Robust level-3 BLAS Inverse Iteration from the Hessenberg Matrix

ANGELIKA SCHWARZ, Umeå University, Sweden

Inverse iteration is known to be an effective method for computing eigenvectors corresponding to simple and well-separated eigenvalues. In the non-symmetric case, the solution of shifted Hessenberg systems is a central step. Existing inverse iteration solvers approach the solution of the shifted Hessenberg systems with either RQ or LU factorizations and, once factored, solve the corresponding systems. This approach has limited level-3 BLAS potential since distinct shifts have distinct factorizations. This paper rearranges the RQ approach such that data shared between distinct shifts can be exploited. Thereby the backward substitution with the triangular R factor can be expressed mostly with matrix–matrix multiplications (level-3 BLAS). The resulting algorithm computes eigenvectors in a tiled, overflow-free, and task-parallel fashion. The numerical experiments show that the new algorithm outperforms existing inverse iteration solvers for the computation of both real and complex eigenvectors.

CCS Concepts: • **Mathematics of computing** → **Solvers**; *Mathematical software performance;*

Additional Key Words and Phrases: Inverse iteration, shifted Hessenberg systems, overflow-free computation

ACM Reference format:
Angelika Schwarz. 2022. Robust level-3 BLAS Inverse Iteration from the Hessenberg Matrix. *ACM Trans. Math. Softw.* 48, 3, Article 25 (September 2022), 30 pages.
https://doi.org/10.1145/3544789

1 INTRODUCTION

Inverse iteration is an established method for computing eigenvectors. When an approximation λ to an eigenvalue of a matrix $\boldsymbol{A}$ is known, inverse iteration approximates an eigenvector by solving

$$(\boldsymbol{A} - \lambda \boldsymbol{I})z^{(k)} = \rho^{(k)} z^{(k-1)} \qquad k \geq 1.$$

Here, $z^{(0)}$ is a unit norm starting vector and $\rho^{(k)}$ is a scalar that normalizes the iterate $z^{(k)}$. A converging sequence of $z^{(k)}$ yields a right eigenvector $z \neq \mathbf{0}$ that is an exact eigenvector of a nearby matrix $\boldsymbol{A} + \boldsymbol{E}$ with $||\boldsymbol{E}|| = O(\epsilon||\boldsymbol{A}||)$, where ϵ denotes the machine precision, implying a small residual $||\boldsymbol{A}z - \lambda z|| = O(\epsilon||\boldsymbol{A}||)$.

This paper concerns the case when $\boldsymbol{A}$ is non-symmetric and real. A standard approach reduces $\boldsymbol{A}$ to an upper Hessenberg matrix $\boldsymbol{H} = \boldsymbol{Q}_0^T \boldsymbol{A} \boldsymbol{Q}_0$, where $\boldsymbol{Q}_0$ is an orthogonal matrix. Then inverse

Computing resources have been provided by the Swedish National Infrastructure for Computing (SNIC) at High-Performance Computing Center North (HPC2N), Umeå, Sweden, under the grants SNIC 2019/3-311 and SNIC 2020/5-286.
Author's address: A. Schwarz, Department of Computing Science, Umeå University, 901 87 Umeå, Sweden; email: angies@cs.umu.se.

0098-3500/2022/09-ART25 $15.00
https://doi.org/10.1145/3544789

iteration approximates an eigenvector $\boldsymbol{x} \neq \boldsymbol{0}$ of $\boldsymbol{H}$ by

$$(\boldsymbol{H} - \lambda \boldsymbol{I})\boldsymbol{x}^{(k)} = \sigma^{(k)}\boldsymbol{x}^{(k-1)} \qquad k \geq 1. \tag{1}$$

The scalar $\sigma^{(k)}$ is again chosen so that $\boldsymbol{x}^{(k)}$ has unit norm. A computed eigenvector $\boldsymbol{x}$ is backtransformed into an eigenvector of $\boldsymbol{A}$ by $\boldsymbol{z} \leftarrow \boldsymbol{Q}_0\boldsymbol{x}$. Varah [1968] showed that the starting vector $\boldsymbol{x}^{(0)}$ can be chosen such that a single iteration of (1) suffices. Then the task of computing an eigenvector $\boldsymbol{z}$ necessitates the efficient solution of shifted Hessenberg systems. This will be the topic of this paper.

Inverse iteration (1) hinges on the availability of good approximations λ to the true eigenvalues. These approximations can be computed through the QR algorithm *without* accumulating the orthogonal transformations [Golub and Van Loan 1996, Section 7.6.1]. This approach is indeed supported by the LAPACK 3.9.0 inverse iteration routine DHSEIN [Anderson et al. 1999]. If a subdiagonal entry of the Hessenberg matrix is zero, the eigenproblem decouples into smaller block-triangular problems. When the QR algorithm processes the blocks separately, DHSEIN exploits the known affiliation of the eigenvalue and the block it belongs to and performs inverse iteration only on the relevant block. This paper is therefore based on two assumptions. First, good approximations to the eigenvalues of $\boldsymbol{H}$ are available. In other words, each approximation λ is an exact eigenvalue of some matrix $\boldsymbol{H} + \boldsymbol{F}$, where $||\boldsymbol{F}|| = O(\epsilon)$. Second, the Hessenberg matrix $\boldsymbol{H}$ is unreduced, i.e., all subdiagonal entries are non-zeros $h_{j+1,j} \neq 0$ for all $j = 1, \ldots, n-1$.

This paper proposes a new algorithm for a more efficient computation of (1) if a batch of eigenvectors corresponding to distinct eigenvalues is sought. The new algorithm combines two ideas. The first idea is due to Henry [1994] who addresses the solution of a shifted Hessenberg system through an RQ factorization. In a single sweep, $\boldsymbol{H}$ is reduced column-by-column to an upper triangular $\boldsymbol{R}$ such that the newly computed column of $\boldsymbol{R}$ is immediately used in the backward substitution. The level-3 BLAS potential in this approach is limited since distinct shifts result in distinct RQ factorizations. This problem has been addressed by Bosner et al. [2018, Section 3] and gives the second idea. Level-3 BLAS can be introduced in spite of distinct shifts. A *partially* computed (tiled) RQ factorization preserves data that is shared between distinct shifts. The computation can be arranged such that most of the computation corresponds to matrix–matrix multiplications. Specifically, this paper contributes an inverse iteration algorithm based on solving shifted Hessenberg systems through the RQ approach with the following highlights.

- The RQ approach is revised such that most of the backward substitution corresponds to matrix–matrix multiplications (level-3 BLAS [Dongarra et al. 1990]) in spite of distinct eigenvalues.
- The new algorithm is tiled and can naturally be parallelized with tasks.
- In existing inverse iteration solvers, complex eigenvectors are computationally more expensive than real eigenvectors. The new algorithm supports the computation of real and complex eigenvectors alike, meaning that the computational cost per column is approximately the same.

The rest of this paper is organized as follows. Section 2 reviews the LU approach realized in DHSEIN and the RQ approach by Henry [1994] as well as the tiled RQ factorization by Bosner et al. [2018]. Inspired by these ideas, Section 3 presents the new algorithm for solving shifting Hessenberg system, which is the core of the inverse iteration routine developed in Section 4. Section 5 describes the numerical experiments and presents their results.

2 RELATED WORK

This section reviews approaches to the solution of shifted Hessenberg systems. First, the LU approach implemented in LAPACK and the RQ factorization advocated by Henry [1994] are discussed.

Then, the tiled RQ factorization by Bosner et al. [2018] delivers key ideas used in the new algorithm presented in Section 3.

2.1 LU Factorization

LAPACK includes the inverse iteration routine DLAEIN [Anderson et al. 1999] for computing a single eigenvector. DLAEIN addresses the solution of the first iteration of (1) through an LU factorization with partial pivoting

$$\boldsymbol{P}(\boldsymbol{H}-\lambda\boldsymbol{I})=\boldsymbol{L}\boldsymbol{U},\quad \boldsymbol{L}\boldsymbol{y}=\boldsymbol{P}\boldsymbol{x}^{(0)},\quad \boldsymbol{U}\boldsymbol{x}^{(1)}=\alpha\boldsymbol{y},\quad \boldsymbol{x}\leftarrow\sigma^{(1)}\boldsymbol{x}^{(1)}.$$

Here, $\boldsymbol{U}$ is upper triangular, $\boldsymbol{L}$ is lower unit triangular, and $\boldsymbol{P}$ is a permutation matrix. As the starting vector can be chosen such that a single iteration suffices [Peters and Wilkinson 1971; Varah 1968], DLAEIN executes only one half iteration solving $\boldsymbol{U}\boldsymbol{x}^{(1)}=\alpha\boldsymbol{y}$. The scalar $\alpha\in(0,1]$ serves the avoidance of overflow. The vector $\boldsymbol{y}$ is set to a scaled vector of ones $\boldsymbol{e}$, where the scaling depends on $\boldsymbol{H}$. In other words, the starting vector is implicitly selected as $\boldsymbol{x}^{(0)}=\boldsymbol{P}^{-1}\boldsymbol{L}\boldsymbol{e}$. If the first initial vector $\boldsymbol{y}$ does not satisfy the convergence criterion in the first half iteration, the initial vector is exchanged rather than computing more iterations. Distinct shifts yield distinct LU factorizations of $\boldsymbol{H}-\lambda\boldsymbol{I}$. Hence, when several eigenvectors corresponding to distinct eigenvalues are sought, a different LU factorization is computed for each eigenvalue. To not overwrite $\boldsymbol{H}$, the upper triangular factor $\boldsymbol{U}$ is computed in a workspace.

2.2 RQ and UL Factorization

Henry [1994; 1995] approaches the solution of $(\boldsymbol{H}-\lambda\boldsymbol{I})\boldsymbol{x}=\boldsymbol{b}$ through an RQ factorization. By applying suitable Givens rotations from the right,

$$(\boldsymbol{H}-\lambda\boldsymbol{I})\underbrace{\boldsymbol{G}_n^T\boldsymbol{G}_{n-1}^T\ldots\boldsymbol{G}_2^T}_{\boldsymbol{Q}^T}\underbrace{\boldsymbol{G}_2\ldots\boldsymbol{G}_n}_{\boldsymbol{Q}}\boldsymbol{x}=\boldsymbol{b},$$

the shifted Hessenberg matrix is transformed into an upper triangular matrix $\boldsymbol{R}=(\boldsymbol{H}-\lambda\boldsymbol{I})\boldsymbol{Q}^T$. The solution $\boldsymbol{x}$ is obtained by solving $\boldsymbol{R}\boldsymbol{y}=\boldsymbol{b}$ and backtransforming $\boldsymbol{x}=\boldsymbol{Q}^T\boldsymbol{y}$. The application of the Givens rotations successively computes the columns of $\boldsymbol{R}$ from right to left. Specifically, the Givens rotation $\boldsymbol{G}_k^T$ is constructed to annihilate the subdiagonal entry $h(k,k-1)$ and transforms the columns $k-1$ and k. As soon as a column of $\boldsymbol{R}$ has been computed, it is immediately used in a column-oriented backward substitution and then discarded. Henry uses an auxiliary column to compute and store this single column of $\boldsymbol{R}$. The full procedure is listed in Algorithm 1. The flop count of Algorithm 1 is $3.5n^2+O(n)$ for a real shift. By virtue of merging the computation of the R factor and the backward substitution, the matrix $\boldsymbol{H}$ is accessed only once. Since $\boldsymbol{H}$ is left untouched, several shifted Hessenberg systems can be solved simultaneously and benefit from improved temporal locality of accessing columns of $\boldsymbol{H}$.

When the shift in Algorithm 1 is complex, both the R and the Q factor become complex. The backward substitution with R then relies fully on complex arithmetic and multiplies complex vectors with complex scalars. To exploit that all entries but the diagonal of $\boldsymbol{H}-\lambda\boldsymbol{I}$ are real, Henry [1994, Section 4] employs the RQ approach only for real shifts. Complex shifts, by contrast, are addressed by an UL factorization. The UL approach can benefit from mixed real-complex arithmetic such that real vectors are multiplied with complex scalars. Depending on what Gauss transformation is used, the UL approach requires $3n^2+O(n)$ or $3.5n^2+O(n)$ flops.

2.3 Tiled RQ Factorization

Henry's RQ factorization has been generalized to better support the simultaneous solution of many Hessenberg systems. The origin of this generalization is in the computation of the complex

ALGORITHM 1: Solving $(\boldsymbol{H} - \lambda \boldsymbol{I})\boldsymbol{x} = \boldsymbol{b}$ with an RQ decomposition (Henry [1994, Algorithm 2])

$\boldsymbol{v} \leftarrow \boldsymbol{h}(1:n, n)$; $v(n) \leftarrow v(n) - \lambda$; $\boldsymbol{x} \leftarrow \boldsymbol{b}$;
for $k \leftarrow n : -1 : 2$ **do**
  Determine a Givens rotation $\boldsymbol{G}_k^T = \begin{bmatrix} c_k & -s_k \\ s_k & c_k \end{bmatrix}$ such that $[h(k, k-1) \quad v(k)]\boldsymbol{G}_k^T = [0 \quad \phi]$;
  // Backward substitution
  $x(k) = x(k)/\phi$;
  $\tau_1 \leftarrow s_k x(k)$; $\tau_2 \leftarrow c_k x(k)$;
  $\boldsymbol{x}(1:k-2) \leftarrow \boldsymbol{x}(1:k-2) - \tau_2 \boldsymbol{v}(1:k-2) + \tau_1 \boldsymbol{h}(1:k-2, k-1)$;
  $x(k-1) \leftarrow x(k-1) - \tau_2 v(k-1) + \tau_1 (h(k-1, k-1) - \lambda)$;
  // Compute column of the triangular factor
  $\boldsymbol{v}(1:k-2) \leftarrow c_k \boldsymbol{h}(1:k-2, k-1) + s_k \boldsymbol{v}(1:k-2)$;
  $v(k-1) \leftarrow c_k (h(k-1, k-1) - \lambda) + s_k v(k-1)$;
$\tau_1 \leftarrow x(1)/v(1)$;
// Backtransform $\boldsymbol{x} \leftarrow \boldsymbol{Q}^T \boldsymbol{x}$
for $k \leftarrow 2 : n$ **do**
  $\tau_2 \leftarrow x(k)$;
  $x(k-1) \leftarrow c_k \tau_1 - s_k \tau_2$;
  $\tau_1 \leftarrow c_k \tau_2 + s_k \tau_1$;
return $\boldsymbol{x}$;

frequency response function (transfer function) $\boldsymbol{G}(\sigma_\ell) = \boldsymbol{C}(\boldsymbol{A} - \sigma_\ell \boldsymbol{I})^{-1}\boldsymbol{B}$ of a linear time-invariant dynamical system $\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}\boldsymbol{u}(t)$, $\boldsymbol{y}(t) = \boldsymbol{C}\boldsymbol{x}(t)$. Here, $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is the system matrix, $\boldsymbol{B} \in \mathbb{R}^{n \times m}$ is the input, $\boldsymbol{C} \in \mathbb{R}^{p \times n}$ is the output, the shifts σ_ℓ are complex and $m, p \ll n$. Beattie, Drmač and Gugercin [2012] and Bosner, Bujanović and Drmač [2013; 2018] target the computation of $\boldsymbol{G}(\sigma_\ell)$ through a reduction to the controller-Hessenberg form $\tilde{\boldsymbol{C}}(\tilde{\boldsymbol{A}} - \sigma_\ell \boldsymbol{I})^{-1}\tilde{\boldsymbol{B}}$. Here, $\tilde{\boldsymbol{A}} = \boldsymbol{Q}^H \boldsymbol{A} \boldsymbol{Q} \in \mathbb{C}^{n \times n}$ is m-Hessenberg (Hessenberg with m subdiagonals), $\tilde{\boldsymbol{C}} = \boldsymbol{C}\boldsymbol{Q} \in \mathbb{C}^{p \times m}$, $\tilde{\boldsymbol{B}} = \boldsymbol{Q}^H \boldsymbol{B} \in \mathbb{C}^{n \times m}$ is upper trapezoidal and $\boldsymbol{Q} \in \mathbb{C}^{n \times n}$ is unitary. A solution can be obtained by substituting the RQ factorization $\tilde{\boldsymbol{A}} - \sigma_\ell \boldsymbol{I} = \boldsymbol{R}_\ell \boldsymbol{Q}_\ell$ into the controller-Hessenberg form. Then $\tilde{\boldsymbol{C}}(\boldsymbol{R}_\ell \boldsymbol{Q}_\ell)^{-1}\tilde{\boldsymbol{B}} = (\tilde{\boldsymbol{C}}\boldsymbol{Q}^H)(\boldsymbol{R}_\ell^{-1}\tilde{\boldsymbol{B}})$ can be evaluated by solving a triangular system and two matrix–matrix multiplications.

Next we summarize the idea presented in Bosner et al. [2013, 2018]. The authors devise a tile column-oriented algorithm that computes RQ factorizations simultaneously for distinct shifts σ_ℓ and with a significant fraction of level-3 BLAS operations. Given a block size n_b, the RQ factorizations are computed in tile columns of width $m + n_b$, where m is the number of subdiagonals of the m-Hessenberg matrix. For a batch of shifts $\ell = 1, \ldots, s$, the current tile column is split into a window on the diagonal and the remaining offdiagonal part. The window is n_b-by-$(m + n_\mathrm{b})$ and m-Hessenberg. It is reduced to triangular shape with m leading zero columns by applying unitary transformations from the right. The unitary transformations are *only* applied to the window and are accumulated into a matrix $\boldsymbol{Q}_\ell \in \mathbb{C}^{(m+n_b) \times (m+n_b)}$. It remains to update the offdiagonal part with $\boldsymbol{Q}_\ell^H$. The offdiagonal part is partitioned $[\boldsymbol{D} \quad \boldsymbol{E}_\ell]$. The matrix $\boldsymbol{D}$ is shared across shifts. The matrix $\boldsymbol{E}_\ell$ holds the shift-distinct columns that have been transformed with previous unitary transformations. The unitary matrix $\boldsymbol{Q}_\ell^H$ is partitioned conformally as $\boldsymbol{Q}_\ell^H = [\boldsymbol{U}_\ell \quad \boldsymbol{V}_\ell]^H$. The update $[\boldsymbol{D} \quad \boldsymbol{E}_\ell][\boldsymbol{U}_\ell \quad \boldsymbol{V}_\ell]^H$ is executed as a block matrix operation. First, $\boldsymbol{Z}_\ell \leftarrow \boldsymbol{E}_\ell \boldsymbol{V}_\ell^H$ is executed as a batched matrix–matrix multiplication for $\ell = 1, \ldots, s$. Then, the matrix–matrix multiplication $[\boldsymbol{Z}_1 \quad \cdots \quad \boldsymbol{Z}_s] \leftarrow [\boldsymbol{Z}_1 \quad \cdots \quad \boldsymbol{Z}_s] + \boldsymbol{D}[\boldsymbol{U}_1^H \quad \cdots \quad \boldsymbol{U}_s^H]$ with the shared $\boldsymbol{D}$ and copying back the tile columns complete the update.

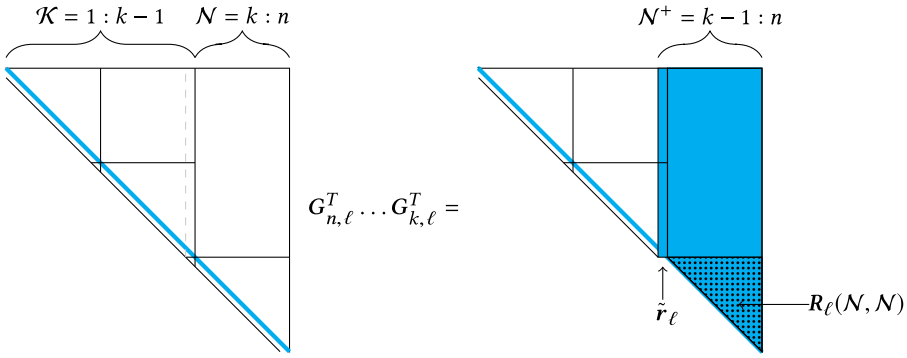

Fig. 1. Reduction of the rightmost tile column.

The tiled RQ approach delivers two key concepts that are used for the algorithm developed in the next section. First, the algorithm is designed as a tile column-oriented algorithm such that the shift-specific orthogonal transformations are only applied to the small window on the diagonal. Second, working with a partially reduced Hessenberg matrix and rearranging the computation reveals data that is shared between distinct shift. The next section applies these two concepts to Henry's column-oriented backward substitution algorithm to introduce level-3 BLAS potential.

3 SOLUTION OF SHIFTED HESSENBERG SYSTEMS WITH LEVEL-3 BLAS

In this section we devise a tile column-oriented algorithm that solves $(\boldsymbol{H} - \lambda_\ell \boldsymbol{I})\boldsymbol{x}_\ell = \boldsymbol{b}_\ell$ simultaneously for many distinct shifts λ_ℓ. For simplicity, we assume λ_ℓ to be real and defer the complex case to Section 4.3. Using ideas by Bosner et al. [2018], we adapt the RQ approach by Henry [1994] such that a large part of the data is shared. This way, the backward substitution phases can be merged for several shifts such that a large fraction of the computation corresponds to matrix–matrix multiplications.

3.1 Simultaneous Backward Substitution of a Batch of Shifts

The RQ approach requires a different sequence of Givens rotations for every shift λ_ℓ, $\ell = 1, \ldots, m$. The Given rotation

$$\boldsymbol{G}_{k,\ell}^T = \boldsymbol{I}_{k-2} \oplus \begin{bmatrix} c(k,\ell) & -s(k,\ell) \\ s(k,\ell) & c(k,\ell) \end{bmatrix} \oplus \boldsymbol{I}_{n-k} \in \mathbb{R}^{n \times n}, \tag{2}$$

where $\oplus$ denotes the direct sum, transforms the columns j and $j-1$ of $\boldsymbol{H} - \lambda_\ell \boldsymbol{I}$. To be available in the backtransformation, the cosine and sine components of all Givens rotations are recorded in the matrices $\boldsymbol{C} = [c(k,\ell)] \in \mathbb{R}^{n \times m}$ and $\boldsymbol{S} = [s(k,\ell)] \in \mathbb{R}^{n \times m}$.

The RQ approach transforms the Hessenberg matrix $(\boldsymbol{H} - \lambda_\ell \boldsymbol{I})\boldsymbol{G}_{n,\ell}^T \ldots \boldsymbol{G}_{2,\ell}^T = \boldsymbol{R}_\ell$. We apply the Givens rotations in batches and thereby compute a triangular factor $\boldsymbol{R}_\ell$ tile column by tile column starting from the right.

Rightmost tile column: The first batch of Givens rotations $\boldsymbol{G}_{n,\ell}^T, \ldots, \boldsymbol{G}_{k,\ell}^T$ transforms the column range $\mathcal{N} := k : n$ of $\boldsymbol{H} - \lambda_\ell \boldsymbol{I}$ into $\boldsymbol{R}_\ell(:, \mathcal{N})$. Figure 1 illustrates the situation; matrix entries that are different for every shift are highlighted. The Givens rotation $\boldsymbol{G}_{k,\ell}^T$ transforms the columns k and $k-1$, where $k-1$ is outside of the currently processed tile column. Hence, applying $\boldsymbol{G}_{k,\ell}^T$ yields a cross-over column $\tilde{\boldsymbol{r}}_\ell$.

The diagonal tile has been transformed into a triangular matrix $\boldsymbol{R}_\ell(\mathcal{N}, \mathcal{N})$. The system $\boldsymbol{R}_\ell(\mathcal{N}, \mathcal{N})\boldsymbol{x}_\ell(\mathcal{N}) = \boldsymbol{b}_\ell(\mathcal{N})$ can be solved by backward substitution for every shift (level-2 BLAS). Let $\mathcal{K} := 1 : k-1$ be the index range of the remaining row indices. In a tile column-oriented

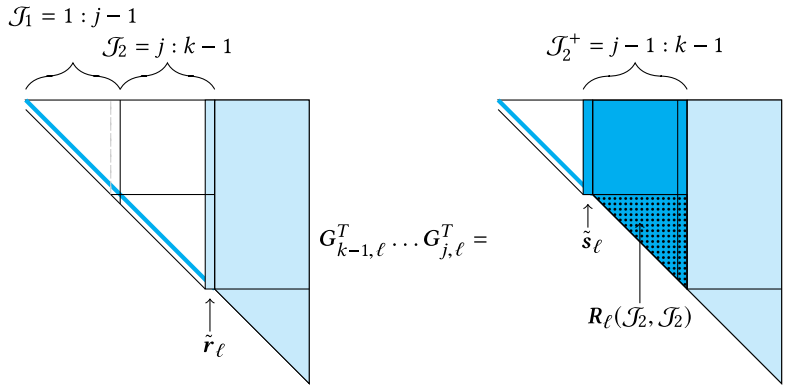

Fig. 2. Reduction of a center tile column.

backward substitution algorithm, the computed solution is used to update $\boldsymbol{b}_\ell(\mathcal{K}) \leftarrow \boldsymbol{b}_\ell(\mathcal{K}) - \boldsymbol{R}_\ell(\mathcal{K}, \mathcal{N})\boldsymbol{x}_\ell(\mathcal{N})$. If $\boldsymbol{R}_\ell(\mathcal{K}, \mathcal{N})$ is *not* generated explicitly, the update reads

$$\boldsymbol{b}_\ell(\mathcal{K}) \leftarrow \boldsymbol{b}_\ell(\mathcal{K}) - \underbrace{\begin{bmatrix}\boldsymbol{h}(\mathcal{K}, k-1) - \lambda_\ell \boldsymbol{e}_{k-1} & \boldsymbol{H}(\mathcal{K}, \mathcal{N})\end{bmatrix} G_{n,\ell}^T(\mathcal{N}^+, \mathcal{N}^+) \cdots G_{k,\ell}^T(\mathcal{N}^+, \mathcal{N}^+)}_{\begin{bmatrix}\tilde{\boldsymbol{r}}_\ell & \boldsymbol{R}_\ell(\mathcal{K}, \mathcal{N})\end{bmatrix}} \begin{bmatrix}0 \\ \boldsymbol{x}_\ell(\mathcal{N})\end{bmatrix},$$

where $\mathcal{N}^+ := k-1 : n$ and $\boldsymbol{e}_{k-1}$ is the vector whose $(k-1)$-th entry is one and all other entries are zero. If the Givens rotations are applied to the right-hand side, the update

$$\boldsymbol{b}_\ell(\mathcal{K}) \leftarrow \boldsymbol{b}_\ell(\mathcal{K}) - \begin{bmatrix}\boldsymbol{h}(\mathcal{K}, k-1) - \lambda_\ell \boldsymbol{e}_{k-1} & \boldsymbol{H}(\mathcal{K}, \mathcal{N})\end{bmatrix} \underbrace{G_{n,\ell}^T(\mathcal{N}^+, \mathcal{N}^+) \cdots G_{k,\ell}^T(\mathcal{N}^+, \mathcal{N}^+) \begin{bmatrix}0 \\ \boldsymbol{x}_\ell(\mathcal{N})\end{bmatrix}}_{\boldsymbol{z}_\ell}$$

can be interpreted as a block matrix multiplication where the second block $\boldsymbol{H}(\mathcal{K}, \mathcal{N})$ is shared across distinct shifts. Hence, when several right-hand sides $\boldsymbol{z}_\ell$ are computed simultaneously, the multiplication with the second block $\boldsymbol{H}(\mathcal{K}, \mathcal{N})$ corresponds to a matrix–matrix multiplication (level-3 BLAS). Only the scalar-vector multiplication with the first block $\boldsymbol{h}(\mathcal{K}, k-1) - \lambda_\ell \boldsymbol{e}_{k-1}$ is shift-specific. Note that the computation of $\boldsymbol{z}_\ell$ is cheap since every Givens rotation only transforms two entries of $\boldsymbol{x}_\ell$.

Center tile columns: In contrast to the rightmost tile column, center tile columns have shift-dependent cross-over columns and diagonal entries. Figure 2 illustrates the situation. Let $\mathcal{J}_1 := 1 : j-1$, $\mathcal{J}_2 := j : k-1$ and $\mathcal{J}_2^+ := j-1 : k-1$.

The applications of the batch of Givens rotations $\boldsymbol{G}_{k-1,\ell}^T, \ldots, \boldsymbol{G}_{j,\ell}^T$ yields

$$\left(\left[\begin{array}{c|cc}\boldsymbol{h}(\mathcal{J}_1, j-1) & \boldsymbol{H}(\mathcal{J}_1, j:k-2) & \tilde{\boldsymbol{r}}_\ell(\mathcal{J}_1) \\ \boldsymbol{h}(\mathcal{J}_2, j-1) & \boldsymbol{H}(\mathcal{J}_2, j:k-2) & \tilde{\boldsymbol{r}}_\ell(\mathcal{J}_2)\end{array}\right] - \lambda_\ell \left[\begin{array}{cc|c}\boldsymbol{e}_{j-1} & & \boldsymbol{0} \\ \boldsymbol{0} & & \boldsymbol{I} - \boldsymbol{e}_{k-1}\boldsymbol{e}_{k-1}^T\end{array}\right]\right)$$
$$G_{k-1,\ell}^T(\mathcal{J}_2^+, \mathcal{J}_2^+) \ldots G_{j,\ell}^T(\mathcal{J}_2^+, \mathcal{J}_2^+) = \left[\begin{array}{c|cc}\tilde{\boldsymbol{s}}_\ell & \boldsymbol{R}_\ell(\mathcal{J}_1, \mathcal{J}_2) \\ \boldsymbol{0} & \boldsymbol{R}_\ell(\mathcal{J}_2, \mathcal{J}_2)\end{array}\right]. \quad (3)$$

The matrix $\boldsymbol{I} - \boldsymbol{e}_{k-1}\boldsymbol{e}_{k-1}^T$ differs from the identity matrix in the last diagonal entry, which is zero. The matrix $\left[\begin{array}{c|c}\boldsymbol{e}_{j-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I} - \boldsymbol{e}_{k-1}\boldsymbol{e}_{k-1}^T\end{array}\right]$ ensures that the shift affects all diagonal entries of $\boldsymbol{H}$ in the current tile column, but not $\tilde{\boldsymbol{r}}_\ell$. The application of the Givens rotations yields $\boldsymbol{R}_\ell(:, \mathcal{J}_2)$ and a new cross-over column $\tilde{\boldsymbol{s}}_\ell$.

The diagonal tile is transformed into a triangular matrix $\boldsymbol{R}_\ell(\mathcal{J}_2, \mathcal{J}_2)$. The resulting triangular system $\boldsymbol{R}_\ell(\mathcal{J}_2, \mathcal{J}_2)\boldsymbol{x}_\ell(\mathcal{J}_2) = \boldsymbol{b}_\ell(\mathcal{J}_2)$ can be solved with backward substitution for every shift. Its solution $\boldsymbol{x}_\ell(\mathcal{J}_2)$ is used in a linear update. Instead of generating $\boldsymbol{R}_\ell(\mathcal{J}_1, \mathcal{J}_2)$ explicitly, the Givens rotations can again be applied to the right-hand side. Then the update

$$\boldsymbol{b}_\ell(\mathcal{J}_1) \leftarrow \boldsymbol{b}_\ell(\mathcal{J}_1) - \begin{bmatrix}\boldsymbol{h}(\mathcal{J}_1, j-1) - \lambda_\ell \boldsymbol{e}_{j-1} & \boldsymbol{H}(\mathcal{J}_1, j:k-2) & \tilde{\boldsymbol{r}}_\ell(\mathcal{J}_1)\end{bmatrix} \left(\boldsymbol{G}^T_{k-1,\ell}(\mathcal{J}_2^+, \mathcal{J}_2^+) \dots \boldsymbol{G}^T_{j,\ell}(\mathcal{J}_2^+, \mathcal{J}_2^+) \begin{bmatrix}0 \\ \boldsymbol{x}_\ell(\mathcal{J}_2)\end{bmatrix}\right) \tag{4}$$

reveals the shared data $\boldsymbol{H}(\mathcal{J}_1, j : k-2)$. If many right-hand sides are computed simultaneously, the computation can benefit from level-3 BLAS. Specifically, when interpreted as a block matrix multiplication, the update comprises a scalar-vector multiplication with $\boldsymbol{h}(\mathcal{J}_1, j-1) - \lambda_\ell \boldsymbol{e}_{j-1}$ (level-1 BLAS), a matrix–matrix multiplication with $\boldsymbol{H}(\mathcal{J}_1, j : k-2)$ (level-3 BLAS) and a shift-specific column update with $\tilde{\boldsymbol{r}}_\ell(\mathcal{J}_1)$ (level-1 BLAS).

The column range of the center tile is intentionally left open. It is certainly a possibility to have several center tile columns. In this case, a realization can iterate over center tile column from right to left until the leftmost tile column is reached.

Leftmost tile column: It remains to transform the top-left diagonal tile of $\boldsymbol{H}$ to triangular form. The application of the Givens rotations $\boldsymbol{G}^T_{j-1,\ell} \dots \boldsymbol{G}^T_{2,\ell}$ yields

$$\left(\begin{bmatrix}\boldsymbol{H}(\mathcal{J}_1, 1:j-2) & \tilde{\boldsymbol{s}}_\ell\end{bmatrix} - \lambda_\ell(\boldsymbol{I} - \boldsymbol{e}_{j-1}\boldsymbol{e}^T_{j-1})\right) \boldsymbol{G}^T_{j-1,\ell}(\mathcal{J}_1, \mathcal{J}_1) \dots \boldsymbol{G}^T_{2,\ell}(\mathcal{J}_1, \mathcal{J}_1) = \boldsymbol{R}_\ell(\mathcal{J}_1, \mathcal{J}_1).$$

The solution of the triangular system $\boldsymbol{R}_\ell(\mathcal{J}_1, \mathcal{J}_1)\boldsymbol{x}_\ell(\mathcal{J}_1) = \boldsymbol{b}_\ell(\mathcal{J}_1)$ with backward substitution finalizes the backward substitution phase.

Remark. The computational efficiency of (4) hinges on the availability of the cross-over column $\tilde{\boldsymbol{r}}_\ell$. If $\tilde{\boldsymbol{r}}_\ell$ is not stored, the right side of (4) is

$$\boldsymbol{b}_\ell(\mathcal{J}_1) - \underbrace{\boldsymbol{F}\begin{bmatrix}\boldsymbol{H}(\mathcal{J}_1, j-1:n) \\ \boldsymbol{H}(\mathcal{J}_2, j-1:n) - \lambda_\ell \boldsymbol{I}\end{bmatrix} \boldsymbol{G}^T_{n,\ell}(j-1:n, j-1:n) \dots \boldsymbol{G}^T_{j,\ell}(j-1:n, j-1:n)}_{\begin{bmatrix}\tilde{\boldsymbol{s}}_\ell & \boldsymbol{R}_\ell(\mathcal{J}_1, j:n)\end{bmatrix}} \begin{bmatrix}0 \\ \boldsymbol{x}_\ell(\mathcal{J}_2) \\ \boldsymbol{0}_{n-k+1}\end{bmatrix},$$

where $\boldsymbol{F} = [\boldsymbol{I}_{j-1\times j-1} \quad \boldsymbol{0}_{j-1\times n-j+1}]$ extracts the top $j-1$ rows and ensures matching dimensions with $\boldsymbol{b}_\ell(\mathcal{J}_1)$. If the Givens rotations are applied to the right vector encompassing $\boldsymbol{x}_\ell(\mathcal{J}_2)$, fill-in is generated and the vector becomes dense. Hence, no matter if the Givens rotations are applied to $\boldsymbol{H}$ or to $\boldsymbol{x}_\ell$, the computation depends on the n-th column of $\boldsymbol{H} - \lambda_\ell \boldsymbol{I}$. Storing the cross-over column $\tilde{\boldsymbol{r}}$ effectively prunes the computation of the triangular factor. If the cross-over columns are stored, the reduction and the backward substitution phase can be separated.

3.2 A Tiled Algorithm for Solving Shifted Hessenberg Systems

This section uses the ideas presented in the previous section to derive a tiled algorithm for the simultaneous solution of shifted Hessenberg systems. The Hessenberg matrix $\boldsymbol{H} \in \mathbb{R}^{n\times n}$ is partitioned into a grid of $N \times N$ tiles. To simplify the presentation of the algorithms, we assume that all tiles are $b_\mathrm{r} \times b_\mathrm{r}$ and that $n = Nb_\mathrm{r}$. The shifts are processed in batches of size b_c. We assume that the total number of shifts m is an integer multiple of b_c, i.e., $m = Mb_\mathrm{c}$. The two partitionings induce a partitioning of the right-hand sides $\boldsymbol{B} \in \mathbb{R}^{n\times m}$ into a grid of $N \times M$ tiles, where every tile is $b_\mathrm{r} \times b_\mathrm{c}$.

The algorithm is split into three phases. The *reduction phase* computes the Givens rotations that transform the shifted Hessenberg matrix into a triangular factor. Furthermore, it records the

ALGORITHM 2: Reduction of a shifted Hessenberg tile to triangular form

Data: $\boldsymbol{H} \in \mathbb{R}^{n\times n-1}$ and cross-over columns $\tilde{\boldsymbol{R}} = [\tilde{\boldsymbol{r}}_1, \ldots, \tilde{\boldsymbol{r}}_m] \in \mathbb{R}^{n\times m}$ such that $[\boldsymbol{H} \quad \tilde{\boldsymbol{r}}_\ell] \in \mathbb{R}^{n\times n}$ is upper Hessenberg, column $\boldsymbol{h}^{\text{left}} \in \mathbb{R}^n$, shifts $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_m]^T \in \mathbb{R}^m$

Result: Components $\boldsymbol{C} = [c(j,\ell)] \in \mathbb{R}^{n\times m}$ and $\boldsymbol{S} = [c(j,\ell)] \in \mathbb{R}^{n\times m}$ of $(n+1)$-by$(n+1)$ Givens rotations $\hat{\boldsymbol{G}}_{k,\ell}^T = \boldsymbol{I}_{k-1} \oplus \begin{bmatrix} c(k,\ell) & -s(k,\ell) \\ s(k,\ell) & c(,\ell) \end{bmatrix} \oplus \boldsymbol{I}_{n-k}$ whose application yields an upper triangular $\boldsymbol{R}_\ell$ through

$$([\boldsymbol{h}^{\text{left}} \mid \boldsymbol{H}\ \tilde{\boldsymbol{r}}_\ell + \lambda_\ell \boldsymbol{e}_n] - [\boldsymbol{0} \mid \lambda_\ell \boldsymbol{I}_n])\hat{\boldsymbol{G}}_{n,\ell}^T \ldots \hat{\boldsymbol{G}}_{1,\ell}^T = [\boldsymbol{0} \mid \boldsymbol{R}_\ell]. \tag{5}$$

1 **function** ReduceDiag($\boldsymbol{H}, \boldsymbol{h}^{\text{left}}, \boldsymbol{\Lambda}, \tilde{\boldsymbol{R}}$)
2 **for** $\ell \leftarrow 1 : m$ **do**
3 $\boldsymbol{v} \leftarrow \tilde{\boldsymbol{r}}_\ell$;
4 **for** $k \leftarrow n : -1 : 2$ **do**
5 Determine and record a Givens rotation such that $[h(k,k-1) \quad v(k)] \begin{bmatrix} c(k,\ell) & -s(k,\ell) \\ s(k,\ell) & c(k,\ell) \end{bmatrix} = [0 \quad \star]$;
6 $\boldsymbol{v}(1:k-2) \leftarrow c(k,\ell)\boldsymbol{h}(1:k-2,k-1) + s(k,\ell)\boldsymbol{v}(1:k-2)$;
7 $v(k-1) \leftarrow c(k,\ell)\,(h(k-1,k-1) - \lambda_\ell) + s(k,\ell)v(k-1)$;
8 **if** $\boldsymbol{h}^{\text{left}} \neq \boldsymbol{0}$ **then**
9 Determine and record a Givens rotation such that $[h^{\text{left}}(1) \quad v(1)] \begin{bmatrix} c(1,\ell) & -s(1,\ell) \\ s(1,\ell) & c(1,\ell) \end{bmatrix} = [0 \quad \star]$;
10 **else**
11 $c(1,\ell) \leftarrow 1; s(1,\ell) \leftarrow 0$; `// Identity matrix (padding)`
12 **return** $\boldsymbol{C}, \boldsymbol{S}$;

cross-over columns that allow computing linear updates in the backward substitution as in (4). The *backward substitution phase* solves the triangular systems. The *backtransform phase* transforms the computed solutions into solutions to the original shifted Hessenberg systems.

Reduction phase. The reduction phase computes and records the Givens rotations and cross-over columns required for the backward substitution phase. The computation proceeds tile column by tile column. The transformation of a tile column is split into ReduceDiag and ReduceOffdiag. ReduceDiag addresses the second block row of (3); ReduceOffdiag concerns the first block row of (3).

The kernel ReduceDiag computes the Givens rotations necessary to transform a diagonal tile to triangular form. Algorithm 2 lists the details. It closely resembles Algorithm 1, but omits the backward substitution and adds the cross-over columns. Note that in Algorithm 2 (5) the matrix $[\boldsymbol{h}^{\text{left}} \quad \boldsymbol{H} \quad \tilde{\boldsymbol{r}}_\ell]$ is n-by-$(n+1)$ and the Givens rotations are $(n+1)$-by-$(n+1)$ due to the cross-over columns. The case distinction (lines 8–11) aims at supporting the computation of the top-left diagonal tile. For a top-left diagonal tile, the column $\boldsymbol{h}^{\text{left}}$ does not physically exist. The driver routine TiledReduce discussed further down sets $\boldsymbol{h}^{\text{left}}$ to a zero column. In this case the computation of the Givens rotation (line 11) is meaningless. The first column of the storage matrices $\boldsymbol{C}$ and $\boldsymbol{S}$ of the Givens rotations is never read and can be viewed as padding. The flop count of Algorithm 2 is $1.5n^2 + O(n)$ per shift.

The kernel ReduceOffdiag concerns the first block row of (3) and is realized in Algorithm 3. It applies the Givens rotations computed in ReduceDiag to an offdiagonal tile and records the left cross-over column. Algorithm 3 requires $3nk + O(k)$ flops per shift if the tile being processed is k-by-n.

ALGORITHM 3: Reduction of an offdiagonal Hessenberg tile

Data: $\boldsymbol{H} \in \mathbb{R}^{k\times n}$, cross-over columns $\tilde{\boldsymbol{R}}^{\text{right}} = [\tilde{\boldsymbol{r}}_1^{\text{right}}, \ldots, \tilde{\boldsymbol{r}}_m^{\text{right}}] \in \mathbb{R}^{k\times m}$, components $\boldsymbol{C} = [c(j,\ell)] \in \mathbb{R}^{n\times m}$ and $\boldsymbol{S} = [s(j,\ell)] \in \mathbb{R}^{n\times m}$ of Givens rotations $\hat{\boldsymbol{G}}_{k,\ell}^T = \boldsymbol{I}_{k-1} \oplus \begin{bmatrix} c(k,\ell) & -s(k,\ell) \\ s(k,\ell) & c(k,\ell) \end{bmatrix} \oplus \boldsymbol{I}_{n-k}$ as computed by ReduceDiag, shifts $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_m]^T \in \mathbb{R}^m$.

Result: Cross-over columns $\tilde{\boldsymbol{R}}^{\text{left}} = [\tilde{\boldsymbol{r}}_1^{\text{left}}, \ldots, \tilde{\boldsymbol{r}}_m^{\text{left}}] \in \mathbb{R}^{k\times m}$ satisfying

$$\left[\begin{array}{c|cc} \boldsymbol{h}(1:k,1) - \lambda_\ell \boldsymbol{e}_k & \boldsymbol{H}(1:k,2:n) & \tilde{\boldsymbol{r}}_\ell^{\text{right}} \end{array}\right] \hat{\boldsymbol{G}}_{n,\ell}^T \ldots \hat{\boldsymbol{G}}_{1,\ell}^T = \left[\begin{array}{c|c} \tilde{\boldsymbol{r}}_\ell^{\text{left}} & \star \end{array}\right].$$

1 **function** ReduceOffdiag($\boldsymbol{H}, \tilde{\boldsymbol{R}}^{\text{right}}, \boldsymbol{\Lambda}$)
2 **for** $\ell \leftarrow 1 : m$ **do**
3 $\boldsymbol{v} \leftarrow \tilde{\boldsymbol{r}}_\ell^{\text{right}}$; `// Start with previously computed cross-over column`
4 **for** $j \leftarrow n : -1 : 2$ **do**
5 $\boldsymbol{v}(1:k) \leftarrow c(j,\ell)\boldsymbol{h}(1:k,j) + s(j,\ell)\boldsymbol{v}(1:k)$;
  `// Compute cross-over column`
6 $\tilde{\boldsymbol{r}}_\ell^{\text{left}}(1:k-1) \leftarrow c(1,\ell)\boldsymbol{h}(1:k-1,1) + s(1,\ell)\boldsymbol{v}(1:k-1)$; `// Shift-independent part`
7 $\tilde{r}_\ell^{\text{left}}(k) \leftarrow c(1,\ell)(h(k,1) - \lambda_\ell) + s(1,\ell)v(k)$; `// Last shift-dependent entry`
8 **return** $[\tilde{\boldsymbol{r}}_1^{\text{left}}, \ldots, \tilde{\boldsymbol{r}}_m^{\text{left}}]$;

The partitioning of $\boldsymbol{H}$ into tiles and the kernels ReduceDiag and ReduceOffdiag can be combined into a tiled algorithm for the reduction phase. The resulting algorithm TiledReduce is listed in Algorithm 4. In a tiled algorithm, the row range covered by the first block row of (3) is split into smaller tile rows. Since only the tiles directly above the diagonal tiles are affected by the shift, a case distinction is necessary. Line 17 of Algorithm 4 handles the tiles directly above the diagonal tiles. Line 19, by contrast, handles the remaining far-from-diagonal tiles that are unaffected by the shift. The shift is disabled by setting the corresponding parameter to zero.

Backward substitution phase. The backward substitution phase solves $\boldsymbol{R}_\ell \boldsymbol{x}_\ell = \boldsymbol{b}_\ell$ for every ℓ. The computation follows a standard pattern of tiled backward substitution. It iterates over tiles of the solution from bottom to top. For each iteration a new tile of the solution is computed with a small backward substitution. The small backward substitution is realized by Algorithm 5 Solve. Then the readily computed part of the solution is used in a tile update of above-lying tiles, realized by Algorithm 6 Update. These two kernels are combined to the tiled backward substitution algorithm TiledSolve listed in Algorithm 7. In the following, we present the details of the algorithms.

Solve realizes the small backward substitution. Since our algorithm splits the reduction phase and the backward substitution phase, the relevant part of the triangular factor has to be recomputed. Algorithm 5 gives the details. Analogously to Algorithm 2 ReduceDiag, the top-left diagonal tile of $\boldsymbol{H} - \lambda_\ell \boldsymbol{I}$ does not have a physical column $\boldsymbol{h}^{\text{left}}$ to its left. In that case, the driver routine TiledSolve discussed below sets $\boldsymbol{h}^{\text{left}}$ to a zero column and thereby skips the computation of the left cross-over column (line 7). The flop count of Algorithm 5 is $3.5n^2 + O(n)$.

Next we focus on the Update kernel. Recall the index ranges $\mathcal{J}_1 := 1 : j-1$, $\mathcal{J}_2 := j : k-1$ and $\mathcal{J}_2^+ := j-1 : k-1$. Algorithm 6 Update realizes (4). The application of the Givens rotations in (4) to the right transforms

$$\boldsymbol{G}_{k-1,\ell}^T(\mathcal{J}_2^+, \mathcal{J}_2^+) \ldots \boldsymbol{G}_{j,\ell}^T(\mathcal{J}_2^+, \mathcal{J}_2^+) \begin{bmatrix} 0 \\ \boldsymbol{x}_\ell(\mathcal{J}_2) \end{bmatrix} = \begin{bmatrix} \rho \\ \boldsymbol{z}_\ell \end{bmatrix},$$

ALGORITHM 4: Tiled reduction of a shifted Hessenberg matrix to triangular form and recording of the applied Givens rotations and relevant cross-over columns

Data: Hessenberg matrix $\boldsymbol{H} \in \mathbb{R}^{n\times n}$, tile size b_r with $n = Nb_\mathrm{r}$ and $N \geq 2$, shifts $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_m]^T \in \mathbb{R}^m$, tile size b_c with $m = Mb_\mathrm{c}$

Result: Givens rotations $\boldsymbol{G}_{j,\ell}^T$ as in (2) and stored as $\boldsymbol{C} = [c(j,\ell)] \in \mathbb{R}^{n\times m}$ and $\boldsymbol{S} = [s(j,\ell)] \in \mathbb{R}^{n\times m}$ such that $(\boldsymbol{H} - \lambda_\ell \boldsymbol{I})\boldsymbol{G}_{n,\ell}^T \ldots \boldsymbol{G}_{2,\ell}^T = \boldsymbol{R}_\ell$, cross-over columns $\tilde{\boldsymbol{R}}_\ell \in \mathbb{R}^{n\times N}$

1 **function** TiledReduce($\boldsymbol{H}, \boldsymbol{\Lambda}$)
2 **for** $\ell \leftarrow 1 : m$ **do**
3 $\tilde{\boldsymbol{r}}_\ell(:, N) \leftarrow \boldsymbol{h}(:, n)$; $\tilde{r}_\ell(n, N) \leftarrow \tilde{r}_\ell(n, N) - \lambda_\ell$;
4 **for** $\ell \leftarrow 1 : b_\mathrm{c} : m$ **do**
5 Set the tile column index $j_\mathrm{tile} \leftarrow N - 1$;
6 $\mathcal{L} \leftarrow \ell : \ell + b_\mathrm{c} - 1$; // Index range of current batch of shifts
7 **for** $j \leftarrow n - b_\mathrm{r} + 1 : -b_\mathrm{r} : b_\mathrm{r} + 1$ **do**
8 $\mathcal{J} \leftarrow j : j + b_\mathrm{r} - 1$; // Index range of current tile column
9 $\mathcal{J}^- \leftarrow j - 1 : j + b_\mathrm{r} - 2$; // Tile column index range shifted by 1
10 Partition $\left[\underbrace{\boldsymbol{h}^\mathrm{left}}_{b_\mathrm{r}\times 1} \quad \underbrace{\boldsymbol{H}^\mathrm{diag}}_{b_\mathrm{r}\times(b_\mathrm{r}-1)} \right] \leftarrow \boldsymbol{H}(\mathcal{J}, \mathcal{J}^-)$;
11 Pack $\tilde{\boldsymbol{R}}^\mathrm{right} \leftarrow \left[\tilde{\boldsymbol{r}}_\ell(\mathcal{J}, j_\mathrm{tile} + 1), \tilde{\boldsymbol{r}}_{\ell+1}(\mathcal{J}, j_\mathrm{tile} + 1), \ldots, \tilde{\boldsymbol{r}}_{\ell+b_\mathrm{c}-1}(\mathcal{J}, j_\mathrm{tile} + 1)\right]$;
12 $\boldsymbol{C}(\mathcal{J}, \mathcal{L}), \boldsymbol{S}(\mathcal{J}, \mathcal{L}) \leftarrow$ ReduceDiag($\boldsymbol{H}^\mathrm{diag}, \boldsymbol{h}^\mathrm{left}, \boldsymbol{\Lambda}(\mathcal{L}), \tilde{\boldsymbol{R}}^\mathrm{right}$);
13 **for** $i \leftarrow j - b_\mathrm{r} : -b_\mathrm{r} : 1$ **do**
14 $\mathcal{I} \leftarrow i : i + b_\mathrm{r} - 1$; // Index range of current tile row
15 Pack $\tilde{\boldsymbol{R}}^\mathrm{right} \leftarrow \left[\tilde{\boldsymbol{r}}_\ell(\mathcal{I}, j_\mathrm{tile} + 1), \tilde{\boldsymbol{r}}_{\ell+1}(\mathcal{I}, j_\mathrm{tile} + 1), \ldots, \tilde{\boldsymbol{r}}_{\ell+b_\mathrm{r}-1}(\mathcal{I}, j_\mathrm{tile} + 1)\right]$;
16 **if** $i + b_\mathrm{r} = j$ **then**
17 $[\tilde{\boldsymbol{r}}_\ell(\mathcal{I}, j_\mathrm{tile}), \tilde{\boldsymbol{r}}_{\ell+1}(\mathcal{I}, j_\mathrm{tile}), \ldots, \tilde{\boldsymbol{r}}_{\ell+b_\mathrm{r}-1}(\mathcal{I}, j_\mathrm{tile})] \leftarrow$ ReduceOffdiag($\boldsymbol{H}(\mathcal{I}, \mathcal{J}^-)$, $\tilde{\boldsymbol{R}}^\mathrm{right}, \boldsymbol{\Lambda}(\mathcal{L})$);
18 **else**
19 $[\tilde{\boldsymbol{r}}_\ell(\mathcal{I}, j_\mathrm{tile}), \tilde{\boldsymbol{r}}_{\ell+1}(\mathcal{I}, j_\mathrm{tile}), \ldots, \tilde{\boldsymbol{r}}_{\ell+b_\mathrm{r}-1}(\mathcal{I}, j_\mathrm{tile})] \leftarrow$ ReduceOffdiag($\boldsymbol{H}(\mathcal{I}, \mathcal{J}^-)$, $\tilde{\boldsymbol{R}}^\mathrm{right}, \boldsymbol{0}$);
20 $j_\mathrm{tile} \leftarrow j_\mathrm{tile} - 1$;
21 $\mathcal{J} \leftarrow 1 : b_\mathrm{r}$; // Index range of top-left corner
22 Pack $\tilde{\boldsymbol{R}}^\mathrm{right} \leftarrow \left[\tilde{\boldsymbol{r}}_\ell(\mathcal{J}, 1), \tilde{\boldsymbol{r}}_{\ell+1}(\mathcal{J}, 1), \ldots, \tilde{\boldsymbol{r}}_{\ell+b_\mathrm{c}-1}(\mathcal{J}, 1)\right]$;
23 $\boldsymbol{C}(\mathcal{J}, \mathcal{L}), \boldsymbol{S}(\mathcal{J}, \mathcal{L}) \leftarrow$ ReduceDiag($\boldsymbol{H}(\mathcal{J}, 1 : b_\mathrm{r} - 1), \boldsymbol{0}, \boldsymbol{\Lambda}(\mathcal{L}), \tilde{\boldsymbol{R}}^\mathrm{right}$);
24 **return** $\boldsymbol{C}, \boldsymbol{S}, (\tilde{\boldsymbol{R}}_1, \ldots, \tilde{\boldsymbol{R}}_m)$

which is computed in lines 3 and 9–13. The block matrix multiplication (4) with the left block yields the scalar-vector multiplication $\boldsymbol{b}_\ell \leftarrow \boldsymbol{b}_\ell - \rho(\boldsymbol{h}(\mathcal{J}_1, j-1) - \lambda_\ell \boldsymbol{e}_{j-1})$ (lines 4–5). The multiplication with the center block $\boldsymbol{H}(\mathcal{J}_1, j : k-2)$ is executed in line 14. When many vectors $\boldsymbol{z}_\ell$ are computed simultaneously, this operation corresponds to a matrix–matrix multiplication. Consequently, it can benefit from an efficient implementation of DGEMM available in an optimized BLAS library [Blackford et al. 2002; Dongarra et al. 1990]. The multiplication with the right block $\tilde{\boldsymbol{r}}_\ell$ is realized with the vector-scalar multiplication (line 16). The total flop count of Algorithm 6 is $2mnk + O((m+k)n)$. When several right-hand sides are computed simultaneously, the matrix–matrix multiplication in line 14 dominates the computation.

ALGORITHM 5: Small backward substitution

Data: $\boldsymbol{H} \in \mathbb{R}^{n\times n-1}$ and cross-over columns $\tilde{\boldsymbol{R}} = [\tilde{\boldsymbol{r}}_1, \ldots, \tilde{\boldsymbol{r}}_m] \in \mathbb{R}^{n\times m}$ such that $[\boldsymbol{H} \quad \tilde{\boldsymbol{r}}_\ell] \in \mathbb{R}^{n\times n}$ is upper Hessenberg, column $\boldsymbol{h}^{\text{left}}$, $\boldsymbol{X} \in \mathbb{R}^{n\times m}$, shifts $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_m]^T \in \mathbb{R}^m$, $\boldsymbol{B} \in \mathbb{R}^{n\times m}$, Givens rotations computed by TiledReduce and stored as $\boldsymbol{C} = [c(j,\ell)] \in \mathbb{R}^{n\times m}$ and $\boldsymbol{S} = [s(j,\ell)] \in \mathbb{R}^{n\times m}$.
Result: $\boldsymbol{X} \in \mathbb{R}^{n\times m}$ satisfying $\boldsymbol{R}_\ell \boldsymbol{x}(:,\ell) = \boldsymbol{b}(:,\ell)$ where $\boldsymbol{R}_\ell$ is given by (5).

```
1  function Solve(H, h^left, Λ, R̃, C, S, B)
2      X ← B;
3      for ℓ ← 1 : m do
4          v ← r̃_ℓ;
5          Compute x(2 : n, ℓ) using lines 2–9 of Algorithm 1;
6          if h^left ≠ 0 then
7              v(1) ← v(1)c(1, ℓ) − h^left(1)s(1, ℓ);
8          x(1, ℓ) ← x(1, ℓ)/v(1);
9      return X;
```

ALGORITHM 6: Linear tile update

Data: $\boldsymbol{H} \in \mathbb{R}^{m\times k}$, $\boldsymbol{X} \in \mathbb{R}^{k\times n}$, $\boldsymbol{B} \in \mathbb{R}^{m\times n}$, cross-over columns $\tilde{\boldsymbol{R}} = [\tilde{\boldsymbol{r}}_1, \ldots, \tilde{\boldsymbol{r}}_n] \in \mathbb{R}^{m\times n}$, $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_n]^T \in \mathbb{R}^n$, Givens rotations computed by TiledReduce and stored as $\boldsymbol{C} = [c(j,\ell)] \in \mathbb{R}^{k\times n}$ and $\boldsymbol{S} = [s(j,\ell)] \in \mathbb{R}^{k\times n}$
Result: (4) for $\ell = 1, \ldots, n$

```
1  function Update(H, R̃, X, Λ, C, S, B)
2      for ℓ ← 1 : n do
3          ρ ← s(1, ℓ)x(1, ℓ);
4          b(1 : m, ℓ) ← b(1 : m, ℓ) + ρh(1 : m, 1);          // Shift-specific DAXPY
5          b(m, ℓ) ← b(m, ℓ) − λ_ℓ ρ;
6      Copy Z ← X ;          // Use workspace to apply Givens rotation to the solution
7      for ℓ ← 1 : n do
8          z(1, ℓ) ← c_ℓ(1)z(1, ℓ);
9          for j ← 2 : k do
10             τ1 ← z(j − 1, ℓ);
11             τ2 ← z(j, ℓ);
12             z(j − 1, ℓ) ← c(j, ℓ)τ1 − s(j, ℓ)τ2;
13             z(j, ℓ) ← s(j, ℓ)τ1 + c(j, ℓ)τ2;
14     B ← B − H(1 : m, 2 : k)Z(1 : k − 1, 1 : n) ;                          // DGEMM
15     for ℓ ← 1 : n do
16         b(1 : m, ℓ) ← b(1 : m, ℓ) − r̃_ℓ z(k, ℓ);          // Shift-specific DAXPY
17     return B;
```

The partitioning into tiles and the routines Solve and Update lead to the tiled backward substitution algorithm TiledSolve listed in Algorithm 7. Offdiagonal tile column updates are split into smaller tile row updates. Only the tiles directly above diagonal tiles are affected by the shifts (line 16). Far-from-diagonal tile updates, by contrast, are unaffected by the shifts (line 18). The computation of $\boldsymbol{z}_\ell$ is repeated for each tile row. Based on numerical experiments, these additional flops are negligible compared to the gain from task parallelism, which will be introduced in the next section.

ALGORITHM 7: Tiled backward substitution and backtransform

Data: Hessenberg matrix $\boldsymbol{H} \in \mathbb{R}^{n\times n}$, tile size b_r with $n = Nb_\mathrm{r}$ and $N \geq 2$, shifts $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_m]^T \in \mathbb{R}^m$, $\boldsymbol{B} \in \mathbb{R}^{n\times m}$, tile size b_c with $m = Mb_\mathrm{c}$, Givens rotations $\boldsymbol{G}_{j,\ell}^T$ computed with Algorithm 4 and stored as $\boldsymbol{C} \in \mathbb{R}^{n\times m}$ and $\boldsymbol{S} \in \mathbb{R}^{n\times m}$, cross-over columns $\tilde{\boldsymbol{R}}_\ell \in \mathbb{R}^{n\times N}$ for $\ell = 1, \ldots, m$ computed with Algorithm 4.

Result: Solution $\boldsymbol{X} \in \mathbb{R}^{n\times m}$ to $\boldsymbol{HX} = \boldsymbol{B}\,\mathrm{diag}(\lambda_1, \ldots, \lambda_m)$

1 **function** TiledSolve($\boldsymbol{H}, \boldsymbol{\Lambda}, \boldsymbol{C}, \boldsymbol{S}, (\tilde{\boldsymbol{R}}_1, \ldots, \tilde{\boldsymbol{R}}_m), \boldsymbol{B}$)
2 $\quad \boldsymbol{X} \leftarrow \boldsymbol{B}$;
3 $\quad$ **for** $\ell \leftarrow 1 : b_\mathrm{c} : m$ **do**
4 $\quad\quad j_\mathrm{tile} \leftarrow N$; // Index of current tile column
5 $\quad\quad \mathcal{L} \leftarrow \ell : \ell + b_\mathrm{c} - 1$; // Index range of current batch of shifts
6 $\quad\quad$ **for** $j \leftarrow n - b_\mathrm{r} + 1 : -b_\mathrm{r} : 1$ **do**
7 $\quad\quad\quad \mathcal{J} \leftarrow j : j + b_\mathrm{r} - 1$; // Index range of current tile column
8 $\quad\quad\quad \mathcal{J}^- \leftarrow j - 1 : j + b_\mathrm{r} - 2$; // Tile column index range shifted by 1
9 $\quad\quad\quad$ Partition $\left[\underbrace{\boldsymbol{h}^\mathrm{left}}_{b_\mathrm{r}\times 1} \quad \underbrace{\boldsymbol{H}^\mathrm{diag}}_{b_\mathrm{r}\times(b_\mathrm{r}-1)} \right] \leftarrow \boldsymbol{H}(\mathcal{J}, \mathcal{J}^-)$;
10 $\quad\quad\quad$ Pack $\tilde{\boldsymbol{R}}^\mathrm{right} \leftarrow \left[\tilde{\boldsymbol{r}}_\ell(\mathcal{J}, j_\mathrm{tile}), \ldots, \tilde{\boldsymbol{r}}_{\ell+b_\mathrm{c}-1}(\mathcal{J}, j_\mathrm{tile})\right]$;
11 $\quad\quad\quad \boldsymbol{X}(\mathcal{J}, \mathcal{L}) \leftarrow$ Solve($\boldsymbol{H}^\mathrm{diag}, \boldsymbol{h}^\mathrm{left}, \tilde{\boldsymbol{R}}^\mathrm{right}, \boldsymbol{\Lambda}(\mathcal{L}), \boldsymbol{C}(\mathcal{J}, \mathcal{L}), \boldsymbol{S}(\mathcal{J}, \mathcal{L}), \boldsymbol{X}(\mathcal{J}, \mathcal{L})$);
12 $\quad\quad\quad$ **for** $i \leftarrow j - b_\mathrm{r} : -b_\mathrm{r} : 1$ **do**
13 $\quad\quad\quad\quad \mathcal{I} \leftarrow i : i + b_\mathrm{r} - 1$; // Index range of current tile row
14 $\quad\quad\quad\quad$ Pack $\tilde{\boldsymbol{R}}^\mathrm{right} \leftarrow \left[\tilde{\boldsymbol{r}}_\ell(\mathcal{I}, j_\mathrm{tile}), \tilde{\boldsymbol{r}}_{\ell+1}(\mathcal{I}, j_\mathrm{tile}), \ldots, \tilde{\boldsymbol{r}}_{\ell+b_\mathrm{c}-1}(\mathcal{I}, j_\mathrm{tile})\right]$;
15 $\quad\quad\quad\quad$ **if** $i + b_\mathrm{r} = j$ **then**
16 $\quad\quad\quad\quad\quad \boldsymbol{X}(\mathcal{I}, \mathcal{L}) \leftarrow$ Update($\boldsymbol{H}(\mathcal{I}, \mathcal{J}^-), \tilde{\boldsymbol{R}}^\mathrm{right}, \boldsymbol{X}(\mathcal{J}, \mathcal{L}), \boldsymbol{\Lambda}(\mathcal{L}), \boldsymbol{C}(\mathcal{J}, \mathcal{L}), \boldsymbol{S}(\mathcal{J}, \mathcal{L}), \boldsymbol{X}(\mathcal{I}, \mathcal{L})$);
17 $\quad\quad\quad\quad$ **else**
18 $\quad\quad\quad\quad\quad \boldsymbol{X}(\mathcal{I}, \mathcal{L}) \leftarrow$ Update($\boldsymbol{H}(\mathcal{I}, \mathcal{J}^-), \tilde{\boldsymbol{R}}^\mathrm{right}, \boldsymbol{X}(\mathcal{J}, \mathcal{L}), \boldsymbol{0}, \boldsymbol{C}(\mathcal{J}, \mathcal{L}), \boldsymbol{S}(\mathcal{J}, \mathcal{L}), \boldsymbol{X}(\mathcal{I}, \mathcal{L})$);
19 $\quad\quad\quad j_\mathrm{tile} \leftarrow j_\mathrm{tile} - 1$;
20 $\quad\quad \mathcal{J} \leftarrow 1 : b_\mathrm{r}$;
21 $\quad\quad$ Pack $\tilde{\boldsymbol{R}}^\mathrm{right} \leftarrow \left[\tilde{\boldsymbol{r}}_\ell(\mathcal{J}, j_\mathrm{tile}), \ldots, \tilde{\boldsymbol{r}}_{\ell+b_\mathrm{c}-1}(\mathcal{J}, j_\mathrm{tile})\right]$;
22 $\quad\quad \boldsymbol{X}(\mathcal{J}, \mathcal{L}) \leftarrow$ Solve($\boldsymbol{H}(\mathcal{J}, 1 : b_\mathrm{r} - 1), \boldsymbol{0}, \boldsymbol{\Lambda}(\mathcal{L}), \tilde{\boldsymbol{R}}^\mathrm{right}, \boldsymbol{C}(\mathcal{J}, \mathcal{L}), \boldsymbol{S}(\mathcal{J}, \mathcal{L}), \boldsymbol{X}(\mathcal{J}, \mathcal{L})$);
23 $\quad\quad \boldsymbol{X}(:, \mathcal{L}) \leftarrow$ Backtransform($\boldsymbol{C}(:, \mathcal{L}), \boldsymbol{S}(:, \mathcal{L}), \boldsymbol{X}(:, \mathcal{L})$);
24 $\quad$ **return** $\boldsymbol{X}$;

Backtransform phase. The solution $\boldsymbol{x}_\ell$ corresponding to a shift λ_ℓ is backtransformed

$$\boldsymbol{y}_\ell \leftarrow \boldsymbol{G}_{n,\ell}^T \left(\ldots \left(\boldsymbol{G}_{2,\ell}^T \boldsymbol{x}_\ell \right) \right). \tag{5}$$

The components of the Givens rotations $\boldsymbol{G}_{j,\ell}^T$ (2) have been recorded as $\boldsymbol{C} = [c(j,\ell)] \in \mathbb{R}^{n\times m}$ and $\boldsymbol{S} = [s(j,\ell)] \in \mathbb{R}^{n\times m}$ in the reduction phase. A routine Backtransform realizes (5) for a batch of vectors. A possible implementation is a column-by-column backtransform with the lines 11–14 of Algorithm 1. The flop count for the backtransform is $O(n)$ per shift. Line 23 of Algorithm 7 uses Backtransform to backtransform a batch of vectors.

The successive execution of TiledReduce and TiledSolve yields a tiled solver for the simultaneous solution of shifted Hessenberg systems. This solver is listed in Algorithm 8 and can be viewed as an extension to Algorithm 1. The next section parallelizes Algorithm 8.

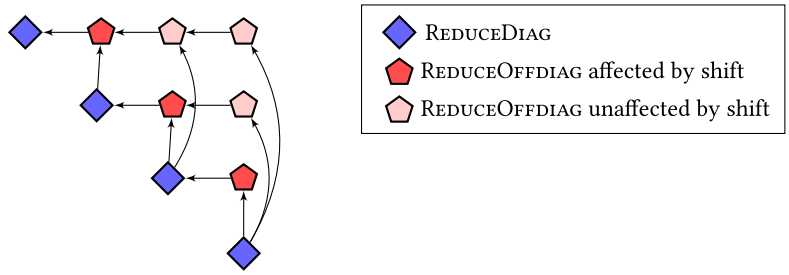

Fig. 3. DAG constructed for a single batch of shifts and a tile column count of $n/b_r = 4$ in Algorithm 4 TiledReduce. The task arrangement is based on what tile of $\boldsymbol{H}$ is processed.

ALGORITHM 8: Solving $(\boldsymbol{H} - \lambda_\ell \boldsymbol{I})\boldsymbol{x}_\ell = \boldsymbol{b}_\ell$ simultaneously in a tiled fashion

1 $C, S, (\tilde{R}_1, \ldots, \tilde{R}_m) \leftarrow$ TiledReduce$(\boldsymbol{H}, \boldsymbol{\Lambda})$;
2 $\boldsymbol{X} \leftarrow$ TiledSolve$(\boldsymbol{H}, \boldsymbol{\Lambda}, C, S, (\tilde{R}_1, \ldots, \tilde{R}_m), \boldsymbol{B})$;

3.3 Parallelization with Tasks

Libraries such as PLASMA [Dongarra et al. 2019] and FLAME [Van Zee et al. 2009] show that the design of dense linear algebra algorithms as tiled algorithms and the parallelization with tasks is an effective way of achieving high performance. A task-parallel algorithm expresses the computation as a **Directed Acyclic Graph (DAG)**. The nodes of the DAG correspond to tasks and represent tile operations. The edges of the DAG represent data dependencies between the tasks. A runtime system asynchronously executes the tasks while satisfying all data dependencies.

This section parallelizes Algorithm 8 with tasks. The implementation follows the usual approach and defines each tile operation as a task. Hence, every function call in Algorithm 4 TiledReduce and Algorithm 7 TiledSolve is a task. What remains is the definition of the edges of the DAG. To reduce the amount of edges and the associated overhead, the parallel implementation maintains two separate DAGs. The first DAG represents the computation executed in TiledReduce; the second DAG represents TiledSolve. All tasks related to TiledReduce must have been completed before any task of TiledSolve is started. The implementation realizes this with a synchronization point. Next we define the edges of each DAG.

The data dependencies of Algorithm 4 TiledReduce are as follows. A task ReduceDiag processing the tile (j, j) of $\boldsymbol{H}$ has outgoing dependencies to ReduceOffdiag tasks processing above-lying tiles (i, j) in the same tile column of $\boldsymbol{H}$, $i = 1, \ldots, j-1$. A ReduceOffdiag task on (i, j) has an outgoing dependence to the task processing the left-lying tile $(i, j-1)$. This is either a ReduceDiag tasks $(j - 1 = i)$ or another ReduceOffdiag task $(j - 1 > i)$. Figure 3 illustrates the task graph for one batch of shifts, i.e., one iteration of line 4 in Algorithm 4. Batches of shifts are independent of each other.

Algorithm 7 TiledSolve has three task types whose dependencies are as follows. A Solve task on (j, j) has outgoing dependencies to Update tasks (i, j), $i = 1, \ldots, j-1$. Before a Solve on the tile (j, j) can be executed, all updates (i, j), $j > i$ must have been completed. Once the backward substitution is completed, the Backtransform can be executed. Figure 4 shows the DAG for one iteration of line 3 in Algorithm 7. Analogously to the task-parallel execution of TiledReduce, batches of shifts do not have any dependencies.

The implementation adds one simplification to the DAG shown in Figure 4. The computation of the final Solve task and the Backtransform are merged. This step aims at reducing scheduling overhead.

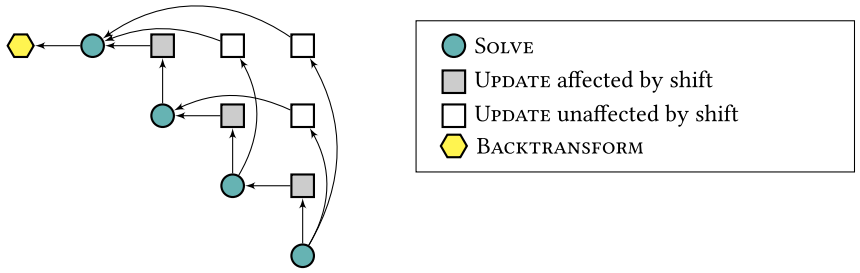

Fig. 4. DAG constructed for a single batch of shifts and a tile column count of $n/b_r = 4$ in Algorithm 7 TiledSolve. The task arrangement is based on what tile of $\boldsymbol{H}$ is processed.

3.4 Further Improvements

The presentation of the algorithms has focused on the algorithmic aspect, namely the introduction of level-3 BLAS operations to the backward substitution phase and the phrasing as tiled algorithms that can be parallelized with tasks. A natural step is to incorporate the ideas proposed for the original RQ approach. This includes improvements of the cache efficiency and the aggregation of a small number of orthogonal transformations for a joint application [Henry 1994, p. 8; Beattie et al. 2012, Sec. 5.2.1] to lower the flop count. The impact of these improvements has been investigated in Schwarz [2020]. The results of the numerical experiments in this work are solely based on the algorithmic improvements presented in this work.

4 ROBUST COMPUTATION OF EIGENVECTORS BY INVERSE ITERATION

This section extends Algorithm 8 for solving shifted Hessenberg systems to the computation of eigenvectors. The algorithm relies on the ability to solve triangular systems. The solution of triangular systems is known to be prone to overflow. This is particularly true if shifted Hessenberg systems are solved as part of an inverse iteration algorithm. In that case $\boldsymbol{H} - \lambda \boldsymbol{I}$ can be expected to be ill-conditioned if λ is close to a true eigenvalue of $\boldsymbol{H}$. By adding overflow protection, Section 4.1 renders Algorithm 8 robust while preserving the tiled structure. This yields the robust, tiled shifted Hessenberg system solver DHSRQ3. Section 4.2 concerns the convergence criterion and the choice of the starting vector. Together with the robust, tiled shifted Hessenberg solver, this results in a robust tiled inverse iteration algorithm DHSRQ3IN. Section 4.3 discusses the modifications necessary to support complex shifts.

4.1 Ill-conditioned Systems and Overflow Protection

Inverse iteration assumes that a good approximation λ to a true eigenvalue of $\boldsymbol{H}$ is available. In that case λ is an exact eigenvalue of a nearby matrix $\boldsymbol{H} + \boldsymbol{E}$ where $\|\boldsymbol{E}\| = O(\epsilon)$ is of the order of the machine precision. The matrix $\boldsymbol{H} + \boldsymbol{E} - \lambda \boldsymbol{I}$ is exactly singular. Hence, $\boldsymbol{H} - \lambda \boldsymbol{I}$ is close-to-singular and can be expected a have high condition number. A detailed explanation why the solution of a very ill-conditioned system does not produce completely erroneous solutions is given by Ipsen [1997, Section 6.3], and Parlett [1998, Section 4.3]. In fact, inverse iterations turns the ill-conditioning into an advantage because $\|(\boldsymbol{H} - \lambda \boldsymbol{I})^{-1}\| \geq 1/\|\boldsymbol{E}\| = 1/\epsilon$ shows that there is at least one vector whose norm grows in the order of $1/\epsilon$. Section 4.2 picks this up for the selection of a starting vector.

While the ill-conditioning is not a problem for the accuracy, the computed solution can be so large that the representational range is exceeded. A mechanism to avoid such a floating-point overflow is necessary. For the RQ approach, once $\boldsymbol{H} - \lambda \boldsymbol{I}$ has been factored, the backward substitution with the corresponding triangular factor can encounter overflow. To eliminate the possibility of floating-point overflow, implementations of inverse iteration [Peters and Wilkinson 1971, p. 435]

and shifted Hessenberg system solvers [Henry 1994, 1995] introduce a scaling factor $\gamma \in (0, 1]$ and solve the scaled triangular linear system $\boldsymbol{R}\boldsymbol{x} = \gamma\boldsymbol{b}$ for the scaled solution $\gamma^{-1}\boldsymbol{x}$. By virtue of γ, the current representation of the solution can be rescaled such that overflow is avoided. This renders the solution process robust. A robust solver for this scaled triangular system is, for example, DLATRS [Anderson 1991] available in LAPACK 3.9.0.

An extension to the solution of scaled triangular linear systems with n right-hand sides has been introduced by Kjelgaard Mikkelsen and Karlsson [2017a] and concerns $\boldsymbol{R}\boldsymbol{X} = \boldsymbol{B}\,\mathrm{diag}(\gamma_1, \ldots, \gamma_n)$. Each right-hand-side is associated with a scaling factor $\gamma_k \in (0, 1]$. If $\boldsymbol{B}$ is overwritten with the solution $\boldsymbol{X}$ in a standard (non-robust) tiled backward substitution algorithm, tile updates read $\boldsymbol{X}_i \leftarrow \boldsymbol{X}_i - \boldsymbol{R}_{ij}\boldsymbol{X}_j, i < j$. These tile updates are rendered robust through segment-wise scaling factors. There is one scaling factor per column per tile. Then robust tile updates are $\boldsymbol{X}_i\,\mathrm{diag}(\delta_1^{-1}, \ldots, \delta_n^{-1}) \leftarrow \boldsymbol{X}_i\,\mathrm{diag}(\beta_1^{-1}, \ldots, \beta_n^{-1}) - \boldsymbol{R}_{ij}(\boldsymbol{X}_j\,\mathrm{diag}(\alpha_1^{-1}, \ldots, \alpha_n^{-1}))$. Overflow is avoided by bounding the maximum possible growth in each update, computing suited scaling factors $\delta_1^{-1}, \ldots, \delta_n^{-1}$ and, if necessary, rescaling the current representation of the solution prior to the matrix–matrix multiplication. This preserves the level-3 BLAS potential of a tile update. Since tile updates during the tiled backward substitution can require different scalings, the final tiled representation of the solution can be inconsistently scaled. A consistent scaling is computed by reducing the segment-wise scaling factors to the global scaling factor γ_k for each column. Then all column segments are rescaled with respect to γ_k. This approach supports task parallelism as demonstrated by Kjelgaard Mikkelsen et al. [2019] and is used to render the algorithms presented in this paper robust.

A robust tiled algorithm for solving shifted Hessenberg systems requires (a) a robust routine for the solution of small shifted Hessenberg systems, (b) a robust tile update, and (c) a robust back-transform. The combination of these three robust kernels yields a robust version of TiledReduce. The robust routines are marked with the prefix R to easily distinguish between the non-robust and the robust version.

Robust solution of small shifted Hessenberg systems. A robust counterpart of Algorithm 5 Solve requires a small robust backward substitution routine solving $\boldsymbol{R}_\ell\boldsymbol{x}_\ell = \gamma_\ell\boldsymbol{b}_\ell$, $\gamma_\ell \in (0, 1]$. If the (small) $\boldsymbol{R}_\ell$ is generated explicitly, a possible realization is a call to the LAPACK routine DLATRS. This approach is realized by Algorithm 9 RSolve. DLATRS returns $\boldsymbol{x}_\ell$ and γ_ℓ representing $\gamma_\ell^{-1}\boldsymbol{x}_\ell$. The total scaling of the computed solution is computed in line 13 by multiplying the input scaling factor β_ℓ and γ_ℓ. Together, RSolve returns the scaled vector $(\gamma_\ell\beta_\ell)^{-1}\boldsymbol{x}_\ell$.

Robust tile update. The robust version of the tile update (4) is

$$\delta_\ell^{-1}\boldsymbol{b}_\ell(\mathcal{J}_1) \leftarrow \beta_\ell^{-1}\boldsymbol{b}_\ell(\mathcal{J}_1) - \Big[\boldsymbol{h}(\mathcal{J}_1, j-1) - \lambda_\ell\boldsymbol{e}_{j-1} \quad \boldsymbol{H}(\mathcal{J}_1, j:k-2) \quad \tilde{\boldsymbol{r}}_\ell(\mathcal{J}_1)\Big] \left(\boldsymbol{G}_{k-1,\ell}^T(\mathcal{J}_2^+, \mathcal{J}_2^+) \ldots \boldsymbol{G}_{j,\ell}^T(\mathcal{J}_2^+, \mathcal{J}_2^+) \begin{bmatrix} 0 \\ \alpha_\ell^{-1}\boldsymbol{x}_\ell(\mathcal{J}_2) \end{bmatrix}\right), \tag{6}$$

where $\delta_l \in (0, 1]$ is chosen such that $\boldsymbol{b}_\ell(\mathcal{J}_1)$ does not exceed the overflow threshold. There are many instantiations of δ_l and $\boldsymbol{b}_\ell(\mathcal{J}_1)$ that satisfy (4). Algorithm 10 computes one feasible instantiation. Following Kjelgaard Mikkelsen and Karlsson [2017b], the right-hand side tile $\boldsymbol{X}$ is associated with a vector of scaling factors $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_n] \in (0, 1]^n$ and represents the column-wise scaled matrix $\boldsymbol{X}\,\mathrm{diag}(\alpha_1^{-1}, \ldots, \alpha_n^{-1})$. Similarly, the tile $\boldsymbol{B}$ is associated with $\boldsymbol{\beta} = [\beta_1, \ldots, \beta_n] \in (0, 1]^n$ and represents $\boldsymbol{B}\,\mathrm{diag}(\beta_1^{-1}, \ldots, \beta_n^{-1})$. To compute a tile update robustly, $\boldsymbol{X}$ and $\boldsymbol{B}$ must be consistently scaled. The consistent scaling factor corresponds to the smaller of the two scaling factors (line 5). The remaining computation requires the overflow-free realization of three linear updates.

Each of the three linear updates is guarded by ProtectUpdate introduced by Kjelgaard Mikkelsen and Karlsson [2017b, Section 2.2]. ProtectUpdate receives $\|\boldsymbol{B}\|_\infty$, $\|\boldsymbol{H}\|_\infty$, $\|\boldsymbol{X}\|_\infty$ and

ALGORITHM 9: Small robust backward substitution

Data: $\boldsymbol{H} \in \mathbb{R}^{n\times n-1}$ and cross-over columns $\tilde{\boldsymbol{R}} = [\tilde{\boldsymbol{r}}_1, \ldots, \tilde{\boldsymbol{r}}_m] \in \mathbb{R}^{n\times m}$ such that $\begin{bmatrix}\boldsymbol{H} & \tilde{\boldsymbol{r}}_\ell\end{bmatrix} \in \mathbb{R}^{n\times n}$ is upper Hessenberg, column $\boldsymbol{h}^{\text{left}}$, $\boldsymbol{B} \in \mathbb{R}^{n\times m}$ and scaling factors $\boldsymbol{\beta} = [\beta_1, \ldots, \beta_m] = (0, 1]^m$ representing $\boldsymbol{B}\,\text{diag}(\beta_1, \ldots, \beta_m)$, shifts $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_m]^T \in \mathbb{R}^m$, Givens rotations computed by TiledReduce and stored as $\boldsymbol{C} = [c(j,\ell)] \in \mathbb{R}^{n\times m}$ and $\boldsymbol{S} = [s(j,\ell)] \in \mathbb{R}^{n\times m}$.

Result: $\boldsymbol{X} \in \mathbb{R}^{n\times m}$ and $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_m] \in (0, 1]^m$ satisfying $\boldsymbol{R}_\ell \boldsymbol{x}(:,\ell) = \alpha_\ell \boldsymbol{b}(:,\ell)$ where $\boldsymbol{R}_\ell$ is given by (5).

1 **function** RSolve($\boldsymbol{H}, \boldsymbol{h}^{\text{left}}, \tilde{\boldsymbol{R}}, \boldsymbol{\Lambda}, \boldsymbol{C}, \boldsymbol{S}, \boldsymbol{\beta}, \boldsymbol{B}$)
2 $\quad \boldsymbol{X} \leftarrow \boldsymbol{B}$;
3 $\quad$ **for** $\ell \leftarrow 1 : m$ **do**
$\quad\quad$ `// Recompute the triangular factor`
4 $\quad\quad \boldsymbol{R}_\ell \leftarrow \boldsymbol{0}_{n\times n}$;
5 $\quad\quad \boldsymbol{r}_\ell(:, n) \leftarrow \tilde{\boldsymbol{r}}_\ell$;
6 $\quad\quad$ **for** $k \leftarrow n : -1 : 2$ **do**
7 $\quad\quad\quad \boldsymbol{t}_\ell \leftarrow \boldsymbol{h}(1 : k, k-1)$; $t_\ell(k-1) \leftarrow t(k-1) - \lambda_\ell$;
8 $\quad\quad\quad \boldsymbol{R}_\ell(1 : k, k-1 : k) \leftarrow \begin{bmatrix}\boldsymbol{t}_\ell(1:k) & \boldsymbol{r}_\ell(1:k,k)\end{bmatrix}\begin{bmatrix}c(k,\ell)\\ s(k,\ell)\end{bmatrix}$;
9 $\quad\quad$ **if** $\boldsymbol{h}^{\text{left}} \neq \boldsymbol{0}$ **then**
10 $\quad\quad\quad r_\ell(1,\ell) \leftarrow r_\ell(1,\ell)c(1,\ell) - h^{\text{left}}(1)s(1,\ell)$;
11 $\quad\quad \gamma_\ell \leftarrow 1$; `// Initialize routine-local scaling factor`
12 $\quad\quad$ Solve robustly $\boldsymbol{R}_\ell \boldsymbol{x}(:,\ell) = \gamma_\ell \boldsymbol{b}(:,\ell)$; `// DLATRS`
13 $\quad\quad \alpha_\ell \leftarrow \gamma_\ell \beta_\ell$; `// Update total scaling`
14 $\quad$ **return** $\boldsymbol{\alpha}, \boldsymbol{X}$;

computes a scaling factor $\xi \in (0, 1]$ such that the linear update $\xi\boldsymbol{B} - \boldsymbol{H}(\xi\boldsymbol{X})$ cannot overflow. Then line 6 computes the column-wise scaling factors required for the first linear update (line 8). The upper bounds are rescaled to account for a consistent scaling of $\boldsymbol{B}$ and $\boldsymbol{X}$. After rescaling $\boldsymbol{X}$ (line 9), $\boldsymbol{B}$ and $\boldsymbol{X}$ are consistently scaled. Line 12 computes the column-wise scaling factors required for the second linear update (line 15). As line 14 applies the computed scaling factors prior to this linear update, the linear update itself can safely be implemented with a call to DGEMM. Line 17 computes the scaling factors necessary for the third linear update (line 19). The scaling of the final output $\boldsymbol{B}$ is $\boldsymbol{\delta}$ and corresponds to the product of all scaling factors.

The application of the Givens rotations in line 10 is not guarded by overflow protection logic, but can result in growth that possibly exceeds the overflow threshold. Since the evaluation of overflow protection logic is expensive compared to the cheap Givens transformations, our software lowers the true overflow threshold Ω by some safety margin. This safety margin is set to the tile height b_r, which overestimates the maximum growth possible by Givens transformations within a tile. Working with Ω/b_r is cheap to compute and guarantees that the Givens transformations in line 10 do not trigger overflow.

Robust backtransform. The robust backward substitution returns segment-wise scaled solution vectors. It remains to compute consistently scaled solutions and backtransform these. Recall that the partitioning into tile rows divides the vector evenly into N segments of length b_r and that $n = Nb_r$. The robust counterpart of (5) is then

$$\boldsymbol{y}_\ell = \boldsymbol{G}_{n,\ell}^T \left(\ldots \left(\boldsymbol{G}_{2,\ell}^T \alpha_{\min}^{-1} \begin{bmatrix} \frac{\alpha_{\min}}{\alpha_1}\boldsymbol{x}_\ell(1 : b_r) \\ \vdots \\ \frac{\alpha_{\min}}{\alpha_N}\boldsymbol{x}_\ell(n - b_r + 1 : n) \end{bmatrix} \right) \right). \tag{7}$$

ALGORITHM 10: Robust linear tile update

Data: $\boldsymbol{H} \in \mathbb{R}^{m\times k}$, $\boldsymbol{X} \in \mathbb{R}^{k\times n}$, $\boldsymbol{B} \in \mathbb{R}^{m\times n}$, cross-over columns $\tilde{\boldsymbol{R}} = [\tilde{\boldsymbol{r}}_1, \ldots, \tilde{\boldsymbol{r}}_n] \in \mathbb{R}^{m\times n}$,
$\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_n]^T \in \mathbb{R}^n$, Givens rotations computed by TiledReduce and stored as
$\boldsymbol{C} = [c(j,\ell)] \in \mathbb{R}^{k\times n}$ and $\boldsymbol{S} = [s(j,\ell)] \in \mathbb{R}^{k\times n}$
Result: (6) for $\ell = 1, \ldots, n$

1 **function** RUpdate($\boldsymbol{H}, \tilde{\boldsymbol{R}}, \boldsymbol{\alpha}, \boldsymbol{X}, \boldsymbol{\Lambda}, \boldsymbol{C}, \boldsymbol{S}, \boldsymbol{\beta}, \boldsymbol{B}$)
2 Allocate $\boldsymbol{\delta} = [\delta_1, \ldots, \delta_n] \in (0,1]^n$;
3 **for** $\ell \leftarrow 1 : n$ **do**
4 $\boldsymbol{t} \leftarrow \boldsymbol{h}(1:m, 1)$; $t(m) \leftarrow t(m) - \lambda_\ell$;
5 $\gamma_\ell \leftarrow \min\{\alpha_\ell, \beta_\ell\}$;
// Compute a scaling factor such that DAXPY in line 8 cannot overflow
6 $\xi_\ell \leftarrow$ ProtectUpdate $\left(\left\|\frac{\gamma_\ell}{\beta_\ell}\boldsymbol{b}(1:m,\ell)\right\|_\infty, \|\boldsymbol{t}\|_\infty, \left|\frac{\gamma_\ell}{\alpha_\ell}s(1,\ell)x(1,\ell)\right|\right)$;
7 $\delta_\ell \leftarrow \xi_\ell\gamma_\ell$; // Update global scaling
8 $\boldsymbol{b}(1:m,\ell) \leftarrow \frac{\delta_\ell}{\beta_\ell}\boldsymbol{b}(1:m,\ell) - \boldsymbol{t}\left(\frac{\delta_\ell}{\beta_\ell}s(1,\ell)x(1,\ell)\right)$; // Shift-specific DAXPY
9 Copy $\boldsymbol{Z} \leftarrow \boldsymbol{X}\,\mathrm{diag}\left(\frac{\delta_1}{\alpha_1}, \ldots, \frac{\delta_n}{\alpha_n}\right)$;
10 Apply Givens rotations to $\boldsymbol{Z}$ as in lines 7–13 of Algorithm 6;
11 **for** $\ell \leftarrow 1 : n$ **do**
// Compute a scaling factor such that DGEMM in line 15 cannot overflow
12 $\xi_\ell \leftarrow$ ProtectUpdate$(\|\boldsymbol{b}(1:m,\ell)\|_\infty, \|\boldsymbol{H}\|_\infty, \|\boldsymbol{z}(1:k-1,\ell)\|_\infty)$;
13 $\delta_\ell \leftarrow \delta_\ell\xi_\ell$; // Update global scaling
14 $\boldsymbol{b}(1:m,\ell) \leftarrow \xi_\ell\boldsymbol{b}(1:m,\ell)$; $\boldsymbol{z}(1:k,\ell) \leftarrow \xi_\ell\boldsymbol{z}(1:k,\ell)$; // Apply the scaling factors
15 $\boldsymbol{B} \leftarrow \boldsymbol{B} - \boldsymbol{H}(1:m, 2:k)\boldsymbol{Z}(1:k-1, 1:n)$; // DGEMM
16 **for** $\ell \leftarrow 1 : n$ **do**
// Compute a scaling factor such that DAXPY in line 19 cannot overflow
17 $\xi_\ell \leftarrow$ ProtectUpdate$(\|\boldsymbol{b}(1:m,\ell)\|_\infty, \|\tilde{\boldsymbol{r}}_\ell\|_\infty, |z(k,\ell)|)$;
18 $\delta_\ell \leftarrow \delta_\ell\xi_\ell$; // Update global scaling
19 $\boldsymbol{b}(1:m,\ell) \leftarrow \xi_\ell\boldsymbol{b}(1:m,\ell) - \tilde{\boldsymbol{r}}_\ell\,(\xi_\ell z(k,\ell))$; // Shift-specific DAXPY
20 **return** $\boldsymbol{\delta}$, $\boldsymbol{B}$;

The application of the Givens rotations can exceed the overflow threshold. To avoid overflow, Henry [1994, Algorithm 4] evaluates the maximum possible growth possible during the backtransform. If overflow *can* occur, the entire vector is rescaled prior to the backtransform. If the computation targets eigenvectors, an alternative strategy is possible. Eigenvectors are commonly normalized. The backtransform can be executed safely if the vector is normalized with respect to the Euclidean norm before the backtransform. The consistency scaling, the normalization, and the backtransform can be computed in two sweeps over the vector. Algorithm 11 RBacktransform gives the details. The lines 4–12 closely follow the LAPACK 3.9.0 routine DNRM2, which computes the Euclidean norm with scaling to avoid overflow. The lines 14–18 simultaneously normalize and backtransform an eigenvector. After the backtransform, the eigenvectors are still normalized.

By replacing all routines with their robust counterparts and adding segment-wise scaling factors, Algorithm 7 TiledSolve can be rendered robust. The resulting algorithm RobustTiledSolve is listed in Algorithm 12. This leads to DHSRQ3 listed in Algorithm 13, which solves shifted Hessenberg systems in a tiled, robust fashion. Note that the first part TiledReduce is untouched. Only the second part TiledSolve is replaced with its robust counterpart. Since the structure is identical to the non-robust version, task parallelism as introduced in Section 3.3 is valid for DHSRQ3 as well.

ALGORITHM 11: Consistency scaling, backtransform and normalization of a single eigenvector

Data: Givens rotations $G_{j,\ell}^T$ as in (2) computed by TiledReduce and stored as $\boldsymbol{C} = [c(j,\ell)] \in \mathbb{R}^{n\times m}$ and $\boldsymbol{S} = [s(j,\ell)] \in \mathbb{R}^{n\times m}$, matrix of scaling factors $\boldsymbol{\alpha} = [\alpha(i,\ell)] \in (0,1]^{N\times m}$, segment-wise scaled $\boldsymbol{X} \in \mathbb{R}^{n\times m}$, tile size b_r with $n = Nb_\mathrm{r}$

Result: Normalized (7) for $\ell \leftarrow 1, \ldots, m$

function RBacktransform($\boldsymbol{C}, \boldsymbol{S}, \boldsymbol{\alpha}, \boldsymbol{X}$)
  for $\ell \leftarrow 1 : m$ **do**
    $\alpha_\mathrm{min} \leftarrow \min_{1\le h\le N} \{\alpha(h,\ell)\}$; // Global scaling factor of current column
    $x_\mathrm{max} \leftarrow 0$;
    $t \leftarrow 0$;
    for $i \leftarrow 1 : n$ **do**
      $i_\mathrm{tile} \leftarrow \lceil i/b_\mathrm{r} \rceil$; // Index of current segment
      if $x(i,\ell) \neq 0$ **then**
        $x(i,\ell) \leftarrow \frac{\alpha_\mathrm{min}}{\alpha(i_{\mathrm{tile},\ell})} x(i,\ell)$; // Consistency scaling
        if $x_\mathrm{max} < |x(i,\ell)|$ **then**
          $t \leftarrow 1 + t\left(\frac{x_\mathrm{max}}{|x(i,\ell)|}\right)^2$; $x_\mathrm{max} \leftarrow |x(i,\ell)|$;
        else
          $t \leftarrow t + \left(\frac{|x(i,\ell)|}{x_\mathrm{max}}\right)^2$;
    $x_\mathrm{nrm} \leftarrow \alpha_\mathrm{min}^{-1} x_\mathrm{max}\sqrt{t}$; // Euclidean norm of current column
    // Simultaneous normalization and backtransform
    $\tau_1 \leftarrow x(1,\ell)/x_\mathrm{nrm}$;
    for $i \leftarrow 2 : n$ **do**
      $\tau_2 \leftarrow x(i,\ell)/x_\mathrm{nrm}$;
      $y(i-1,\ell) \leftarrow c(i)\tau_1 - s(i,\ell)\tau_2$;
      $\tau_1 \leftarrow c(i,\ell)\tau_2 + s(i,\ell)\tau_1$;
  return Y ; // $\|\boldsymbol{y}(:,\ell)\|_2 = 1$

4.2 Starting Vector and Convergence Test

It is well understood that the choice of the starting vector is crucial for both the convergence and the performance of inverse iteration (1). This is particularly true because the residual can increase by doing more than one iteration. Ipsen [1997, Sections 2.5, 2.6, 6.2] presents a comprehensive summary of work by Varah, Wilkinson and Peters on choosing a suited starting vector. Furthermore, an example demonstrating an increasing residual when more than one iteration is computed can be found in Section 5.4 in the same reference.

A standard choice for the starting vector is a scaled vector of ones $\boldsymbol{x}^{(0)} \leftarrow \rho[1,\ldots,1]^T$, $\rho > 0$ [Ipsen 1997, p. 259]. In LAPACK 3.9.0 the inverse iteration routine DHSEIN chooses $\rho = \|\boldsymbol{H}\|_\infty \epsilon$ assuming $\|\boldsymbol{H}\|_\infty > 0$. With this choice a single iteration of (1) most frequently leads to convergence [Ipsen 1997, p. 264]. If not, LAPACK tries other starting vectors orthogonal to previous choices rather than computing more iterations of (1). The convergence test is passed if $\|\boldsymbol{x}^{(1)}\|_1 > 0.1/\sqrt{n}$. LAPACK thereby follows Varah's [1968, p. 786] stopping criterion $\frac{\|\boldsymbol{x}^{(1)}\|_2}{\|\boldsymbol{x}^{(0)}\|_2} > \frac{1}{c\epsilon}$, where c is a problem-dependent constant and $\boldsymbol{H}$ is assumed to be normalized $\|\boldsymbol{H}\|_2 = 1$. Specifically, since LAPACK does not assume $\|\boldsymbol{H}\|_2 = 1$, we obtain using $\|\boldsymbol{x}^{(0)}\|_1 = n\|\boldsymbol{H}\|_\infty\epsilon$, $1/\|\boldsymbol{x}\|_2 \ge 1/\|\boldsymbol{x}\|_1$

ALGORITHM 12: Robust tiled backward substitution and backtransform

Data: Hessenberg matrix $\boldsymbol{H} \in \mathbb{R}^{n\times n}$, tile size b_r with $n = Nb_\mathrm{r}$, shifts $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_m]^T \in \mathbb{R}^m$, $\boldsymbol{B} \in \mathbb{R}^{n\times m}$, tile size b_c with $m = Mb_\mathrm{c}$, Givens rotations $\boldsymbol{G}^T_{j,\ell}$ computed with Algorithm 4 and stored as $\boldsymbol{C} \in \mathbb{R}^{n\times m}$ and $\boldsymbol{S} \in \mathbb{R}^{n\times m}$, cross-over columns $\tilde{\boldsymbol{R}}_\ell \in \mathbb{R}^{n\times N}$ for each shift $\ell = 1, \ldots, m$ computed with Algorithm 4.

Result: Solution $\boldsymbol{X} \in \mathbb{R}^{n\times m}$ to $\boldsymbol{HX} = \boldsymbol{B}\,\mathrm{diag}(\lambda_1, \ldots, \lambda_m)$

function RobustTiledSolve($\boldsymbol{H}, \boldsymbol{\Lambda}, \boldsymbol{C}, \boldsymbol{S}, (\tilde{\boldsymbol{R}}_1, \ldots, \tilde{\boldsymbol{R}}_m), \boldsymbol{B}$)
- $\boldsymbol{\alpha} \leftarrow$ ones(N, m) ; // Initialize all tile-local scaling factors with 1
- $\boldsymbol{X} \leftarrow \boldsymbol{B}$;
- **for** $\ell \leftarrow 1 : b_\mathrm{c} : m$ **do**
  - $j_\mathrm{tile} \leftarrow N$; // Index of current tile column
  - $\mathcal{L} \leftarrow \ell : \ell + b_\mathrm{c} - 1$; // Index range of current batch of shifts
  - **for** $j \leftarrow n - b_\mathrm{r} + 1 : -b_\mathrm{r} : 1$ **do**
    - $\mathcal{J} \leftarrow j : j + b_\mathrm{r} - 1$; // Index range of current tile column
    - $\mathcal{J}^- \leftarrow j - 1 : j + b_\mathrm{r} - 2$; // Tile column index range shifted by 1
    - Partition $\left[\underbrace{\boldsymbol{h}^\mathrm{left}}_{b_\mathrm{r}\times 1} \quad \underbrace{\boldsymbol{H}^\mathrm{diag}}_{b_\mathrm{r}\times(b_\mathrm{r}-1)} \right] \leftarrow \boldsymbol{H}(\mathcal{J}, \mathcal{J}^-)$;
    - Pack $\tilde{\boldsymbol{R}}^\mathrm{right} \leftarrow \left[\tilde{\boldsymbol{r}}_\ell(\mathcal{J}, j_\mathrm{tile}), \ldots, \tilde{\boldsymbol{r}}_{\ell+b_\mathrm{c}-1}(\mathcal{J}, j_\mathrm{tile})\right]$;
    - $\boldsymbol{\alpha}(j_\mathrm{tile}, \mathcal{L}), \boldsymbol{X}(\mathcal{J}, \mathcal{L}) \leftarrow$ RSolve($\boldsymbol{H}^\mathrm{diag}, \boldsymbol{h}^\mathrm{left}, \tilde{\boldsymbol{R}}^\mathrm{right}, \boldsymbol{\Lambda}(\mathcal{L}), \boldsymbol{C}(\mathcal{J}, \mathcal{L}), \boldsymbol{S}(\mathcal{J}, \mathcal{L}), \boldsymbol{\alpha}(j_\mathrm{tile}, \mathcal{L}), \boldsymbol{X}(\mathcal{J}, \mathcal{L})$);
    - **for** $i \leftarrow j - b_\mathrm{r} : -b_\mathrm{r} : 1$ **do**
      - $\mathcal{I} \leftarrow i : i + b_\mathrm{r} - 1$; // Index range of current tile row
      - $i_\mathrm{tile} \leftarrow (i + b_\mathrm{r} - 1)/b_\mathrm{r}$; // Index of current tile row
      - Pack $\tilde{\boldsymbol{R}}^\mathrm{right} \leftarrow \left[\tilde{\boldsymbol{r}}_\ell(\mathcal{I}, j_\mathrm{tile}), \tilde{\boldsymbol{r}}_{\ell+1}(\mathcal{I}, j_\mathrm{tile}), \ldots, \tilde{\boldsymbol{r}}_{\ell+b_\mathrm{c}-1}(\mathcal{I}, j_\mathrm{tile})\right]$;
      - **if** $i + b_\mathrm{r} = j$ **then**
        - $\boldsymbol{\alpha}(i_\mathrm{tile}, \mathcal{L}), \boldsymbol{X}(\mathcal{I}, \mathcal{L}) \leftarrow$ RUpdate($\boldsymbol{H}(\mathcal{I}, \mathcal{J}^-), \tilde{\boldsymbol{R}}^\mathrm{right}$, $\boldsymbol{\alpha}(j_\mathrm{tile}, \mathcal{L}), \boldsymbol{X}(\mathcal{J}, \mathcal{L}), \boldsymbol{\Lambda}(\mathcal{L}), \boldsymbol{C}(\mathcal{J}, \mathcal{L}), \boldsymbol{S}(\mathcal{J}, \mathcal{L}), \boldsymbol{\alpha}(i_\mathrm{tile}, \mathcal{L}), \boldsymbol{X}(\mathcal{I}, \mathcal{L})$);
      - **else**
        - $\boldsymbol{\alpha}(i_\mathrm{tile}, \mathcal{L}), \boldsymbol{X}(\mathcal{I}, \mathcal{L}) \leftarrow$ RUpdate($\boldsymbol{H}(\mathcal{I}, \mathcal{J}^-), \tilde{\boldsymbol{R}}^\mathrm{right}$, $\boldsymbol{\alpha}(j_\mathrm{tile}, \mathcal{L}), \boldsymbol{X}(\mathcal{J}, \mathcal{L}), \boldsymbol{0}, \boldsymbol{C}(\mathcal{J}, \mathcal{L}), \boldsymbol{S}(\mathcal{J}, \mathcal{L}), \boldsymbol{\alpha}(i_\mathrm{tile}, \mathcal{L}), \boldsymbol{X}(\mathcal{I}, \mathcal{L})$);
    - $j_\mathrm{tile} \leftarrow j_\mathrm{tile} - 1$;
  - $\mathcal{J} \leftarrow 1 : b_\mathrm{r}$;
  - Pack $\tilde{\boldsymbol{R}}^\mathrm{right} \leftarrow \left[\tilde{\boldsymbol{r}}_\ell(\mathcal{J}, j_\mathrm{tile}), \ldots, \tilde{\boldsymbol{r}}_{\ell+b_\mathrm{c}-1}(\mathcal{J}, j_\mathrm{tile})\right]$;
  - $\boldsymbol{\alpha}(1, \mathcal{L}), \boldsymbol{X}(\mathcal{J}, \mathcal{L}) \leftarrow$ RSolve($\boldsymbol{H}(\mathcal{J}, 1 : b_\mathrm{r} - 1), \boldsymbol{0}, \boldsymbol{\Lambda}(\mathcal{L}), \tilde{\boldsymbol{R}}^\mathrm{right}$, $\boldsymbol{C}(\mathcal{J}, \mathcal{L}), \boldsymbol{S}(\mathcal{J}, \mathcal{L}), \boldsymbol{\alpha}(1, \mathcal{L}), \boldsymbol{X}(\mathcal{J}, \mathcal{L})$);
  - $\boldsymbol{X}(:, \mathcal{L}) \leftarrow$ RBacktransform($\boldsymbol{C}(:, \mathcal{L}), \boldsymbol{S}(:, \mathcal{L}), \boldsymbol{\alpha}(:, \mathcal{L}), \boldsymbol{X}(:, \mathcal{L})$);
- **return** $\boldsymbol{X}$;

and $\|\boldsymbol{x}\|_2 \geq \|\boldsymbol{x}\|_1/\sqrt{n}$

$$\frac{\|\boldsymbol{x}^{(1)}\|_2}{\|\boldsymbol{x}^{(0)}\|_2} \geq \frac{\|\boldsymbol{x}^{(1)}\|_1}{\sqrt{n}\|\boldsymbol{x}^{(0)}\|_1} = \frac{\|\boldsymbol{x}^{(1)}\|_1}{\sqrt{n}n\|\boldsymbol{H}\|_\infty\epsilon} > \frac{1}{10n^2\|\boldsymbol{H}\|_\infty\epsilon}.$$

Our inverse iteration solver chooses $\rho = \|\boldsymbol{H}\|_\infty\epsilon$. Since the norm $\|\boldsymbol{x}^{(1)}\|_2$ is readily available after the backward substitution phase due to the consistency scaling, the check $\|\boldsymbol{x}^{(1)}\|_2 > 0.1/\sqrt{n}$ lends

ALGORITHM 13: Robust tiled simultaneous solution of shifted Hessenberg systems with real shifts

Data: Hessenberg matrix $\boldsymbol{H} \in \mathbb{R}^{n\times n}$, shifts $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_m]^T \in \mathbb{R}^m$, $\boldsymbol{B} \in \mathbb{R}^{n\times m}$
Result: Solution $\boldsymbol{X} \in \mathbb{R}^{n\times m}$ to $\boldsymbol{HX} = \boldsymbol{B}\,\mathrm{diag}(\lambda_1, \ldots, \lambda_m)$

```
function DHSRQ3(H, Λ, B)
    C, S, (R̃_1, ..., R̃_m) ← TiledReduce(H, Λ);
    X ← RobustTiledSolve(H, Λ, C, S, (R̃_1, ..., R̃_m), B);
    return X;
```

ALGORITHM 14: Robust tiled inverse iteration for real eigenvalues

Data: Hessenberg matrix $\boldsymbol{H} \in \mathbb{R}^{n\times n}$, eigenvalues $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_m]^T \in \mathbb{R}^m$
Result: $\boldsymbol{X} \in \mathbb{R}^{n\times m}$ such that $\boldsymbol{HX} = \boldsymbol{X}\,\mathrm{diag}(\lambda_1, \ldots, \lambda_m)$

```
1  function DHSRQ3IN(H, Λ)
2      Allocate X ∈ R^{n×m};
3      Choose the starting vector x^(0) ← 1/(ε‖H‖_∞) [1, ..., 1]^T;
4      Initialize X^(0) with m repeated copies of x^(0);
5      while not converged do
6          Compute robustly X^(1) ← DHSRQ3(H, Λ, X^(0));
7          Split X^(1) into converged eigenvectors X_c and non-converged eigenvectors X_d ∈ R^{n×k};
8          Append X_c to X;
9          if k = 0 then
10             Go to line 13;
11         Choose a new starting vector x^(0) orthogonal to previous choices;
12         Initialize X^(0) with k repeated copies of x^(0);
13     Sort the columns of X such that the order matches Λ;
14     return X;
```

itself to a quick convergence test. This convergence test is in line with Varah's stopping criterion where $c = 10n\|\boldsymbol{H}\|_\infty$. The information that an eigenvector has not converged can, for example, be propagated by setting the eigenvector to zero.

The decision on a starting vector and the convergence test combined with the robust backward substitution leads to the inverse iteration routine HSRQ3IN, a routine for computing individual eigenvectors simultaneously by inverse iteration. The core of the routine is the robust, tiled solver for shifted Hessenberg systems introduced as DHSRQ3. Algorithm 14 lists the inverse iteration solver. Following LAPACK, only a single iteration of (1) is computed for a given starting vector. After this single iteration, converged eigenvectors are separated from non-converged eigenvectors. New starting vectors are tried for the non-converged eigenvectors.

4.3 Complex Shifts

The RQ factorization $\boldsymbol{H} - \lambda\boldsymbol{I} = \boldsymbol{RQ}$ has complex factors $\boldsymbol{R}$ and $\boldsymbol{Q}$ if λ is complex. The backward substitution with $\boldsymbol{R}$ then relies fully on complex arithmetic. Due to costly multiplications of complex scalars with complex vectors, Henry preferred the UL factorization over the RQ factorization for complex shifts, see Section 2.2. This section presents two techniques which allow extending the inverse iteration solver DHSRQ3IN to support complex shifts at a reasonable computational cost. The first technique chooses the complex Givens rotation such that the reduction phase avoids multiplications of complex scalars with complex vectors. The second technique lowers the cost of

the backward substitution by exploiting that most entries of $\boldsymbol{H} - \lambda \boldsymbol{I}$ are real in spite of a complex shift.

The reduction phase requires complex Givens rotations to compute the unitary $\boldsymbol{Q}$ factor. This paper adopts the Givens rotations applied by Beattie et al. [2012, p. 6]

$$\boldsymbol{G}_{j,\ell}^H = \boldsymbol{I}_{j-2} \oplus \begin{bmatrix} c(j,\ell) & -s(j,\ell) \\ s(j,\ell) & \bar{c}(j,\ell) \end{bmatrix} \oplus \boldsymbol{I}_{n-j} \in \mathbb{C}^{n \times n}, \tag{8}$$

where $c \in \mathbb{C}$ and $s \in \mathbb{R}$. Thereby most of the reduction phase corresponds to mixed real-complex multiplications. In view of the analysis of Givens rotations in floating-point arithmetic by Bindel et al. [2002], our implementation of this Givens rotation is numerically robust and takes care of underflow and overflow.

Next we discuss the changes to Algorithm 2 ReduceDiag and Algorithm 3 ReduceOffdiag. The columns of the $\boldsymbol{R}$ factor are complex and so are the cross-over columns. We store a complex vector $\boldsymbol{v} = \boldsymbol{v}^{\mathrm{re}} + i\boldsymbol{v}^{\mathrm{im}}$ as adjacent columns $[\boldsymbol{v}^{\mathrm{re}} \quad \boldsymbol{v}^{\mathrm{im}}]$. Using the complex Givens rotation (8), the complex version of line 5 in Algorithm 3 ReduceOffdiag

$$\underbrace{\boldsymbol{v}(1:k)}_{\mathbb{C}^k} \leftarrow \underbrace{c(j,\ell)}_{\mathbb{C}} \underbrace{\boldsymbol{h}(1:k,j)}_{\mathbb{R}^k} + \underbrace{s(j,\ell)}_{\mathbb{R}} \underbrace{\boldsymbol{v}(1:k)}_{\mathbb{C}^k}$$

can be realized as

$$\begin{aligned} \boldsymbol{v}^{\mathrm{re}}(1:k) &\leftarrow c^{\mathrm{re}}(j,\ell)\boldsymbol{h}(1:k,j) + s(j,\ell)\boldsymbol{v}^{\mathrm{re}}(1:k) \\ \boldsymbol{v}^{\mathrm{im}}(1:k) &\leftarrow c^{\mathrm{im}}(j,\ell)\boldsymbol{h}(1:k,j) + s(j,\ell)\boldsymbol{v}^{\mathrm{im}}(1:k). \end{aligned}$$

The lines 6–7 in ReduceDiag can be realized in a similar fashion. The flop count for the complex versions of ReduceDiag and ReduceOffdiag doubles compared to their real counterparts and is $3n^2 + O(n)$ and $6nk + O(n)$, respectively.

Next we discuss the changes to the backward substitution phase. Systems with a complex shift yield a complex solution. Aiming for a robust backward substitution, every complex solution vector is associated with a single scaling factor. In other words, the real and the imaginary part are scaled alike. Then a robust Solve task addresses $\boldsymbol{R}_\ell \boldsymbol{y}_\ell = \gamma_\ell \boldsymbol{b}_\ell$ where $\gamma_\ell \in (0,1]$ and all other quantities are complex. If the (small) triangular system matrices $\boldsymbol{R}_\ell$ are computed explicitly, a conversion to a complex datatype allows the robust solution of this system with a call to ZLATRS for every complex right-hand side. ZLATRS is available in LAPACK 3.9.0. and is the complex counterpart of DLATRS, see Section 4.1. Thereby Algorithm 9 naturally generalizes to complex arithmetic. The flop count is $15n^2 + O(n)$ per right-hand side.

An analysis of the complex counterpart of the linear update (6)

$$\begin{aligned} \delta_\ell^{-1}\boldsymbol{b}_\ell(\mathcal{J}_1) \leftarrow \beta_\ell^{-1}\boldsymbol{b}_\ell(\mathcal{J}_1) - \Big[\boldsymbol{h}(\mathcal{J}_1, j-1) - \lambda_\ell \boldsymbol{e}_{j-1} \quad \boldsymbol{H}(\mathcal{J}_1, j:k-2) \quad \tilde{\boldsymbol{r}}_\ell(\mathcal{J}_1)\Big] \\ \left(\boldsymbol{G}_{k-1,\ell}^H(\mathcal{J}_2^+, \mathcal{J}_2^+) \ldots \boldsymbol{G}_{j,\ell}^H(\mathcal{J}_2^+, \mathcal{J}_2^+) \begin{bmatrix} 0 \\ \alpha_\ell^{-1}\boldsymbol{x}_\ell(\mathcal{J}_2) \end{bmatrix}\right) \end{aligned}$$

reveals the potential for mixed real-complex arithmetic. Analogously to the real linear update, the block structure of the system matrix suggests three block operations. The first block $\boldsymbol{h}(\mathcal{J}_1, j-1) - \lambda_\ell \boldsymbol{e}_{j-1}$ and the third block $\tilde{\boldsymbol{r}}_\ell(\mathcal{J}_1)$ issue multiplications of a complex scalar with a complex vector and can be emulated using real arithmetic. The second block $\boldsymbol{H}(\mathcal{J}_1, j:k-2)$ is real and requires the multiplication of a real matrix with a complex vector. If complex quantities are stored in interleaved storage, that is, the real and imaginary parts of a complex vector are stored in adjacent columns, the multiplication of a real matrix $\boldsymbol{H}$ and a complex vector $\boldsymbol{z} = \boldsymbol{u} + i\boldsymbol{v}$ can be realized as $\boldsymbol{H}[\boldsymbol{u} \quad \boldsymbol{v}]$. Hence, the computationally expensive part of the linear update can be realized with a wide DGEMM

Table 1. Flops Approximations for a Single Shift/Eigenvector Segment of Length n and Assuming Square Tiles

Routine	Real eigenvector	Complex eigenvector
ReduceDiag	$1.5n^2 + O(n)$	$3n^2 + O(n)$
ReduceOffdiag	$3n^2 + O(n)$	$6n^2 + O(n)$
Solve	$3.5n^2 + O(n)$	$15n^2 + O(n)$
Update	$2n^2 + O(n)$	$4n^2 + O(n)$
Backtransform	$6(n-1)$	$20(n-1)$

Flops due to overflow protection (norm computations, overflow protection logic, numerical scaling) are disregarded.

operation if many right-hand sides are computed simultaneously. The flop count of a linear update with an m-by-n matrix $\boldsymbol{H}$ is approximately $4mn + O(m+n)$ per complex right-hand side.

The complex solution $\boldsymbol{x}_\ell$ is backtransformed analogously to (5) using complex arithmetic. The flop count sums to $20n + O(1)$. An overview of all flop counts is listed in Table 1. The computational cost per column is approximately the same for a real eigenvector and a complex eigenvector comprising two columns.

Coupling the complex routines of all task types results in ZHSRQ3, the complex counterpart of DHSRQ3. Analogously to LAPACK, the starting vector and the convergence criterion are chosen identically for real and complex eigenvalues. ZHSRQ3 allows generalizing the inverse iteration routine DHSRQ3IN to handle *any* selection of eigenvalues, see Algorithm 15. The selection of eigenvalues is split into real eigenvalues and complex eigenvalues. This requires additional tracking of the affiliation between computed eigenvectors and selected eigenvalues. Recall that complex eigenvectors occur in complex conjugate pairs. Hence, for each selected complex conjugate pair of eigenvalues, only one eigenvector has to be computed. The other one can be obtained for free by complex conjugation. HSRQ3IN assumes that the provided eigenvalues exploit this.

The real/complex eigenvectors are computed by successive calls to the real/complex version of HSRQ3IN. The successive computation of real/complex eigenvectors is justified because the storage requirement quickly limits the problem sizes solvable with HSRQ3IN. An analysis of the storage requirement is given in the next section.

4.4 Storage Requirement Analysis

Algorithm 14 records the Givens rotations and the cross-over columns during the reduction phase. If the Hessenberg matrix is n-by-n, n Givens rotations including the first padded entry have to be stored for every eigenvector. The storage requirement of the cross-over columns depends on the tiling of $\boldsymbol{H}$. The partitioning of $\boldsymbol{H}$ into an N-by-N grid of $b_\mathrm{r} \times b_\mathrm{r}$ tiles requires the storage of N (potentially complex) cross-over columns. The storage requirement of the cross-over columns sums to $\sum_{i=1}^{N}(ib_\mathrm{r}) = O(nN)$ for every eigenvector. If m columns (1 column per real or 2 columns per complex eigenvector) are computed, the storage of the cross-over columns is $O(nNm)$ and can quickly exceed the memory available on a compute node. This problem can be addressed by computing the eigenvectors in groups. Algorithm 15 realizes this. Only workspace necessary for storing the cross-over columns and the Givens rotations of a single group has to be allocated. Once a group has been computed, the workspace can be reused for the next group. Since complex eigenvectors and cross-over column are stored in interleaved storage, the storage requirement is doubled compared to the real computation. Line 4 calculates the group size for the complex case requiring two columns per cross-over column. To fully harness the available workspace, the real computation doubles the group size (lines 6–8).

ALGORITHM 15: Robust tiled inverse iteration

Data: Hessenberg matrix $\boldsymbol{H} \in \mathbb{R}^{n\times n}$, tile size b_r with $n = Nb_r$, eigenvalues $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_m]^T \in \mathbb{C}^m$ such that only one complex eigenvalue of a complex conjugate pair is included in $\boldsymbol{\Lambda}$
Result: $\boldsymbol{X} \in \mathbb{C}^{n\times m}$ such that $\boldsymbol{HX} = \boldsymbol{X}\,\mathrm{diag}(\lambda_1, \ldots, \lambda_m)$

```
function HSRQ3IN(H, Λ)
    Sort Λ into Λ_sorted = [Λ_real  Λ_complex] so that the first m_1 eigenvalues are real;
    Define the maximum workspace size for the cross-over columns as w_max;
    g ← ⌊w_max/(2nN)⌋;                                              // Group size
    Allocate workspaces for the Givens rotations and the cross-over columns;
    for ℓ ← 1 : 2g : m_1 do
        L ← ℓ : max{ℓ + 2g, m};
        Task-parallel X(L) ← DHSRQ3IN(H, Λ_sorted(L));              // Real eigenvectors
    for ℓ ← m_1 + 1 : g : m do
        L ← ℓ : max{ℓ + g, m_2};
        Task-parallel X(L) ← ZHSRQ3IN(H, Λ_sorted(L));              // Complex eigenvectors
    Sort the columns of X so that the order matches Λ;
    return X;
```

5 NUMERICAL EXPERIMENTS

This section describes how the numerical experiments were set up and executed and presents the results.

5.1 Execution Environment

Hardware. The experiments are run on an Intel Xeon Gold 6132 (Skylake) node where dynamic frequency scaling is enabled. This node has 2 NUMA islands with 14 cores each. In double-precision arithmetic the theoretical peak performance is 83.2 Gflops per core and 2329.6 Gflops per node. The L1 data cache and L2 cache are 32 KB and 1 MB, respectively, and the shared L3 cache is 19.25 MB. The available memory is 192 GB RAM. The memory bandwidth was measured at 12.7 GB/s for one core and 162 GB/s for a full node using the STREAM triad benchmark.

Software and configuration. The software is built with the Intel compiler 19.0.1.144 where the optimization level is set to `-O2`, AVX-512 instructions are enabled and interprocedural optimizations `-ipo` are activated. We link against the MKL 2019.1.144 BLAS implementation. OpenMP threads are bound to physical processing units by setting `KMP_AFFINITY` to `compact`.

In the following we describe the routines and their configuration used in the numerical experiments. The first routine targets shifted Hessenberg systems and solves $(\boldsymbol{H} - \lambda_\ell \boldsymbol{I})\boldsymbol{x}_\ell = \alpha_\ell \boldsymbol{e}$. The matrix $\boldsymbol{H}$ is real and upper Hessenberg, the shift λ_ℓ is real or complex, $\boldsymbol{e}$ is the vector with all ones and $\alpha_\ell \in (0, 1]$ is a scaling factor. The next four routines target the computation of eigenvectors by inverse iteration. The routines are supplied with $\|\boldsymbol{H}\|_\infty \epsilon \boldsymbol{e}$ as the starting vector, see Section 4.2. This starting vector leads to convergence in one iteration in all of our numerical experiments.

- {Z,D}HSRQ3. This robust routine was introduced in this work and solves $(\boldsymbol{H} - \lambda_\ell \boldsymbol{I})\boldsymbol{x}_\ell = \alpha_\ell \boldsymbol{e}$. It generalizes the RQ approach originally proposed by Henry. The reduction phase records the Givens rotations necessary to compute the RQ factorizations for all shifts. The backward substitution phase utilizes level-3 BLAS for the linear updates.
- RQIN (Henry). This routine extends the shifted Hessenberg system solver by Henry [1994, Algorithm 2] to the computation of eigenvectors. It solves $(\boldsymbol{H} - \lambda_\ell \boldsymbol{I})\boldsymbol{x}_\ell = \|\boldsymbol{H}\|_\infty \epsilon \boldsymbol{e}$ through an RQ factorization for every eigenvalue and normalizes every eigenvector after the

backtransform. The core of this routine corresponds to Algorithm 1. While Henry uses the RQ decomposition only for real shifts, the numerical experiments use the RQ approach both for real and complex shifts. The matrix $\boldsymbol{H}$ is not overwritten.

- ULIN (Henry). This routine solves $(\boldsymbol{H} - \lambda_\ell \boldsymbol{I})\boldsymbol{x}_\ell = \|\boldsymbol{H}\|_\infty \epsilon \boldsymbol{e}$ through an UL factorization for every shift and, in a final step, normalizes the computed eigenvectors. Henry [1994, Section 4] introduced the UL approach for complex shifts to avoid costly complex-complex multiplications. Since the UL approach overwrites $\boldsymbol{H}$, the original matrix $\boldsymbol{H}$ is copied when more than one system is solved.
- DHSEIN. LAPACK 3.9.0 contains the driver routine DHSEIN for successively computing selected left and/or right eigenvectors of a real upper Hessenberg matrix. DHSEIN calls DLAEIN for computing a single eigenvector by inverse iteration. The routine is supplied with $\epsilon \boldsymbol{e}$ as the user-defined starting vector (INITV = 'U'). For each shift, the matrix $\boldsymbol{B} = \boldsymbol{H} - \lambda \boldsymbol{I}$ is explicitly constructed in a workspace. Then $\boldsymbol{B}\boldsymbol{x} = \alpha(\|\boldsymbol{H}\|_\infty \epsilon \boldsymbol{e})$ is solved through an LU factorization with partial pivoting. In all numerical experiments conducted here, this initial guess leads to convergence in the first iteration. In other words, DHSEIN effectively solves a single shifted Hessenberg system through an LU decomposition with partial pivoting for each eigenvector.
- HSRQ3IN. This driver routine, listed in Algorithm 15, splits real and complex eigenvalues and computes the corresponding eigenvectors by successive calls to DHSRQ3IN and ZSRQ3IN. It solves $(\boldsymbol{H} - \lambda_\ell \boldsymbol{I})\boldsymbol{x}_\ell = \alpha_\ell(\|\boldsymbol{H}\|_\infty \epsilon \boldsymbol{e})$ and normalizes the eigenvectors with respect to the Euclidean norm before the backtransform.

5.2 Test Problems

The numerical experiments use two test problems. The **first test problem** is designed to have known, well-separated eigenvalues and computes the corresponding eigenvectors. This experiment controls the ratio of real/complex eigenvalues and allows us to examine the cost of complex arithmetic. For this purpose, a quasi-triangular matrix $\boldsymbol{T} \in \mathbb{R}^{n \times n}$ is constructed where the eigenvalues are placed as 1-by-1 or 2-by-2 blocks on the diagonal of $\boldsymbol{T}$. If the k-th eigenvalue is real, then the 1-by-1 diagonal block is $t(k,k) = k$. Complex eigenvalues occur in complex conjugate pairs and correspond to 2-by-2 blocks. Such a 2-by-2 block is set to

$$\boldsymbol{T}(k : k+1, k : k+1) = \begin{bmatrix} k & k \\ -k & k \end{bmatrix} \in \mathbb{R}^{2\times 2} \tag{9}$$

and corresponds to the eigenvalues $k + ik$ and $k - ik$. This choice of diagonal blocks ensures that all eigenvalues are well-separated. In particular, the case with 100% real eigenvalues yields an upper triangular matrix with eigenvalues $1, 2, \ldots, n$. Assuming that n is even, the case with 100% complex eigenvalues yields a matrix with only 2-by-2 blocks on the diagonal and eigenvalues $1 \pm i, 3(1 \pm i), \ldots, (n-1)(1 \pm i)$. The remaining superdiagonal entries are random in $(0, 1]$.

The matrix $\boldsymbol{T}$ is then transformed into a Hessenberg matrix through an orthogonal similarity transformation. For this purpose, two orthogonal transformations are applied. First, a random Householder matrix is constructed, $\boldsymbol{Q}_0 = \boldsymbol{I} - 2\boldsymbol{v}\boldsymbol{v}^T$, where $\boldsymbol{v}$ is a random unit norm vector. By applying the (symmetric) Householder matrix, a dense matrix $\boldsymbol{A} = \boldsymbol{Q}_0 \boldsymbol{T} \boldsymbol{Q}_0$ is computed. Second, $\boldsymbol{A}$ is reduced to Hessenberg form through the LAPACK routine DGEHRD $\boldsymbol{H}_1 \leftarrow \boldsymbol{Q}_1^T \boldsymbol{A} \boldsymbol{Q}_1$. Together, $\boldsymbol{H}_1$ is given by $\boldsymbol{H}_1 \leftarrow (\boldsymbol{Q}_0 \boldsymbol{Q}_1)^T \boldsymbol{T} (\boldsymbol{Q}_1 \boldsymbol{Q}_0)$. The numerical routines receive the exact eigenvalues as input parameter.

The **second test problem** solves shifted Hessenberg systems and aims at quantifying the overhead from overflow protection. Two systems are constructed: the "bad" system requires frequent numerical scaling system, whereas the "good" system never requires numerical scaling.

The bad system constructs the Hessenberg matrix $\boldsymbol{H}_2 = \boldsymbol{R}_2\boldsymbol{Q}_2 + \gamma\boldsymbol{I}$ where $\boldsymbol{R}_2 \in \mathbb{R}^{n\times n}$ and $\boldsymbol{Q}_2 \in \mathbb{R}^{n\times n}$ are given by

$$r_2(i,j) = \begin{cases} n-i+1 & i=j \\ -n & j>i\,, \\ 0 & i>j \end{cases} \quad q_2(i,j) = \begin{cases} -1 & (i-1=j) \text{ or } (i=1 \text{ and } j=n) \\ 0 & \text{otherwise.} \end{cases}$$

As an example, consider how $\boldsymbol{H}_2 - \gamma\boldsymbol{I}$ is constructed for $n = 6$ by

$$\boldsymbol{H}_2 - \gamma\boldsymbol{I} = \begin{bmatrix} 6 & 6 & 6 & 6 & 6 & -6 \\ -5 & 6 & 6 & 6 & 6 & 0 \\ & -4 & 6 & 6 & 6 & 0 \\ & & -3 & 6 & 6 & 0 \\ & & & -2 & 6 & 0 \\ & & & & -1 & 0 \end{bmatrix}$$

$$= \boldsymbol{R}_2\boldsymbol{Q}_2 = \begin{bmatrix} 6 & -6 & -6 & -6 & -6 & -6 \\ & 5 & -6 & -6 & -6 & -6 \\ & & 4 & -6 & -6 & -6 \\ & & & 3 & -6 & -6 \\ & & & & 2 & -6 \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ & -1 & 0 & 0 & 0 & 0 \\ & & -1 & 0 & 0 & 0 \\ & & & -1 & 0 & 0 \\ & & & & -1 & 0 \end{bmatrix}.$$

The numerical experiments solve $\boldsymbol{H}_2\boldsymbol{X} = \boldsymbol{E}\boldsymbol{\Gamma}$ where $\boldsymbol{E}$ is the matrix with all ones and $\boldsymbol{\Gamma} = \mathrm{diag}(2,\ldots,2)$. Hence, the same shifted system is solved repeatedly without exploiting that the shift is shared. Kjelgaard Mikkelsen [2020] has shown that the matrix $\boldsymbol{R}_2$ introduces quick growth to the solution vectors during the backward substitution.

The good system constructs the Hessenberg matrix $\boldsymbol{H}_3 = \boldsymbol{R}_3\boldsymbol{Q}_2 + \gamma\boldsymbol{I}$ where $\boldsymbol{R}_3 \in \mathbb{R}^{n\times n}$ is given by

$$r_3(i,j) = \begin{cases} n+1-i & i=j \\ \frac{1}{2} & j>i \\ 0 & i>j. \end{cases}$$

The numerical experiments solve $\boldsymbol{H}_3\boldsymbol{X} = \boldsymbol{E}\boldsymbol{\Gamma}$. Following Kjelgaard Mikkelsen [2020], the solution of $\boldsymbol{R}_3\boldsymbol{y} = \boldsymbol{e}$ with backward substitution never requires numerical scaling to avoid overflow.

5.3 Validation

The experiments are validated by comparing the relative normwise backward error with 10^{-12}. Specifically, the first test problem evaluates $\frac{\|\boldsymbol{H}_1\boldsymbol{x}-\lambda\boldsymbol{x}\|_F}{\|\boldsymbol{H}_1\|_F\|\boldsymbol{x}\|_F+|\lambda|\|\boldsymbol{x}\|_F}$ for each computed eigenvector. The second test problem solves the same system repeatedly. The evaluation computes $\frac{\|\boldsymbol{A}\boldsymbol{x}-\gamma\boldsymbol{e}\|_F}{\|\boldsymbol{A}\|_F\|\boldsymbol{x}\|_F+|\gamma|\|\boldsymbol{e}\|_F}$ only for a single vector, where $\boldsymbol{A}$ corresponds to $\boldsymbol{H}_2$ or $\boldsymbol{H}_3$, respectively.

5.4 Results

This section presents the results of four numerical experiments. The first experiment concerns the sequential runtimes of the inverse iteration solvers and compares the existing approaches DHSEIN (LAPACK), RQIN/ULIN (Henry) with HSRQ3IN introduced in this paper. The second experiment aims at identifying bottlenecks in the implementation and analyzes what fraction each computational phase contributes to the total runtime. The third experiment addresses the parallel scalability of HSRQ3IN. The fourth experiment quantifies the cost of robustness.

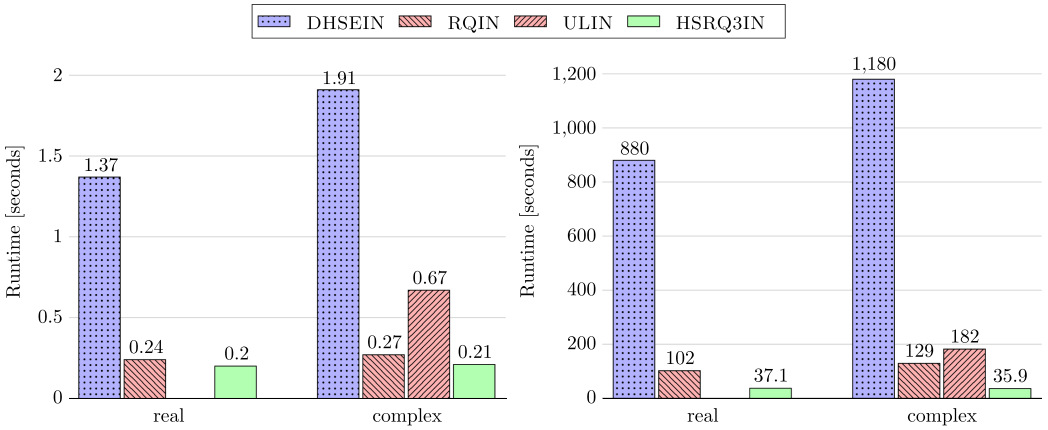

Fig. 5. Sequential runtimes on $\boldsymbol{H}_1$ computing two columns (left) and 1500 columns (right) for $n = 10000$.

Serial Comparison. The first experiment compares the sequential runtimes of the inverse iteration routines DHSEIN, RQIN, ULIN and HSRQ3IN on $\boldsymbol{H}_1$ where $n = 10000$. Numerical scaling is never triggered and a single iteration suffices to satisfy the convergence criterion for every eigenvector. Figure 5 (left) shows the runtimes for computing two columns, either two real vectors or a single complex vector (storing the real and the imaginary part in two columns). In the latter case, the UL approach overwrites the input matrix and, hence, does not require any data copies. For all solvers the computation of the complex vector is at least as expensive as the computation of two real vectors.

Figure 5 (right) compares the runtimes for computation of 1,500 columns, either 1,500 real vectors or 750 complex vectors. The timing of the UL approach includes the overhead of 749 copies of the input matrix. The runtime gap between DHSEIN and the RQ factorization-based solvers widens with an increasing number of right-hand sides. A large number of right-hand sides allows reusing data. Since the RQ decomposition-based solvers do not overwrite the input matrix, the computation may benefit from temporal locality. HSRQ3IN outperforms RQIN, which can be attributed to the matrix–matrix multiplications (level-3 BLAS) in the backward substitution phase. Complex eigenvectors are more expensive than real eigenvectors for DHSEIN and RQIN. HSRQ3IN, by contrast, performs similarly. The next experiment aims at investigating the underlying cause.

Analysis. This experiment decomposes the computational cost of HSRQ3IN and RQIN and thereby analyzes the ratio of the three computational phases (reduction, backward substitution, backtransform). The experiment setup is identical to the one in Figure 5 (right) and uses $\boldsymbol{H}_1$ with $n = 10000$ for the computation of either 1,500 real or 750 complex eigenvectors.

A run of RQIN on 1,500 real eigenvectors spends 46% of the runtime in the reduction phase (lines 3, 8–9 in Algorithm 1) and 53% of the runtime in the backward substitution phase (lines 4–7 in Algorithm 1). This ratio is approximately in line with the flop distribution of these two phases. When RQIN computes 750 complex eigenvectors, the reduction phase constitutes 36% and the backward substitution phase 63% of the runtime. Thus, the majority of the time is spent on backward substitution.

The runtime decompositions of HSRQ3IN are shown in Figure 6. The runtime is split into the contribution of each task type to the total compute time for a sequential (left) and a parallel (right) run. Between 53% and 68% of the runtime is spent on Reduce tasks. The runtime difference between the real and the complex runs is due to the different amount of Givens rotations computed

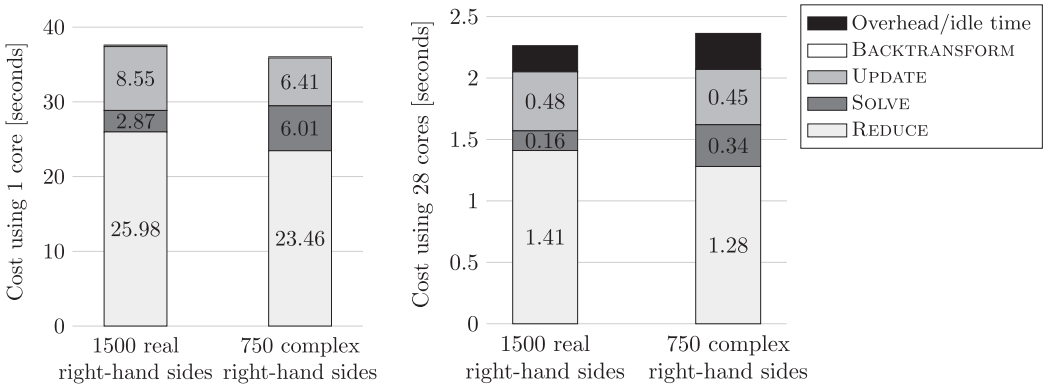

Fig. 6. Runtime decomposition of a sequential (left) and parallel (right) run on $\boldsymbol{H}_1$ for $n = 10000$.

during the run. Since a series of Givens rotation is computed per eigenvector, the complex run computes only half the number of Givens rotations, but requires mixed real-complex multiplications. The backward substitution phase contributes with Solve and Update tasks. The complex runs spend approximately double the time on Solve tasks than the real runs. This can be attributed to the complex-complex multiplications during the small backward substitutions. The runtime difference of Update tasks is due to the application of the Givens rotations. Analogously to the Reduce tasks, the complex runs apply only half the number of Givens compared to the real runs. The backtransform phase makes a negligible contribution to the total runtime. The runtime decompositions of RQIN and HSRQ3IN suggest that the reduction phase has become the new bottleneck of the revised RQ approach.

The parallel speedup ranges in 14–18 for all task types. Due to dynamic frequency scaling, the best possible parallel speedup on the test node is 18.4. Idle cores during the parallel runs contribute to the overhead/idle time when there are not enough tasks available for being scheduled.

Parallel scalability. The third experiment analyzes the strong scalability of HSRQ3IN. Strong scaling concerns the speedup for a fixed problem size subject to an increasing number of processing units. The used test system is $\boldsymbol{H}_1$ where $n \in \{10000, 40000\}$. The eigenvalue selection ratio is chosen as 5%, 15%, or 25%. Then, for example, the experiment with $n = 10000$ and 5% selected eigenvalues computes 500 columns (500 real eigenvectors or 250 complex eigenvectors). The eigenvalues are either all real or all complex.

Figure 7 extends the sequential experiment shown in Figure 5. It reports the runtime of a strong scaling experiment with up to 28 cores. The speedup over the sequential run is approximately by factor or 14. The best attainable speedup is 18.4. Figures 5 and 7 show that the new inverse iteration solver is clearly faster than the existing inverse iteration solvers and has reasonable parallel scalability.

Figure 8 displays the performance results with respect to the machine capabilities. It includes the multithreaded implementation of DGEMM as an upper bound. The plot assumes $3.5n^2$ flops per column, which is a lower bound of the true flop count. The real and the complex experiments attain a similar fraction of the theoretical peak performance and achieve a similar parallel speedup.

Overhead due to numerical scaling. The fourth experiment evaluates the cost of overflow avoidance. For this purpose, a run of DHSRQ3 on the bad system $\boldsymbol{H}_2$ and the good system $\boldsymbol{H}_3$ are compared. Recall that the bad system requires frequent numerical scaling, whereas the good system

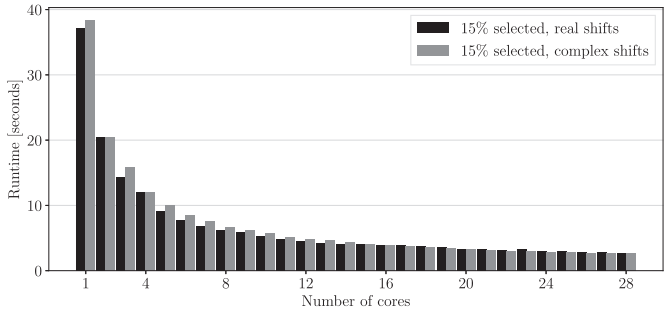

Fig. 7. Parallel runtimes on $\boldsymbol{H}_1$ using $n = 10000$.

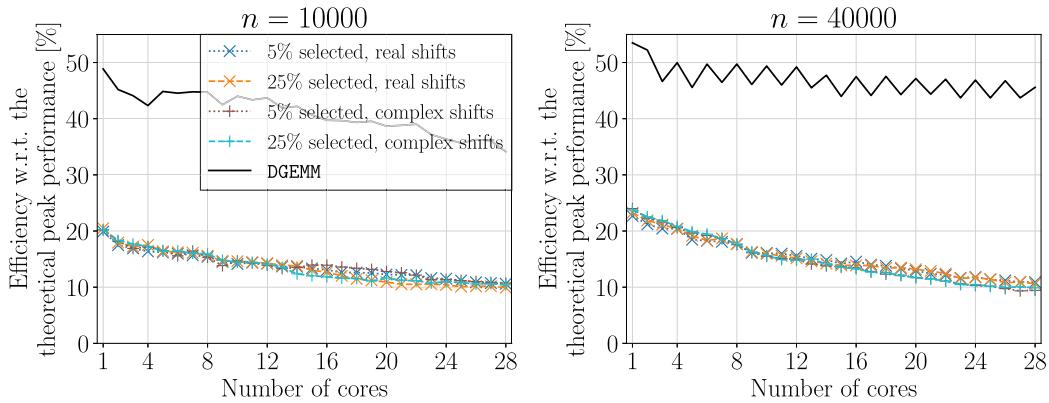

Fig. 8. Parallel scalability on $\boldsymbol{H}_1$ using $n = 10000$ (left column) and $n = 40000$ (right column). For reference, DGEMM (MKL) executes the same amount of flops as the 5% real shifts case and writes to an output matrix of size n-by-n.

never requires numerical scaling. The experiments solve shifted Hessenberg systems rather than an eigenvector problem because the triangular solve associated with $\boldsymbol{H}_2$ and $\boldsymbol{H}_3$ has been shown as the best and worst case, respectively [Kjelgaard Mikkelsen 2020]. As a consequence, the cost of overflow protection for inverse iteration is in between the two extremes. Note that both runs include the cost associated with the evaluation of overflow protection logic. The runtime difference between the two runs is solely the cost of numerical rescaling.

Figure 9 illustrates the measurements using $n = 10000$ and 1,500 real right-hand sides. The runs on the bad system are circa 6% slower than the ones on the good system. Numerical scaling only affects the backward substitution phase, i.e., Solve and Update tasks. The slowdown of these two task types is approximately 20% alike. The parallel speedup is comparable for both the good and the bad system. This suggests that robustness does not affect the parallel scalability.

6 CONCLUSION

This paper revises the RQ approach for solving shifted Hessenberg systems $(\boldsymbol{H} - \lambda_\ell \boldsymbol{I})\boldsymbol{x}_\ell = \alpha_\ell \boldsymbol{b}_\ell$ robustly for a large number of shifts. By rearranging the computation of the partial RQ factorization, matrix–matrix multiplications (level-3 BLAS) are introduced to the backward substitution phase. Since the solution of shifted Hessenberg systems is the most compute-intensive step in the computation of eigenvectors by inverse iteration, the revised RQ approach leads to a new inverse

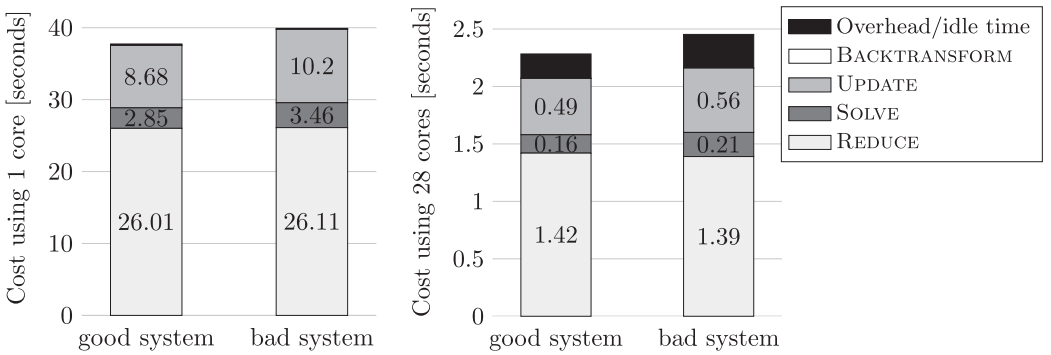

Fig. 9. Runtime decomposition of a sequential (left) and parallel (right) run computing 1500 real right-hand sides on $\boldsymbol{H}_2$ (bad system) and $\boldsymbol{H}_3$ (good system) for $n = 10000$.

iteration solver. The numerical experiments show that the new inverse iteration solver outperforms existing inverse iteration solvers.

ACKNOWLEDGMENTS

The author thanks Lars Karlsson for initiating and supporting this project. Furthermore, the author is grateful for the valuable comments by Martin Berggren, Lars Karlsson, Carl Christian Kjelgaard Mikkelsen and the anonymous reviewers.

REFERENCES

Edward Anderson. 1991. *Robust Triangular Solves for Use in Condition Estimation.* Technical Report. LAWN 36.

Edward Anderson, Zhaojun Bai, Christian Bischof, L. Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Alan McKenney. 1999. *LAPACK Users' Guide* (3rd ed.). SIAM.

Christopher Beattie, Zlatko Drmač, and Serkan Gugercin. 2012. A note on shifted Hessenberg systems and frequency response computation. *ACM Trans. Math. Softw.* 38, 2, Article 12 (Jan. 2012), 16 pages.

David Bindel, James Demmel, William Kahan, and Osni Marques. 2002. On computing Givens rotations reliably and efficiently. *ACM Trans. Math. Softw.* 28, 2 (June 2002), 206–238.

L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

Nela Bosner, Zvonimir Bujanović, and Zlatko Drmač. 2013. Efficient generalized Hessenberg form and applications. *ACM Transactions on Mathematical Software (TOMS)* 39, 3 (2013), 1–19.

Nela Bosner, Zvonimir Bujanović, and Zlatko Drmač. 2018. Parallel solver for shifted systems in a hybrid CPU–GPU framework. *SIAM Journal on Scientific Computing* 40, 4 (2018), C605–C633.

Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, Maksims Abalenkovs, Negin Bagherpour, et al. 2019. PLASMA: Parallel linear algebra software for multicore using OpenMP. *ACM Transactions on Mathematical Software (TOMS)* 45, 2 (2019), 1–35.

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software* 16, 1 (1990), 1–17.

Gene H. Golub and Charles F. Van Loan. 1996. *Matrix Computations* (3rd ed.). John Hopkins University Press.

Greg Henry. 1994. *The Shifted Hessenberg System Solve Computation.* Technical Report. Cornell University, NY, USA.

Greg Henry. 1995. A parallel unsymmetric inverse iteration solver. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*. 546–551.

Ilse C. F. Ipsen. 1997. Computing an eigenvector with inverse iteration. *SIAM Rev.* 39, 2 (1997), 254–291.

Carl Christian Kjelgaard Mikkelsen. 2020. Well-conditioned eigenvalue problems that overflow. (2020). arXiv:2005.05356

Carl Christian Kjelgaard Mikkelsen and Lars Karlsson. 2017a. Blocked algorithms for robust solution of triangular linear systems. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 68–78.

Carl Christian Kjelgaard Mikkelsen and Lars Karlsson. 2017b. Robust Solution of Triangular Linear Systems, NLAFET Working Note 9. (May 2017).

Carl Christian Kjelgaard Mikkelsen, Angelika B. Schwarz, and Lars Karlsson. 2019. Parallel robust solution of triangular linear systems. *Concurrency and Computation: Practice and Experience* 31, 19 (2019), e5064.

Beresford N. Parlett. 1998. *The Symmetric Eigenvalue Problem*. SIAM.

Gwendoline Peters and James H. Wilkinson. 1971. The calculation of specified eigenvectors by inverse iteration. In *Handbook for Automatic Computation*. Springer, 418–439.

Angelika Schwarz. 2020. *Comparing the Performance of Computing Eigenvectors from the Schur Matrix and by Inverse Iteration from the Hessenberg Matrix*. UMINF 21.08. https://webapps.cs.umu.se/uminf/reports/2021/008/part1.pdf.

Field Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. 2009. Introducing: The libflame library for dense matrix computations. *Computing in Science & Engineering* (2009).

J. M. Varah. 1968. The calculation of the eigenvectors of a general complex matrix by inverse iteration. *Math. Comp.* 22, 104 (1968), 785–791.

Received September 2020; revised December 2021; accepted May 2022