replication of the computations on both sides of the border. Since the total work is dominated by the off-diagonal updates, the overhead caused by the replicated work is negligible.
- Finally, the delayed and accumulated orthogonal updates are broadcasted along the corresponding process rows and columns, and used in level 3 updates, see Figure 6.

We remark that in principle the updates are independent and can be performed in parallel without any additional communication or redundant work. But if the orthogonal transformations are applied in their factorized form, each processor computing an update will also compute parts of T and/or Q that are supposed to be computed by another processor at the other side of the border. In case of more than one process in the corresponding mesh dimension, this causes duplicated work in the cross border updates, as well. Our remedy is to use a significantly lower value for r_{mmult} favoring matrix multiplication for cross border reordering.

In case matrix multiplication is used for the cross border reordering updates, we point out that the special structure (6) of the accumulated orthogonal transformations sometimes can be utilized. This happens when all the chosen eigenvalues have reached the other side of the border and the last eigenvalue to cross the border ends up in the position closest to the border. In such a case, the two pairs of GEMMs and TRMMs are split into two pairs of one GEMM and one TRMM, each pair being computed on each side of the border, independently and in parallel. Therefore, the size of the cross border window should be chosen as twice the number of eigenvalues to reorder across the border whenever possible.

3.4 Introducing multiple concurrent computational windows

By using multiple concurrent computational windows, we can work on at least $k_{\text{win}} \leq \min(P_r, P_c)$ adjacent windows at the same time, computing local reordering and broadcasting orthogonal transformations for updates in parallel. With $k = \min(P_r, P_c)$ and a square process mesh ($P_r = P_c$), the degree of concurrency becomes $P_r \cdot P_c$, see Figure 7.

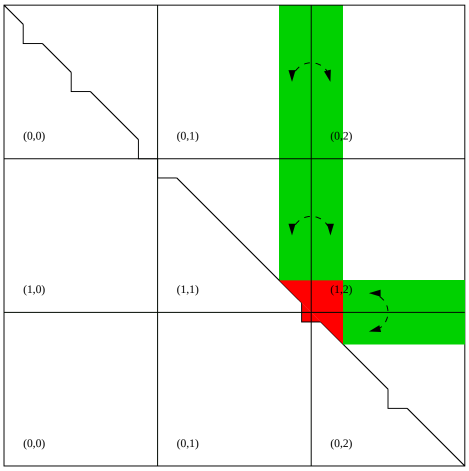


Fig. 5. Exchanges of data in adjacent process rows and columns for updates associated with cross border reordering.

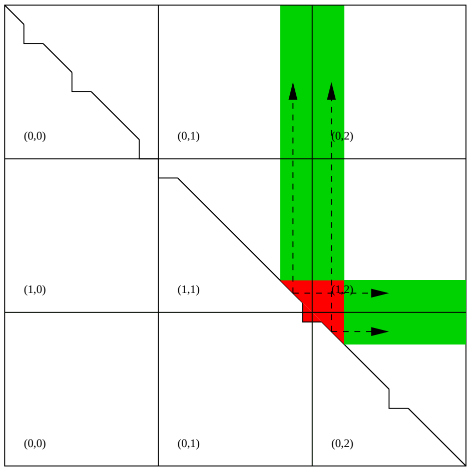


Fig. 6. Broadcasts of the computed orthogonal transformations in the current processor rows and columns for cross border reordering.

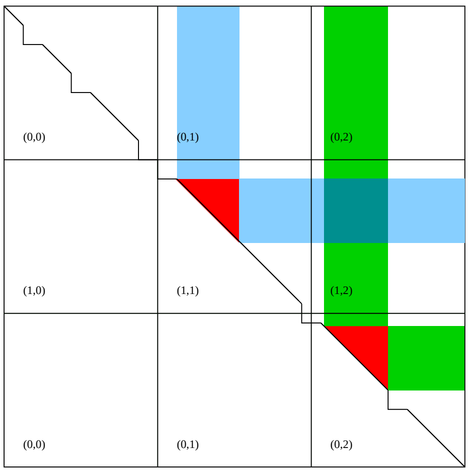


Fig. 7. Using multiple (here two) concurrent computational windows. The computational windows (red) are local but some parts of the update regions (green and blue) are shared.

When all k_{win} windows reach the process borders, they are moved into the next diagonal block as described in the previous section, but in two phases. Since each window requires cooperation between two adjacent process rows and columns, we number the windows by the order in which they appear on the block diagonal of T and start by moving all windows with an odd label across the border, directly followed by all windows with an even label. Care has to be taken to assure that no processor is involved in more than one cross border window at the same time. For example, if $k_{\text{win}} = \min(P_r, P_c) > 1$ is an odd number, the last labelled window will involve processors which are also involved in the first window. In such a case, the last window is reordered across the border after the second (even) phase has finished.

This two-phase approach gives an upper limit of the concurrency of the local reordering and data exchange phases of the cross border reordering as $k_{\text{win}}/2$, which is half of the concurrency of the pre-cross border part of the algorithm.

We present a high-level version of our parallel multi-window method in Algorithms 3–4. Notice that in the presence of 2×2 blocks in the Schur form, the computed indices in the algorithm are subject to slight adjustments (see Section 5).

4 Performance analysis

In this section, we analyze the parallel performance of Algorithms 3–4 and derive a model of the *parallel runtime* using p processes

$$T_p = T_a + T_c, \tag{7}$$

where T_a and T_c denote the *arithmetic* and *communication* (synchronization) runtime, respectively. We assume block cyclic data distribution of $T \in \mathbb{R}^{n \times n}$ over a square $\sqrt{p} \times \sqrt{p}$ process mesh using the square block factor n_b (see Section 3). We define t_a, t_s and t_w as the *arithmetic time* to perform a floating point operation (flop), the *start-up time* (or node latency) for sending a message in our parallel computer system, the *per-word transfer time* or *the inverse of the bandwidth* as the amount of time it takes to send one data word (e.g., a double precision number) through one link of the interconnection network. Usually, t_s and t_w are assumed to be constants while t_a is a function of the data locality. The communication cost model for a single point-to-point communication can be approximated by $t_{p2p} = t_s + t_w l$, where l denotes the message length in words, regardless of the number of links traversed [15]. However, for a *one-to-all broadcast* or its dual operation *all-to one reduction* in a certain scope (e.g., a process row or process column), we assume that such an operation is performed using a hypercube-based algorithm like *recursive doubling*, i.e., in $O(\log_2 p_\star)$ steps, where $p_\star$ is the number of processors in the actual scope.

Algorithms 1 and 2 have an arithmetic complexity of $T_s = O(kn^2)t_a$, where $1 \leq k < n$ is the number of selected eigenvalues to reorder to the top-left corner of the matrix T. The exact cost depends on the distribution of the eigenvalues

Algorithm 3 Parallel blocked algorithm for reordering a real Schur form (main part)

Input and Output: See Algorithm 2. Additional input: data layout block size, n_b, process mesh sizes, P_r and P_c, process mesh indices, *myrow* and *mycol*, maximum cross border window size, n_{wcb}, maximum number of concurrent windows, $k_{\text{win}} \leq min(P_r, P_c)$.

```
Let W = {The set of computational windows} = ∅
i_ord ← 0    % i_ord = #number of eigenvalues already in order
while i_ord < #Λ_s do
  for j ← 0, ..., k_win − 1 do
    s_win = (⌊i_ord/n_b⌋ + j) · n_b + 1
    Add a window to W for T(s_win : s_win + n_b − 1, s_win : s_win + n_b − 1)
  end for
  % Reorder each window to top-left corner of corresponding diagonal block.
  for each window w ∈ W in parallel do
    if (myrow, mycol) owns w then
      % Find first n_ev ≤ n_eig disordered eigenvalues from top of my diagonal block.
      n_ev ← 0,   i_hi ← max{i_ord + 1, s_win}
      while n_ev ≤ n_eig and i_hi ≤ n do
        if T_ii ∈ Λ_s then
          n_ev ← n_ev + 1
        end if
        i_hi ← i_hi + 1
      end while
    end if
    % Reorder these eigenvalues window-by-window to top of diagonal block.
    while i_hi > i_ord + n_ev do
      i_low ← max{i_ord + 1, max{i_hi − n_win + 1, s_win}}
      if (myrow, mycol) owns w then
        Apply Algorithm 1 to the active window T(i_low : i_hi, i_low : i_hi) in order to
        reorder the k ≤ n_ev selected eigenvalues that reside in this window to top
        of the window. Let the corresponding orthogonal transformation matrix
        be denoted by U.
        Broadcast U in process row myrow.
        Broadcast U in process column mycol.
      else if (myrow, mycol) needs U for updates then
        Receive U
      end if
      Update T(i_low : i_hi, i_hi + 1 : n) ← U^T T(i_low : i_hi, i_hi + 1 : n) in parallel.
      Update T(1 : i_low − 1, i_low : i_hi) ← T(1 : i_low − 1, i_low : i_hi)U in parallel.
      Update Q(1 : n, i_low : i_hi) ← Q(1 : n, i_low : i_hi)U in parallel.
      i_hi ← i_low + k − 1
    end while
  end for
  i_ord ← i_ord + n_ev[top-left window in W]
  % Reorder selected clusters across block (process) border.
  Apply Algorithm 4 to W in order to reorder each computational window across
  the next block (process) border
end while
```

Algorithm 4 Parallel blocked algorithm for reordering a real Schur form (cross border part)

Input and Output: See Algorithm 3. Additional input: set of computational windows W.

for each odd window $w \in W$ in parallel **do**
  Form 2×2 process submesh $G = \{(0,0),(0,1),(1,0),(1,1)\}$ corresponding to w
  $i_{\text{ihi}} \leftarrow \min\{i_{\text{ilo}} - 1, s_{\text{win}} + n_{\text{wcb}}/2\}$, $i_{\text{ilo}} \leftarrow \max\{i_{\text{ihi}} - n_{\text{wcb}} + 1, i_{\text{ord}}\}$
  if $(\mathit{myrow}, \mathit{mycol}) \in G$ **then**
    Exchange data in G to build w at $(0,0)$ and $(1,1)$
    if $(\mathit{myrow}, \mathit{mycol}) \in \{(0,0),(1,1)\}$ **then**
      Apply Algorithm 1 to w to compute U and k.
      Broadcast U in process row *myrow*.
      Broadcast U in process column *mycol*.
    end if
  end if
  if $\mathit{myrow} \in \text{prows}(G)$ or $\mathit{mycol} \in \text{pcols}(G)$ **then**
    Exchange local parts of $T(i_{\text{low}} : i_{\text{hi}}, i_{\text{hi}} + 1 : n)$, $T(1 : i_{\text{low}} - 1, i_{\text{low}} : i_{\text{hi}})$ and $Q(1 : n, i_{\text{low}} : i_{\text{hi}})$ for updates with neighboring processes in parallel.
    Receive U.
    Update $T(i_{\text{low}} : i_{\text{hi}}, i_{\text{hi}} + 1 : n) \leftarrow U^T T(i_{\text{low}} : i_{\text{hi}}, i_{\text{hi}} + 1 : n)$ in parallel.
    Update $T(1 : i_{\text{low}} - 1, i_{\text{low}} : i_{\text{hi}}) \leftarrow T(1 : i_{\text{low}} - 1, i_{\text{low}} : i_{\text{hi}})U$ in parallel.
    Update $Q(1 : n, i_{\text{low}} : i_{\text{hi}}) \leftarrow Q(1 : n, i_{\text{low}} : i_{\text{hi}})U$ in parallel.
  end if
end for
if $i_{\text{ilo}} = i_{\text{ord}}$ **then**
  $i_{\text{ord}} \leftarrow i_{\text{ord}} + k$
end if
for each even window $w \in W$ in parallel **do**
  % Similar algorithm as for the odd case above.
end for

over the diagonal of the Schur form. In the following, we assume $1 \ll k < n$ which implies that a majority of the updates are performed using matrix multiplication.

Given that the selected eigenvalues are uniformly distributed over the diagonal of T, the arithmetic cost of executing Algorithms 3–4 can be modelled as

$$T_a = \left(\frac{kn^2 - 3knn_{\mathrm{win}}}{p} + \frac{3knn_{\mathrm{win}}}{\sqrt{p}} \right) t_a, \tag{8}$$

where the first term is the cost of the GEMM updates, which is divided evenly between the p involved processors, and the second term describes the cost for computing the local and cross border reordering in the $n_{\mathrm{win}} \times n_{\mathrm{win}}$ computational windows, where the diagonal processors are working and the off-diagonal processors are idle waiting for the broadcasts to start.

In principle, bubble-sorting the eigenvalues in the unblocked Schur form can be seen as bubble-sorting the diagonal $n_b \times n_b$ blocks of the blocked Schur form in $O((n/nb)^2)$ steps. In each step, each local window is reordered to the top-left corner of the corresponding diagonal block and reordered across the border. By working with $\sqrt{p}$ concurrent windows in parallel (which is the upper limit for a square grid), the communication cost T_c can be modelled as

$$T_c = \frac{D_T^2}{\sqrt{p}} (t_{\mathrm{lbcast}} + t_{\mathrm{cwin}} + t_{\mathrm{cbcast}} + t_{\mathrm{rcexch}}), \tag{9}$$

where $D_T = \lceil n/n_b \rceil$ is the number of diagonal blocks in T, and t_{lbcast}, t_{cwin}, t_{cbcast}, and t_{rcexch} model the cost of the broadcasts local to the corresponding process row and column, the point-to-point communications associated with the construction of the cross border window, the cross border broadcasts in the two corresponding process rows and columns, and the data exchange between neighboring processes in the corresponding process rows and columns for the cross border updates, respectively.

The number of elements representing the factorized form of the orthogonal transformation can be approximated by n_{win}^2 and the broadcasts are initiated once for each time a window is moved to the next position in the corresponding block. Based on these observations, we model t_{lbcast} as

$$t_{\mathrm{lbcast}} = 2 \frac{n_b}{n_{\mathrm{win}}} (t_s + n_{win}^2 t_w) \log_2 \sqrt{p}. \tag{10}$$

By a similar reasoning and taking into account the lower degree of concurrency in the cross border operations, see Section 3.1, we model $t_{\mathrm{cwin}} + t_{\mathrm{cbcast}}$ by

$$t_{\mathrm{cwin}} + t_{\mathrm{cbcast}} = 4k_{\mathrm{cr}}(t_s + n_{crb}^2 t_w)(\log_2 \sqrt{p} + 1) \tag{11}$$

where k_{cr} is the number of passes necessary for bringing the whole locally collected eigenvalue cluster across the border ($k_{\mathrm{cr}} = 1$ if $n_{\mathrm{crb}} = n_{\mathrm{win}}$).

The exchanges of row and column elements in T and column elements in Q between neighboring processes suffer from a lower degree of concurrency. Moreover, we cannot guarantee that the send and receive may take place concurrently rather than in sequence. The average length of each matrix row or column to exchange is $n/2$ and about half of the cross border window is located on each side of the border. To sum up, we model t_{rcexch} as

$$t_{\text{rcexch}} = 12k_{\text{cr}}(t_s + \frac{n \cdot n_{\text{crb}}}{4\sqrt{p}} t_w). \tag{12}$$

Using the derived expressions and assuming $k_{\text{cr}} = 1$, i.e., $n_{\text{crb}} = n_{\text{win}}$, T_c can be boiled down to the approximation

$$\left(\left(\frac{2n^2}{n_b \cdot n_{\text{win}}} + \frac{4n^2}{n_b^2} + \frac{12n^2}{\log_2 \sqrt{p} \cdot n_b^2}\right) t_s + \left(\frac{2n^2 \cdot n_{\text{win}}}{n_b} + \frac{4n^2 \cdot n_{\text{win}}^2}{n_b^2} + \frac{3n^3 \cdot n_{\text{win}}}{\log_2 \sqrt{p} \cdot n_b^2}\right) t_w\right) \frac{\log_2 \sqrt{p} + 1}{\sqrt{p}}.$$

The dominating term is the last fraction of the part associated with t_w and comes from the data exchange associated with the updates (see Figure 5); it is of order $O(n^3/(n_b \cdot \sqrt{p}))$ assuming $n_{\text{win}} = O(n_b)$ and in the general case the communication cost in the algorithm will be dominated by the size of this contribution. The influence of this term is diminished by choosing $n_b \cdot \sqrt{p}$ as close to n as possible and thereby reducing the term closer to $O(n^2)$, which may be necessary when the arithmetic cost is not dominated by GEMM updates. For example, for $n = 1500$, $n_b = 180$ and $p = 64$, we have $n_b \cdot \sqrt{p} = 1440 \approx n$ (see also Section 6). In general and in practice, we will have $n_b \cdot \sqrt{p} = n/l$, where $l < n$ is the average number of row or column blocks of T distributed to each processor in the cyclic distribution. The scenario to strive for is $l << k$, where k is the number of selected eigenvalues. Then we perform significantly more arithmetics than communication which is a rule of thumb in all types of parallel computing. If this is possible depends on the problem, i.e, the number of selected eigenvalues and their distribution over the diagonal of T. Our derived model is compared with real measured data in Section 6.

5 Implementation issues

In this section, we address some important implementation issues.

5.1 Very few eigenvalues per diagonal block

In the extreme case, there will be only one eigenvalue per diagonal block to reorder. In such a situation, we would like to gather larger clusters of eigenvalues over the borders as quickly as possible to increase the serial performance of the local reordering of the diagonal blocks. However, this can be a bad idea since the scalability and the overall efficiency in hardware utilization will be poor when the number of concurrent windows and the concurrently working processors are kept low. Our parallel multi-window method is consequently self-tuning in the sense that its serial performance will vary with the number of selected eigenvalues and, in some sense, their distribution across the main block diagonal of T.

5.2 Splitting clusters in cross border reordering

Sometimes when we cross the process borders, not all the eigenvalues of the cluster can be moved across the border because of lack of reserved storage at the receiver, e.g., when there are other selected eigenvalues on the other side which occupy entries in T close to the border. Then the algorithm splits the cluster in two parts, roughly half on each side of the border, to keep the cross border region as small as necessary. In such a way, more work is performed with a higher rate of concurrency in the pre-cross border phase: the eigenvalues left behind are pushed closer to the border and reordered across the border in the next cross border sweep over the diagonal blocks.

5.3 Size of cross border window and shared 2×2 blocks

To simplify the cross border reordering in the presence of any 2×2 block being shared on the border, we keep the top-left part of the cross border window at a minimum dimension 2. This ensures that the block does not stay in the same position causing an infinite loop. For a similar reason we keep the bottom-right part of the window at a minimum dimension 3 if a non-selected eigenvalue is between the border and a selected 2×2 block.

In our implementation, the size of the cross border window is determined by the parameter n_{ceig} which controls the number of eigenvalues that cross the border. In practice, $n_{\mathrm{crb}} \approx 2n_{\mathrm{ceig}}$ except for the case when it is not possible to bring all eigenvalues in the current cluster across the border for reasons discussed above.

5.4 Detection of reordering failures and special cases

Due to the distributed nature of the parallel algorithm, failures in reordering within any computational window and special cases, like when no selected eigenvalue was found or moved in a certain computational window, must be detected at runtime and signaled to all processors in the affected scope (e.g., the corresponding processor row(s), processor column(s) or the whole mesh) at specific synchronization points. In the current implementation, all processors are synchronized in this way right before and right after each computational window is moved across the next block border.

5.5 Organization of communications and updates

In a practical implementation of Algorithms 3–4, the broadcasts of the orthogonal transformations, the data exchanges for cross border row and column updates and the associated updates of the matrices T and Q should be organized to minimize any risk for sequential bottlenecks. For example, all broadcasts in the row direction should be started before any column oriented broadcast starts to ensure that no pair of broadcasts have intersecting scopes (see Figure 7). In practice, this approach also encourages an implementation that performs all row oriented operations, computations and communications, before any column oriented operations take place. For such a variant of the algorithm, all conclusions from Section 4 are still valid. This technique also paves the way for a greater chance of overlapping communications with computations, possibly leading to better parallel performance.

5.6 Condition estimation of invariant subspaces

Following the methodology of LAPACK, our implementation also computes condition numbers for the invariant (deflating) subspaces and the selected cluster of eigenvalues using the recently developed software package SCASY [16, 17, 30] and adopting a well-known matrix norm estimation technique [18, 20, 23] in combination with parallel high performance software for solving different transpose variants of the triangular (generalized) Sylvester equations.

6 Experimental results

In this section, we demonstrate the performance of a ScaLAPACK-style parallel Fortran 77 implementation of Algorithms 3–4 called `PBDTRSEN`. All experiments were carried out in double precision real arithmetics ($\epsilon_{\text{mach}} \approx 2.2 \times 10^{-16}$).

Our target parallel platform is the Linux Cluster *seth* which consists of 120 dual AMD Athlon MP2000+ nodes (1.667MHz, 384KB L1 cache), where most nodes have 1GB memory and a small number of nodes have 2GB memory. The cluster is connected with a Wolfkit3 SCI high speed interconnect having a peak bandwidth of 667 MB/sec. The network connects the nodes in a 3-dimensional torus organized as a $6\times4\times5$ grid, where each link is "one-way" directed. In total, the system has a theoretical peak performance of 800 Gflops/sec. Moreover, *seth* is a platform which really puts any parallel algorithm to a tough test regarding its utilization of the narrow memory hierarchy of the dual nodes.

All subroutines and programs were compiled using the Portland Group's `pgf90 6.0-5` compiler using the recommended compiler flags `-O2 -tp athlonxp -fast` and the following software libraries: `ScaMPI (MPICH 1.2)`, `LAPACK 3.0`, `ATLAS 3.5.9`, `ScaLAPACK / PBLAS 1.7.0`, `BLACS 1.1`, `RECSY 0.01alpha` and `SLICOT 4.0`.

All presented timings in this section are in seconds. Parallel speedup, S_p, is computed as

$$S_p = T_{p_{\min}}/T_p, \tag{13}$$

where T_p is the parallel execution time using p processors and $p_{\min}$ is the smallest number of processors utilized for the current problem size.

Some initial tests were performed by reordering 50% of the eigenvalues uniformly distributed over the diagonal of 1500×1500 random matrices[1] already in Schur form and updating the corresponding Schur vectors in Q. The purpose

[1] The strictly upper part of T is a random matrix, but we construct T such that 50% of its eigenvalues are in complex conjugate pairs.

of these tests was to find close to optimal configuration parameters for the parallel algorithm executed with one computational window on one processor. By testing all feasible combinations of n_b, $n_{\rm win}$ and $n_{\rm eig} = n_{\rm ceig}$ within the integer search space $\{10, 20, \ldots, 200\}$ for $r_{\rm mmult} \in \{5, 10, \ldots, 100\}$ and $n_{\rm slab} = 32$, we found that $n_b = 180$, $n_{\rm win} = 60$ and $n_{\rm eig} = n_{\rm ceig} = 30$ and $r_{\rm mmult} = 40$ are optimal with runtime 5.66 seconds. This is a bit slower than the result in [27] (4.74 seconds) but still a great improvement over the current LAPACK algorithm (`DTRSEN`) which takes over 20 seconds! The difference may be partly explained by the additional data copying that is performed in the parallel algorithm during the cross border phase.

We display uniprocessor performance results for `DTRSEN` and `PBDTRSEN` in Figure 8, using the optimal configuration parameters listed above. The speedup is remarkable for large problems; $n = 5700$ shows a speedup of 14.0 for the parallel block algorithm executed on one processor compared to the standard algorithm, which is mainly caused by an improved memory reference pattern due to the rich usage of high performance level 3 BLAS.

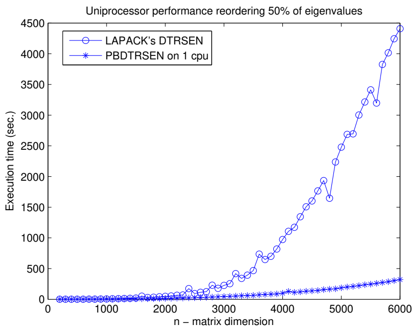


Fig. 8. Uniprocessor performance results for standard LAPACK algorithm `DTRSEN` and parallel block algorithm `PBDTRSEN` reordering 50% of uniformly distributed eigenvalues of random matrices.

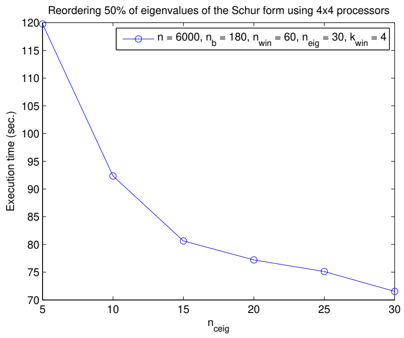


Fig. 9. Parallel execution time for 6000×6000 matrices using 4 concurrent computational windows on a 4×4 processor mesh reordering 50% of uniformly distributed eigenvalues of random matrices.

Next, using the close to optimal parameter settings, we made experiments on a 4×4 processor mesh using 4 computational windows on 6000×6000 matrices of the same form as above (which puts the same memory load per processor as one 1500×1500 matrix does on one processor) to find a close to optimal value of $n_{\rm ceig}$ in the integer search space $\{5, 10, \ldots, 30\}$. In this case, we ended up with $n_{\rm ceig}^{\rm opt} = 30$ with a corresponding execution time of 71.53 seconds, see Figure 9. It turned out that the execution time increased with a decreasing $n_{\rm ceig}$. For instance, the same problem took 119.77 seconds using $n_{\rm ceig} = 5$. With our analysis in mind, this is not surprising since using $n_{\rm eig} = n_{\rm ceig}$ brings each subcluster of eigenvalues

Table 1. Performance of `PBDTRSEN` on *seth*.

			Random		*Bottom*			*Random*		*Bottom*	
Sel.	$P_r \times P_c$	n	time	S_p	time	S_p	n	time	S_p	time	S_p
5%	1×1	1500	1.98	1.00	2.16	1.00	4500	42.7	1.00	42.4	1.00
	2×2	1500	0.90	2.20	1.02	2.12	4500	15.9	2.69	17.6	2.41
	3×3	1500	1.01	1.96	1.04	2.08	4500	10.8	3.95	12.4	3.42
	4×4	1500	0.80	2.48	0.92	2.35	4500	6.65	6.42	7.74	5.48
	5×5	1500	0.34	5.82	0.76	2.84	4500	4.70	9.09	6.25	6.78
	6×6	1500	0.32	6.19	0.35	6.17	4500	4.26	10.0	4.51	9.40
	7×7	1500	0.32	6.19	0.32	6.75	4500	3.48	12.3	3.50	12.1
	8×8	1500	0.24	8.25	0.27	8.00	4500	3.13	13.6	3.50	12.1
35%	1×1	1500	5.78	1.00	9.87	1.00	4500	127	1.00	217	1.00
	2×2	1500	2.96	1.95	5.58	1.77	4500	55.3	2.30	96.7	2.24
	3×3	1500	2.05	2.82	4.03	2.45	4500	40.4	3.14	67.9	3.20
	4×4	1500	1.74	3.32	2.72	3.63	4500	25.5	4.98	43.2	5.02
	5×5	1500	1.51	3.83	2.28	4.33	4500	22.4	5.67	39.0	5.56
	6×6	1500	1.17	4.94	1.82	5.42	4500	16.7	7.60	26.4	8.22
	7×7	1500	1.13	5.12	1.49	6.62	4500	15.8	8.04	21.6	10.0
	8×8	1500	1.02	5.67	1.12	8.81	4500	11.8	10.8	18.7	11.6
50%	1×1	1500	5.66	1.00	12.6	1.00	4500	140	1.00	263	1.00
	2×2	1500	3.50	1.62	6.83	1.84	4500	62.7	2.23	121	2.17
	3×3	1500	1.89	2.99	4.97	2.53	4500	36.1	3.88	84.3	3.12
	4×4	1500	1.87	3.03	4.48	2.81	4500	29.2	4.79	61.5	4.28
	5×5	1500	1.54	3.68	3.20	3.93	4500	21.6	6.48	49.6	5.30
	6×6	1500	1.22	4.64	2.03	6.21	4500	18.8	7.45	34.2	7.69
	7×7	1500	1.30	4.35	1.90	6.63	4500	16.5	8.48	30.3	8.68
	8×8	1500	1.11	5.10	1.76	7.16	4500	12.2	11.5	24.2	10.9
5%	1×1	3000	12.8	1.00	19.8	1.00	6000	97.6	1.00	109	1.00
	2×2	3000	4.93	2.60	6.63	2.99	6000	34.3	2.85	39.4	2.77
	3×3	3000	3.69	3.47	4.00	4.95	6000	23.8	4.10	27.4	3.98
	4×4	3000	3.01	4.26	2.90	6.83	6000	14.0	6.97	16.3	6.69
	5×5	3000	2.97	4.31	2.54	7.80	6000	11.2	8.71	14.4	7.57
	6×6	3000	1.72	7.44	2.12	9.34	6000	8.60	11.3	8.94	12.2
	7×7	3000	1.82	7.03	2.33	8.50	6000	6.20	15.8	7.52	14.5
	8×8	3000	1.45	8.83	1.59	12.5	6000	6.30	15.5	6.58	16.6
35%	1×1	3000	40.1	1.00	72.4	1.00	6000	304	1.00	509	1.00
	2×2	3000	19.0	2.11	38.0	1.91	6000	123	2.47	220	2.31
	3×3	3000	13.8	2.91	23.2	3.12	6000	89.9	3.38	152	3.35
	4×4	3000	8.83	4.54	14.2	5.10	6000	58.8	5.17	98.4	5.17
	5×5	3000	7.81	5.13	14.8	4.89	6000	51.6	5.89	81.8	6.22
	6×6	3000	6.90	5.81	9.26	7.82	6000	37.4	8.13	57.2	8.90
	7×7	3000	5.81	6.90	9.07	7.98	6000	31.7	9.59	49.0	10.4
	8×8	3000	5.30	7.57	7.17	10.1	6000	25.2	12.1	38.7	13.2
50%	1×1	3000	50.2	1.00	91.4	1.00	6000	324	1.00	623	1.00
	2×2	3000	24.2	2.07	44.5	2.05	6000	133	2.44	275	2.27
	3×3	3000	18.9	2.66	28.1	3.25	6000	94.5	3.43	197	3.16
	4×4	3000	11.6	4.33	18.2	5.02	6000	71.5	4.53	135	4.61
	5×5	3000	9.57	5.25	17.2	5.31	6000	50.2	6.45	111	5.61
	6×6	3000	7.17	7.00	12.6	7.25	6000	36.4	8.90	71.5	8.71
	7×7	3000	6.89	7.29	11.4	8.02	6000	33.4	9.70	61.8	10.1
	8×8	3000	5.27	9.53	9.81	9.32	6000	26.4	12.3	50.1	12.4

across the process border in one single step, thereby minimizing the number of messages sent across the border for each subcluster, i.e., the influence on the parallel runtime caused by the node latency is minimized.

In Table 1, we present representative performance results for `PBDTRSEN` for different variations of the number of selected eigenvalues, their distribution over the diagonal of the Schur form and the number of utilized processors. All presented results are obtained executing the parallel reordering algorithm using the close to optimal parameters settings using as many computational windows as possible ($k_{\text{win}} = \sqrt{p}$). For `PBDTRSEN`, the parallel speedup goes up to 16.6 for $n = 6000$ using a 8×8 processor mesh when selecting 5% of the eigenvalues from the lower part of T; in Table 1, *Random* and *Bottom* refer to the parts of T where the selected eigenvalues reside before the reordering starts.

Some remarks regarding the results in Table 1 are in order.

– In general, the parallel algorithm scales better (delivers a higher parallel speedup) for a smaller value of k, the number of chosen eigenvalues for a given problem size. The main reason is that less computational work per processor in general leads to less efficient usage of processor resources, i.e., a lower Mflops-rate, which in turn makes the communication overhead smaller in relation to arithmetics. We also see improved processor utilization when increasing the number of selected eigenvalues to reorder from 5% to 50%. The amount of work is increased by a factor 10 but the uniprocessor execution times increases roughly by a factor of 6 ($n = 3000$).
– Since only the processor rows and columns corresponding to the selected groups of four adjacent processors can be efficiently utilized during the cross border phase (see the formation of the processor groups in Algorithm 4), the performance gain going from a 2×2 to a 3×3 processor mesh is sometimes bad (even negative, see the problem $n = 1500$ and 5% selected eigenvalues). In principle, some performance degradation will occur when going from $2q \times 2q$ to $(2q + 1) \times (2q + 1)$ processors, for $q \geq 1$, since the level of concurrency in the cross border phase will not increase. This effect is not that visible for larger meshes since the relative amount of possibly idle processors in the cross border part decreases with an increasing processor mesh.

To confirm the validity of the derived performance model (see Section 4), we compare the experimental parallel speedup with parallel speedup computed from combining Equations (7), (8) and (9). The results are displayed in Figure 10. The machine specific constants t_a, t_s and t_w are estimated as follows:

– Since the *exact* number of flops needed for computing a reordered Schur form is not known a priori, we approximate t_a for $p = 1$ by $\tilde{t}_a^{(1)}(n, k)$ which is computed from Equation (8) by replacing T_a by T_1, the serial runtime of the parallel algorithm for one processor only. For $p > 1$, we model t_a by

$$\tilde{t}_a^{(p)}(n, k) = \alpha_0 + \alpha_1 n + \alpha_2 k + \alpha_3 \sqrt{p}, \tag{14}$$

where $\alpha_i \in \mathbb{R},\ i = 0, \ldots, 3$. The model captures that the processor speed is expected to be proportional to the matrix data load at each processor.

Since the amount of matrix data per processor and the number of selected eigenvalues per on-diagonal processor are kept fixed going from a problem instance (n, k, p) to $(2n, 2k, 4p)$, we assume

$$\tilde{t}_a^{(p)}(n,k) = \tilde{t}_a^{(4p)}(2n,2k). \tag{15}$$

From this assumption and the available data for $\tilde{t}_a^{(1)}$ derived from Table 1, we compute the α_i-parameters from a least squares fit, see Table 2. With these parameters and fixed values on p and/or n, the model predicts that $\tilde{t}_a^{(p)}(n,k)$ decreases for an increasing value of k. Moreover, for a fixed value on k, $\tilde{t}_a^{(p)}(n,k)$ increases with p and/or n. For such cases, the decrease of the modelled processor performance is marginal, except for large values on p and/or n. With an increasing size of the processor mesh $\sqrt{p} \times \sqrt{p}$ and fixed values on n and k, it is expected that individual processors perform less efficient due to less local arithmetic work.

– The interconnection network parameters t_s and t_w are estimated by performing MPI-based *ping-pong* communication in the network, which also includes solving an overdetermined linear system based on experimental data, see Table 2. In Figure 10, we allow t_w to vary by $\omega > 1$ which models unavoidable network sharing, overhead from the BLACS layer and the potentially (much) lower memory bandwidth inside the dual nodes. For $\omega = 1$, t_w represents the inverse of the practical peak bandwidth for double precision data in the network of *seth*.

Table 2. Experimentally determined machine parameters for *seth*.

Parameter	Value
α_0	6.73×10^{-9}
α_1	6.34×10^{-13}
α_2	-3.10×10^{-12}
α_3	3.08×10^{-10}
t_s	3.7×10^{-6}
t_w	$\omega \cdot 1.1 \times 10^{-8}$

The comparison presented in Figure 10 shows that the performance model is not perfect, but is able to predict a parallel speedup in the same range as the actually observed results.

We close this section by remarking that the accuracy of the parallel reordering algorithm is similar to the results presented in [2, 26, 27] since it is essentially numerically equivalent to those algorithms. Some additional accuracy results are also presented below in Section 7.

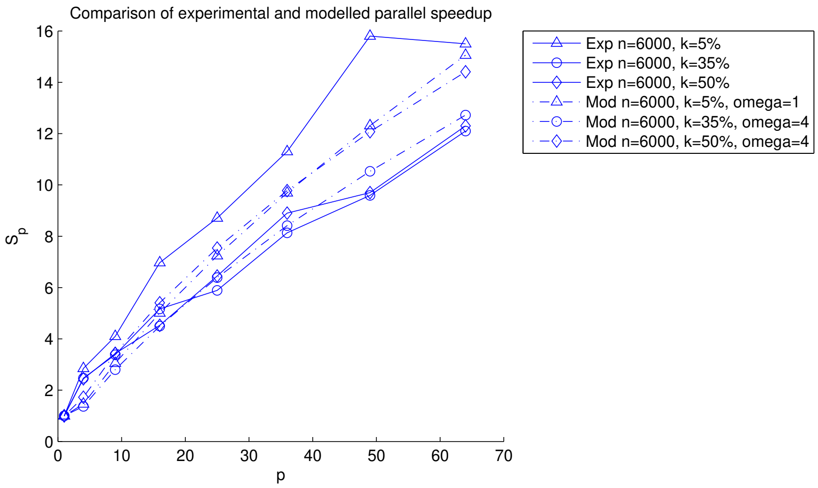


Fig. 10. Comparison of modelled and experimental parallel speedup for problems with *Random* distribution of the selected eigenvalues.

7 Application example: parallel computation of stable invariant subspaces of Hamiltonian matrices

As application example, we consider parallel computation of *c-stable* invariant subspaces corresponding to the eigenvalues $\{\lambda : \mathrm{Re}(\lambda) < 0\}$ and condition estimation of the selected cluster of eigenvalues and invariant subspaces of the Hamiltonian matrix

$$H = \begin{bmatrix} A & bb^T \\ cc^T & -A^T \end{bmatrix},$$

where $A \in \mathbb{R}^{\frac{n}{2} \times \frac{n}{2}}$ is a random diagonal matrix and $b, c \in \mathbb{R}^{\frac{n}{2} \times 1}$ are random vectors with real entries of uniform or normal distribution. For Hamiltonian matrices, $m = n/2 - k_{imag}$ of the eigenvalues are c-stable, where k_{imag} is the number of eigenvalues that lie strictly on the imaginary axis. Solving this Hamiltonian eigenvalue problem for the stable invariant subspaces can be very hard since the eigenvalues tend to cluster closely around the imaginary axis, especially when n gets large [26], leading to a very ill-conditioned separation problem.

The stable subspace computation includes two major steps and one optional third step:

1. Compute the real Schur form $T = Q^T H Q$ of H.
2. Reorder the m stable eigenvalues to the top left corner of T, i.e., compute the ordered Schur decomposition $\tilde{T} = \tilde{Q}^T H \tilde{Q}$, such that the m first columns of $\tilde{Q}$ span the stable invariant subspace of H; the m computed stable eigenvalues may be distributed over the diagonal of T in any order before reordering starts, see below.

3. Compute condition estimates for (a) the selected cluster of eigenvalues and (b) the invariant subspaces, see below.

For the first step, we utilize the parallel Hessenberg reduction routine `PDGEHRD` and a slightly modified[2] version of the existing parallel QR algorithm `PDLAHQR` from ScaLAPACK [6]. For the second step, we use the parallel algorithm described in this paper for reordering the eigenvalues. For the last step we utilize the condition estimation functionality and the corresponding parallel Sylvester-type matrix equation solvers provided by SCASY [16, 17, 30].

We present experimental results in Tables 3 and 4, where the following performance, output and accuracy quantities are presented:

- In Table 3, we present the parallel runtime measures t_1, t_2, t_{3a}, t_{3b}, t_{tot} and S_p corresponding to the individual steps in the stable subspace computation described above, the total execution time and the corresponding parallel speedup.
- In Table 4, we present the outputs m, s and *sep* corresponding to the dimension of the invariant subspace and the reciprocal condition numbers of the selected cluster of eigenvalues and the stable invariant subspace, respectively.
- In Table 4, we also present the accuracy measures $R_{\mathrm{o}1}$, $R_{\mathrm{r}1}$, $R_{\mathrm{o}2}$, $R_{\mathrm{r}2}$ and R_{eig} corresponding to the orthogonality check

$$R_{\mathrm{o}\star} = \max(\|Q_\star^T Q_\star - I\|_F, \|Q_\star Q_\star^T - I\|_F)/(\epsilon_{\mathrm{mach}} \cdot n),$$

the relative residual

$$R_{\mathrm{r}\star} = \max(\|T_\star - Q_\star^T A Q_\star\|_F, \|A - Q_\star T_\star Q_\star^T\|_F)/\|A\|_F,$$

and the relative change in the eigenvalues

$$R_{\mathrm{eig}} = \max_{i=1,n} \frac{|\lambda_k - \tilde{\lambda}_k|}{|\lambda_k|}.$$

Here, $\star = 1$ corresponds to the unordered real Schur form and $\star = 2$ represents the ordered Schur form corresponding to the stable invariant subspace.

The condition numbers s and *sep* are computed as follows:

- Solve the Sylvester equation

$$\tilde{T}_{11} X - X \tilde{T}_{22} = -\gamma \tilde{T}_{12}$$

and check norm of X in parallel and set $s = 1/\sqrt{(1 + \|X\|_F)}$ [4].
- Compute estimate of $\mathrm{sep}^{-1}(\tilde{T}_{11}, \tilde{T}_{22})$ in parallel using a well-known matrix norm estimation technique [18, 20, 23] and compute $sep = 1/\mathrm{sep}^{-1}$ taking care of any risk for numerical overflow.

[2] `PDLAHQR` often delivers a Schur form T with some 2×2 blocks corresponding to two real eigenvalues. Our modification consists of a postprocessing step where all such 'false' 2×2 blocks are removed via a novel implementation of `PDROT` which applies a Givens rotation to a distributed matrix.

s signals ill-conditioning if the norm of the spectral projector X on the invariant subspace associated with $\tilde{T}_{11}$ becomes large, i.e., s becomes small. A large estimate sep^{-1} signals that the invariant subspace associated with $\tilde{T}_{11}$ is ill-conditioned since the separation between the eigenvalues in $\tilde{T}_{11}$ and $\tilde{T}_{22}$ (and the exact value of $\mathrm{sep}(\tilde{T}_{11}, \tilde{T}_{22})$) is small, provided that the estimator computes a reliable estimate. Hence, a small value of *sep* signals an ill-conditioned invariant subspace (see, e.g., [33] and the references therein).

The results in Tables 3 and 4 illustrate the potential of the parallel reordering method in real applications by delivering good serial and parallel performance. Also, the eigenvalues are reordered with perfect accuracy, even though the corresponding invariant subspace may be very ill-conditioned. This ill-conditioning is likely an effect of the clustering of the eigenvalues very close around the imaginary axis, which also, because of round-off errors, affects the ability of the QR algorithm to deliver a correct number of stable eigenvalues, which determines the dimension (m) of the invariant subspace computed by the reordering routine.

We end this section by remarking that the condition number computations also scale well even though they are applied to relatively small problem sizes ($n/2 = 1500,\ 3000$) in this case.

Table 3. Experimental parallel performance results from computing stable invariant subspaces of Hamiltonian matrices on *seth*.

		Timings					
n	$P_r \times P_c$	t_1	t_2	t_{3a}	t_{3b}	t_{tot}	S_p
3000	1×1	2211	165	13.8	107	2497	1.00
3000	2×2	849	95.8	6.79	70.2	1022	2.44
3000	4×4	256	30.5	3.16	18.3	308	8.11
3000	8×8	158	22.6	2.53	14.8	198	12.6
6000	2×2	7057	777	47.4	568	8449	1.00
6000	4×4	2327	360	19.7	267	2974	2.84
6000	8×8	693	135	8.16	99.2	935	9.04

8 Extension to the generalized real Schur form

We have extended the presented parallel algorithm for reordering the standard Schur form to eigenvalue reordering in the generalized Schur form

$$(S, T) = Q^T (A, B) Z, \tag{16}$$

where (A, B) is a regular matrix pair, Q and Z are orthogonal matrices and (S, T) is the generalized real Schur form (see, e.g., [14]). Besides the fact that the parallel block algorithm now works on pairs of matrices, the generalized case does not differ in any substantial way from the standard case, except that

Table 4. Experimental output and accuracy results from computing stable invariant subspaces of Hamiltonian matrices on *seth*.

		Output			Accuracy				
n	$P_r \times P_c$	m	s	sep	R_{o1}	R_{r1}	R_{o2}	R_{r2}	R_{eig}
3000	1×1	1500	0.19E-03	0.14E-04	0.75E+00	0.98E-14	0.88E+00	0.12E-13	0.0E+00
3000	2×2	1503	0.27E-04	0.96E-06	0.75E+00	0.11E-13	0.86E+00	0.11E-13	0.0E+00
3000	4×4	1477	0.30E-04	0.40E-06	0.71E+00	0.89E-14	0.83E+00	0.11E-13	0.0E+00
3000	8×8	1481	0.48E-03	0.57E-06	0.73E+00	0.83E-14	0.86E+00	0.11E-13	0.0E+00
6000	2×2	2988	0.52E-03	0.32E-13	0.74E+00	0.13E-13	0.88E+00	0.16E-13	0.0E+00
6000	4×4	3015	0.57E-04	0.11E-17	0.72E+00	0.12E-13	0.89E+00	0.15E-13	0.0E+00
6000	8×8	2996	0.32E-04	0.25E-12	0.73E+00	0.63E-13	0.88E+00	0.64E-13	0.0E+00

the individual orthogonal transformation matrices from each eigenvalue swap are stored slightly different (see [27] for details). For our prototype Fortran 77 implementation `PBDTGSEN` the following close to optimal parameters was found by extensive tests: $n_b = 180$, $n_{\text{win}} = 60$ and $n_{\text{eig}} = n_{\text{ceig}} = 30$ and $r_{\text{mmult}} = 10$ which resulted in a uniprocessor runtime 18.94 seconds, which is less than 1 second slower that the results in [27], but much faster than LAPACK.

As for the standard case, `PBDTGSEN` optionally computes condition numbers for the selected cluster of eigenvalues and the corresponding deflating subspaces (see, e.g., [24]) by invoking the generalized coupled Sylvester equation solvers and condition estimators from the SCASY software [16, 17, 30].

Finally, we repeated the experiments from the standard case with similar experimental results, see Table 5. For $n = 6000$, the memory on one 2GB node of *seth* available for users is not large enough to hold all data objects (signaled by '-'). In this case, S_p is computed using $p_{\min} = 4$ in Equation (13), i.e., the presented values exemplify the speedup going from $p_0 = 4$ to $p > p_0$ processor (see also $n = 6000$ in Table 3). By assuming a fictive parallel speedup of at least 2 going from 1×1 to 2×2 processors for $n = 6000$ (see the results in Table 1), we conclude that the scalability for the generalized case is as good as or even better than the standard case.

9 Summary and future work

The lack of a fast and reliable parallel reordering routine has turned the attention away from Schur-based subspace methods for solving a wide range of numerical problems in distributed memory (DM) environments. By the introduction of the algorithm presented in this paper, this situation might be subject to change.

We remark that ScaLAPACK still lacks a highly efficient parallel implementation of the parallel QZ algorithm and the existing parallel QR algorithm has far from level 3 node performance. Moreover, the developed techniques here can be used to to implement parallel versions of the advanced deflations techniques described in [9, 22].

Table 5. Performance of `PBDTGSEN` on *seth*.

			Random		*Bottom*			*Random*		*Bottom*	
Sel.	$P_r \times P_c$	n	time	S_p	time	S_p	n	time	S_p	time	S_p
5%	1×1	1500	5.12	1.00	6.56	1.00	4500	98.4	1.00	105	1.00
	2×2	1500	2.65	1.93	2.91	2.25	4500	39.9	2.47	48.0	2.19
	3×3	1500	2.66	1.92	2.74	2.39	4500	23.4	4.21	32.4	3.24
	4×4	1500	2.20	2.32	2.50	2.62	4500	17.3	5.69	18.6	5.65
	5×5	1500	1.48	3.46	1.97	3.33	4500	15.2	6.47	17.0	6.18
	6×6	1500	1.05	4.88	1.69	3.88	4500	10.4	9.46	11.8	8.90
	7×7	1500	0.77	6.65	1.25	5.25	4500	8.93	11.0	9.22	11.4
	8×8	1500	0.60	8.53	0.67	9.79	4500	7.33	13.4	8.15	12.9
35%	1×1	1500	13.6	1.00	22.0	1.00	4500	279	1.00	312	1.00
	2×2	1500	8.02	1.70	11.4	1.93	4500	133	2.10	218	1.43
	3×3	1500	7.92	1.72	7.63	2.88	4500	77.2	3.61	122	2.56
	4×4	1500	3.52	3.86	5.41	4.07	4500	50.4	5.54	73.7	4.23
	5×5	1500	3.21	4.24	4.50	4.89	4500	40.4	6.91	59.5	5.24
	6×6	1500	3.71	3.67	4.85	4.54	4500	32.2	8.66	43.7	7.14
	7×7	1500	2.21	6.16	3.33	6.61	4500	26.1	10.7	40.3	7.74
	8×8	1500	2.15	6.33	2.84	7.75	4500	20.8	13.4	33.3	9.37
50%	1×1	1500	18.9	1.00	26.9	1.00	4500	301	1.00	584	1.00
	2×2	1500	13.8	1.37	14.5	1.89	4500	146	2.06	288	2.03
	3×3	1500	11.1	1.70	10.1	2.66	4500	82.9	3.63	169	3.46
	4×4	1500	4.88	3.87	7.94	3.39	4500	54.9	5.48	104	5.62
	5×5	1500	4.01	4.71	6.57	4.10	4500	41.7	7.22	91.9	6.35
	6×6	1500	3.18	5.94	4.19	6.42	4500	32.8	9.18	64.9	9.00
	7×7	1500	3.35	5.64	4.12	6.53	4500	32.1	9.37	62.9	9.28
	8×8	1500	3.01	6.23	3.96	6.79	4500	25.4	11.9	45.1	12.95
5%	1×1	3000	34.3	1.00	35.9	1.00	6000	-	-	-	-
	2×2	3000	13.5	2.54	15.3	2.35	6000	92.4	1.00	110	1.00
	3×3	3000	8.55	4.02	8.90	4.03	6000	51.1	1.81	58.3	1.89
	4×4	3000	6.13	5.60	5.79	6.20	6000	34.9	2.65	39.2	2.81
	5×5	3000	6.20	5.53	4.32	8.31	6000	29.2	3.16	33.0	3.33
	6×6	3000	4.59	7.47	3.50	10.3	6000	18.8	4.91	20.6	5.34
	7×7	3000	4.14	8.29	3.40	10.6	6000	16.0	5.78	19.6	5.61
	8×8	3000	2.69	12.8	2.69	13.3	6000	13.3	6.95	14.5	7.59
35%	1×1	3000	93.4	1.00	155	1.00	6000	-	-	-	-
	2×2	3000	44.6	2.09	74.4	2.08	6000	326	1.00	523	1.00
	3×3	3000	27.1	3.45	42.0	3.69	6000	180	1.81	280	1.87
	4×4	3000	19.2	4.86	25.4	6.10	6000	104	3.13	163	3.21
	5×5	3000	16.7	5.59	22.3	6.95	6000	88.1	3.70	135	3.87
	6×6	3000	12.4	7.53	18.1	8.56	6000	64.5	5.05	85.4	6.12
	7×7	3000	10.9	8.57	15.2	10.2	6000	50.7	6.43	80.9	6.46
	8×8	3000	9.80	9.53	11.8	13.1	6000	43.3	7.53	61.2	8.54
50%	1×1	3000	105	1.00	194	1.00	6000	-	-	-	-
	2×2	3000	53.9	1.95	98.6	1.97	6000	335	1.00	695	1.00
	3×3	3000	31.9	3.29	64.5	3.01	6000	187	1.79	391	1.78
	4×4	3000	21.5	4.88	35.8	5.42	6000	112	2.99	234	2.97
	5×5	3000	18.5	5.68	31.6	6.14	6000	87.5	3.83	181	3.83
	6×6	3000	14.1	7.45	24.3	7.98	6000	66.6	5.03	127	5.47
	7×7	3000	12.6	8.33	22.6	8.58	6000	62.0	5.40	114	6.10
	8×8	3000	9.81	10.7	20.3	9.56	6000	45.5	7.36	88.0	7.90

Acknowledgements

This research was conducted using the resources of the High Performance Computing Center North (HPC2N). Financial support has been provided by the *Swedish Research Council* under grant VR 621-2001-3284 and by the *Swedish Foundation for Strategic Research* under grant A3 02:128.

References

1. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, third edition, 1999.
2. Z. Bai and J. W. Demmel. On swapping diagonal blocks in real Schur form. *Linear Algebra Appl.*, 186:73–95, 1993.
3. Z. Bai, J. W. Demmel, J. J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems*. Software, Environments, and Tools. SIAM, Philadelphia, PA, 2000.
4. Z. Bai, J. W. Demmel, and A. McKenney. On computing condition numbers for the nonsymmetric eigenproblem. *ACM Trans. Math. Software*, 19(2):202–223, 1993.
5. M. W. Berry, J. J. Dongarra, and Y. Kim. A parallel algorithm for the reduction of a nonsymmetric matrix to block upper-Hessenberg form. *Parallel Comput.*, 21(8):1189–1211, 1995.
6. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. W. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.
7. BLAS - Basic Linear Algebra Subprograms. See `http://www.netlib.org/blas/index.html`.
8. K. Braman, R. Byers, and R. Mathias. The multishift QR algorithm, I: Maintaining well-focused shifts and level 3 performance. *SIAM J. Matrix Anal. Appl.*, 23(4):929–947, 2002.
9. K. Braman, R. Byers, and R. Mathias. The multishift QR algorithm, II: Aggressive early deflation. *SIAM J. Matrix Anal. Appl.*, 23(4):948–973, 2002.
10. J. Choi, J. J. Dongarra, and D. W. Walker. The design of a parallel dense linear algebra software library: reduction to Hessenberg, tridiagonal, and bidiagonal form. *Numer. Algorithms*, 10(3-4):379–399, 1995.
11. K. Dackland and B. Kågström. Blocked algorithms and software for reduction of a regular matrix pair to generalized Schur form. *ACM Trans. Math. Software*, 25(4):425–454, 1999.
12. J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.
13. J. J. Dongarra, S. Hammarling, and J. H. Wilkinson. Numerical considerations in computing invariant subspaces. *SIAM J. Matrix Anal. Appl.*, 13(1):145–161, 1992.
14. G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.
15. A. Grama, A. Gupta, G. Karypsis, and V. Kumar. *Introduction to Parallel Computing, Second Edition*. Addison-Wesley, 2003.
16. R. Granat and B. Kågström. Parallel Solvers for Sylvester-type Matrix Equations with Applications in Condition Estimation, Part I: Theory and Algorithms. Technical Report UMINF-07.15, Department of Computing Science, Umeå University,

SE-90187, UMEÅ, Sweden. Submitted to *ACM Transactions on Mathematical Software*, 2007.
17. R. Granat and B. Kågström. Parallel Solvers for Sylvester-type Matrix Equations with Applications in Condition Estimation, Part II: the SCASY Software. Technical Report UMINF-07.16, Department of Computing Science, Umeå University, SE-90187, UMEÅ, Sweden. Submitted to *ACM Transactions on Mathematical Software*, 2007.
18. W.W. Hager. Condition estimates. *SIAM J. Sci. Statist. Comput.*, (3):311–316, 1984.
19. G. Henry, D. S. Watkins, and J. J. Dongarra. A parallel implementation of the nonsymmetric QR algorithm for distributed memory architectures. *SIAM J. Sci. Comput.*, 24(1):284–311, 2002.
20. N. J. Higham. Fortran codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. *ACM Trans. of Math. Software*, 14(4):381–396, 1988.
21. B. Kågström. A direct method for reordering eigenvalues in the generalized real Schur form of a regular matrix pair (A, B). In *Linear algebra for large scale and real-time applications (Leuven, 1992)*, volume 232 of *NATO Adv. Sci. Inst. Ser. E Appl. Sci.*, pages 195–218. Kluwer Acad. Publ., Dordrecht, 1993.
22. B. Kågström and D. Kressner. Multishift Variants of the QZ Algorithm with Aggressive Early Deflation. *SIAM Journal on Matrix Analysis and Applications 29,* 1, 199–227, 2006.
23. B. Kågström and P. Poromaa. Distributed and shared memory block algorithms for the triangular Sylvester equation with sep^{-1} estimators. *SIAM J. Matrix Anal. Appl.*, 13(1):90–101, 1992.
24. B. Kågström and P. Poromaa. LAPACK-style algorithms and software for solving the generalized Sylvester equation and estimating the separation between regular matrix pairs. *ACM Trans. Math. Software 22,* 1, 78–103, 1996.
25. B. Kågström and P. Poromaa. Computing eigenspaces with specified eigenvalues of a regular matrix pair (A, B) and condition estimation: theory, algorithms and software. *Numer. Algorithms*, 12(3-4):369–407, 1996.
26. D. Kressner. *Numerical Methods and Software for General and Structured Eigenvalue Problems.* PhD thesis, TU Berlin, Institut für Mathematik, Berlin, Germany, 2004.
27. Daniel Kressner. Block algorithms for reordering standard and generalized Schur forms. *ACM Transactions on Mathematical Software*, 32(4):521–532, December 2006.
28. LAPACK - Linear Algebra Package. See `http://www.netlib.org/lapack/`.
29. R. Lehoucq and J. Scott. An evaluation of software for computing eigenvalues of sparse nonsymmetric matrices. Tech. Report MCS-P547-1195, Argonne National Laboratory, 1996.
30. SCASY - ScaLAPACK-style solvers for Sylvester-type matrix equations. See `http://www.cs.umu.se/~granat/scasy.html`.
31. V. Sima. *Algorithms for Linear-Quadratic Optimization*, volume 200 of *Pure and Applied Mathematics*. Marcel Dekker, Inc., New York, NY, 1996.
32. ScaLAPACK Users' Guide. See `http://www.netlib.org/scalapack/slug/`.
33. Stewart, G. W. and Sun, J.-G. *Matrix Perturbation Theory.* Academic Press, New York, 1990.
34. P. Van Dooren. A generalized eigenvalue approach for solving Riccati equations. *SIAM J. Sci. Statist. Comput.*, 2(2):121–135, June, 1981.