UMEÅ UNIVERSITY

Comparing the Performance of Computing Eigenvectors from the Schur Matrix and by Inverse Iteration from the Hessenberg Matrix

Angelika Schwarz

UMINF 21.08

Department of Computing Science

Comparing the Performance of Computing Eigenvectors from the Schur Matrix and by Inverse Iteration from the Hessenberg Matrix

Angelika Schwarz (angies@cs.umu.se)

Department of Computing Science, Umeå University, Sweden

Abstract Two standard approaches to the computation of eigenvectors corresponding to eigenvalues of a non-symmetric real matrix are (1) a backward substitution on the real Schur matrix followed by a matrix-matrix multiplication and (2) inverse iteration on the Hessenberg matrix. Both approaches have been realized as tiled, task-parallel solvers where a large proportion of the operations are matrix–matrix multiplications. The first contribution of this work addresses a memory bottleneck of the inverse iteration solver by improving the locality of memory accesses. The second contribution compares the two approaches quantitatively with numerical experiments.

1 Introduction

The computation of eigenvalues λ and corresponding eigenvectors $\boldsymbol{x} \neq \boldsymbol{0}$ of a matrix $\boldsymbol{A}$ such that $\boldsymbol{A}\boldsymbol{x} = \lambda\boldsymbol{x}$ is a classical problem in numerical linear algebra. This work focuses on the computation of eigenvectors for a dense and non-symmetric matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$. Two standard approaches are (a) the computation of eigenvectors from the Schur form [5, Sec. 7.6.4] and (b) inverse iteration from the Hessenberg matrix [5, Sec. 7.6.1].

The two approaches reduce $\boldsymbol{A}$ in a shared first step with an orthogonal matrix $\boldsymbol{Q}_0$ to an upper Hessenberg matrix $\boldsymbol{H} = \boldsymbol{Q}_0^T\boldsymbol{A}\boldsymbol{Q}_0$. The second step computes the real Schur form $\boldsymbol{S} = \boldsymbol{Q}_1^T\boldsymbol{H}\boldsymbol{Q}_1$, where $\boldsymbol{Q}_1$ is orthogonal and $\boldsymbol{S}$ is quasi-triangular, i.e., block triangular with 1-by-1 or 2-by-2 blocks on the diagonal. The two approaches differ with respect to the accumulation of the orthogonal transformations yielding $\boldsymbol{Q}_1$. If $\boldsymbol{Q}_1$ is available, an eigenvector can be computed from the Schur form by solving a shifted quasi-triangular system followed with a matrix–vector multiplication with $\boldsymbol{Q} = \boldsymbol{Q}_0\boldsymbol{Q}_1$. Inverse iteration, by contrast, only requires the eigenvalues revealed with $\boldsymbol{S}$ and obtains the eigenvectors by solving shifted Hessenberg systems. Hence, inverse iteration does not require the accumulation of $\boldsymbol{Q}_1$, implying a lower flop count.

If the Schur decomposition is computed with the QR algorithm, Golub et al. [5, p. 400] provide as rule-of thumb that inverse iteration is commonly used when at most 25% of the eigenvectors are sought. The author is not aware of a more recent rule-of-thumb. This work aims at reevaluating this rule-of-thumb

based on the reformulation of the two approaches as task-parallel tiled solvers [10, 11], which to the best of the author's knowledge are the fastest realizations of the underlying approaches on shared memory systems. Specifically, this work

- addresses a memory bottleneck of the inverse iteration algorithm presented in [10],
- adds loop blocking originally proposed by Henry [6] to the inverse iteration algorithm, lowering its flops count, and
- compares the two approaches experimentally, targeting the question when which approach is advantageous.

For this purpose, Section 2 summarizes the two approaches. Section 3 discusses improvements to the inverse iteration algorithm. Section 4 describes the experiments whose results are presented in Section 5.

2 Background

This section summarizes the algorithms for computing eigenvectors corresponding to selected eigenvalues from the Schur form and from the Hessenberg matrix. When many eigenvectors are desired, the eigenvalue problem can be written in matrix form as $\boldsymbol{AX} = \boldsymbol{X\Lambda}$, where $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues and $\boldsymbol{X}$ is an eigenvector matrix. Then the algorithms can benefit from level-3 BLAS.

2.1 Eigenvectors from the Schur Decomposition

Eigenvectors of a matrix $\boldsymbol{A}$ can be computed from its Schur decomposition in a two-step procedure. For clarity, the presentation uses complex arithmetic. Then the Schur decomposition is $\boldsymbol{S} = \boldsymbol{Q}^H\boldsymbol{AQ}$. The first step computes an eigenvector $\hat{\boldsymbol{y}}$ of $\boldsymbol{S}$ corresponding to λ. To guarantee an overflow-free representation, eigenvector solvers such as `dtrevc3` (LAPACK 3.10.0, see [3]) represent the eigenvector as a scaled vector $\hat{\boldsymbol{y}} = \gamma^{-1}\boldsymbol{y}$. The scaling factor $\gamma^{-1} \in (0, 1]$ allows scaling such that $\boldsymbol{y}$ is free from infinities (floating-point overflow). A scaled eigenvector $\gamma^{-1}\boldsymbol{y} = \left[\gamma^{-1}\boldsymbol{y}_1, 1, \boldsymbol{0}\right]^T$ satisfying $\boldsymbol{S}(\gamma^{-1}\boldsymbol{y}) = \lambda(\gamma^{-1}\boldsymbol{y})$ can be computed from

$$\begin{bmatrix} \boldsymbol{S}_{11} & \boldsymbol{s}_{12} & \boldsymbol{s}_{13} \\ \boldsymbol{0} & \lambda & \boldsymbol{s}_{12} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{S}_{33} \end{bmatrix} \begin{bmatrix} \alpha^{-1}\boldsymbol{y}_1 \\ 1 \\ \boldsymbol{0} \end{bmatrix} = \lambda \begin{bmatrix} \alpha^{-1}\boldsymbol{y}_1 \\ 1 \\ \boldsymbol{0} \end{bmatrix}$$

by a robust backward substitution of $(\boldsymbol{S}_{11} - \lambda\boldsymbol{I})\boldsymbol{y}_1 = -\gamma\boldsymbol{s}_{12}$. The term 'robust' refers to an overflow-free computation by virtue of the dynamic scaling. The computation of many eigenvectors yields $\boldsymbol{S}(\boldsymbol{Y\Gamma}^{-1}) = (\boldsymbol{Y\Gamma}^{-1})\boldsymbol{\Lambda}$. The diagonal matrix $\boldsymbol{\Gamma}^{-1}$ realizes columnwise scalings. A solution $\boldsymbol{Y\Gamma}^{-1}$ can be obtained

simultaneously by a robust backward substitution rich in matrix–matrix multiplications [11]. Eigenvectors $\boldsymbol{X}$ of $\boldsymbol{A}$ can be obtained by the backtransformation $\boldsymbol{X} \leftarrow \boldsymbol{Q}(\boldsymbol{Y}\boldsymbol{\Gamma}^{-1})$ realized as matrix–matrix multiplication [4], possibly preceded by further scaling to guarantee an overflow-free computation.

The robust backward substitution can be expressed as a tiled task-parallel algorithm that has the same task dependencies as standard tiled backward substitution. The backtransformation can be cast as a tiled matrix–matrix multiplication that exploits the generalized upper triangular shape of $(\boldsymbol{Y}\boldsymbol{\Gamma}^{-1})$. The implementation used in the experiments is EigVecs [11, Algorithm 9].

2.2 Eigenvectors from the Hessenberg Matrix

Eigenvectors of $\boldsymbol{A}$ can be computed from the Hessenberg decomposition $\boldsymbol{H} = \boldsymbol{Q}_0^T \boldsymbol{A} \boldsymbol{Q}_0$ by inverse iteration. Assuming that an eigenvalue λ is known (for example, from the Schur matrix), a corresponding eigenvector $\boldsymbol{y} \neq \boldsymbol{0}$ of $\boldsymbol{H}$ is approximated by a converging sequence

$$(\boldsymbol{H} - \lambda \boldsymbol{I})\boldsymbol{y}^{(k)} = \sigma^{(k)} \boldsymbol{y}^{(k-1)}, \quad k \geq 1. \tag{1}$$

The scalar $\sigma^{(k)}$ normalizes the iterate $\boldsymbol{y}^{(k)}$. An eigenvector of $\boldsymbol{A}$ is then given by $\boldsymbol{x} = \boldsymbol{Q}_0 \boldsymbol{y}$. The starting vector $\boldsymbol{y}^{(0)}$ can be chosen such that (1) reaches convergence in a single iteration [12]. Then the computation of an eigenvector requires the solution of

$$(\boldsymbol{H} - \lambda \boldsymbol{I})\boldsymbol{y}^{(1)} = \sigma^{(1)} \boldsymbol{y}^{(0)}. \tag{2}$$

A solution strategy introduced by Henry [6, 7] is the RQ approach. The RQ approach constructs suited Givens rotations $\boldsymbol{G}_n \dots \boldsymbol{G}_2$ such that

$$\underbrace{(\boldsymbol{H} - \lambda \boldsymbol{I})\boldsymbol{G}_n^T \dots \boldsymbol{G}_2^T}_{\boldsymbol{R}} \underbrace{\boldsymbol{G}_2 \dots \boldsymbol{G}_n \boldsymbol{y}^{(1)}}_{\gamma^{-1}\boldsymbol{z}} = \boldsymbol{y}^{(0)}, \tag{3}$$

where $\boldsymbol{R}$ is triangular and $\boldsymbol{Q} = \boldsymbol{G}_2 \dots \boldsymbol{G}_n$ is orthogonal. The triangular system $\boldsymbol{R}\boldsymbol{z} = \gamma \boldsymbol{y}^{(0)}$ is solved for $\gamma^{-1} \in (0, 1]$ and $\boldsymbol{z}$ representing $\gamma^{-1}\boldsymbol{z}$ with robust backward substitution. Assuming that $\boldsymbol{y}$ is scaled such that the application of the Givens rotations does not overflow, the solution is recovered by the backtransformation $\boldsymbol{y}^{(1)} \leftarrow \boldsymbol{G}_n^T \left(\dots \left(\boldsymbol{G}_2^T \boldsymbol{y}\right)\right) \boldsymbol{z}$, and normalized $\boldsymbol{y} \leftarrow \sigma^{(1)}(\gamma^{-1}\boldsymbol{y}^{(1)})$.

If many systems (2) are solved simultaneously, the RQ approach can be expressed such that the robust backward substitution is rich in matrix–matrix multiplications. The resulting solver `HSRQ3IN` [10] splits the computation into a preparatory reduction phase and a backward substitution phase. The next section details how the RQ approach can be expressed to simultaneously solve shifted Hessenberg systems and presents improvements to the reduction phase.

3 Improvements to the Inverse Iteration Solver

This section summarizes how the RQ-based inverse iteration solver `HSRQ3IN` can benefit from level-3 BLAS operations. Then we improve the reduction phase, the first computational step of `HSRQ3IN`.

3.1 Solving Shifted Hessenberg Systems with mostly Level-3 BLAS Operations

The solver `HSRQ3IN` [10] targets the simultaneous solution of several shifted Hessenberg systems (2), where $\boldsymbol{H} \in \mathbb{R}^{n \times n}$ is shared across shifts, but λ and the right-hand sides $\boldsymbol{y}^{(0)}$ can be distinct. We focus on the case when the shifts are real and robustness is not needed.

The solver `HSRQ3IN` approaches the solution by computing a partial RQ factorization. It constructs Givens rotations whose application reduces $\boldsymbol{H} - \lambda \boldsymbol{I}$ column-by-column into a shift-dependent triangular matrix $\boldsymbol{R}$, starting from the right. Specifically, a Givens rotation $\boldsymbol{G}_j^T$ annihilates the subdiagonal entry $h_{j,j-1}$ and transforms the columns j and $j-1$. We look at the partial RQ factorization after the application of the Givens rotations $\boldsymbol{G}_n^T \dots \boldsymbol{G}_j^T$

$$\left((\boldsymbol{H} - \lambda \boldsymbol{I}) \boldsymbol{G}_n^T \dots \boldsymbol{G}_j^T \right) \left(\boldsymbol{G}_j \dots \boldsymbol{G}_n \boldsymbol{y}^{(1)} \right) = \boldsymbol{y}^{(0)}. \tag{4}$$

For this, we partition the Hessenberg matrix

$$\boldsymbol{H} = \begin{array}{c} \begin{matrix} j-2 & 1 & n-j+1 \end{matrix} \\ \left[\begin{matrix} \boldsymbol{H}_{11} & \boldsymbol{h}_{12} & \boldsymbol{H}_{13} \\ \boldsymbol{0} & \boldsymbol{h}_{22} & \boldsymbol{H}_{23} \end{matrix} \right] \end{array} \begin{matrix} j-1 \\ n-j+1 \end{matrix}$$

such that $\begin{bmatrix} \boldsymbol{H}_{11} & \boldsymbol{h}_{12} \end{bmatrix}$ and $\boldsymbol{H}_{23}$ are square and the blocks have the indicated dimensions. Note that the top entry of $\boldsymbol{h}_{22}$ is the only non-zero entry of this vector segment and will be annihilated with the Givens rotation $\boldsymbol{G}_j^T$. The matrix $\boldsymbol{R}$ is partitioned alike. Further, we partition the solution $\boldsymbol{y}^{(1)}$, the intermediate solution $\boldsymbol{z}$ (see (3)) and the right-hand side $\boldsymbol{y}^{(0)}$ conformally

$$\boldsymbol{y}^{(1)} = \begin{bmatrix} \boldsymbol{y}_1^{(1)} \\ \boldsymbol{y}_2^{(1)} \end{bmatrix}, \qquad \boldsymbol{z} = \begin{bmatrix} \boldsymbol{z}_1 \\ \boldsymbol{z}_2 \end{bmatrix}, \qquad \boldsymbol{y}^{(0)} = \begin{bmatrix} \boldsymbol{y}_1^{(0)} \\ \boldsymbol{y}_2^{(0)} \end{bmatrix}.$$

The application of the Givens rotations to the left term of (4) transforms

$$\begin{aligned} &\left(\begin{bmatrix} \boldsymbol{H}_{11} & \boldsymbol{h}_{12} & \boldsymbol{H}_{13} \\ \boldsymbol{0} & \boldsymbol{h}_{22} & \boldsymbol{H}_{23} \end{bmatrix} - \lambda \boldsymbol{I} \right) \boldsymbol{G}_n^T \dots \boldsymbol{G}_j^T = \\ &\qquad \begin{bmatrix} \boldsymbol{H}_{11} & \tilde{\boldsymbol{r}} & \boldsymbol{R}_{13} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{R}_{23} \end{bmatrix} - \lambda \operatorname{diag}(\underbrace{1, \dots, 1}_{j-2}, \underbrace{0, \dots, 0}_{n-j+2}). \end{aligned} \tag{5}$$

The right block column is reduced to columns of $\boldsymbol{R}$; the column $\begin{bmatrix} \boldsymbol{h}_{12} & \boldsymbol{h}_{22} \end{bmatrix}^T$ is incompletely reduced to $\tilde{\boldsymbol{r}}$. We refer to $\tilde{\boldsymbol{r}}$ as *cross-over column*.

With $\boldsymbol{G}_2 \dots \boldsymbol{G}_n \boldsymbol{y}^{(1)} = \boldsymbol{z}$, the middle term of (4) transforms

$$\boldsymbol{G}_j \dots \boldsymbol{G}_n \boldsymbol{y}^{(1)} = \begin{bmatrix} \star \\ \boldsymbol{z}_2 \end{bmatrix}.$$

The problem size can be shrunk analogously to standard tiled backward substitution. A small backward substitution $\boldsymbol{R}_{23}\boldsymbol{z}_2 = \boldsymbol{y}_2^{(0)}$ is solved. The computed solution $\boldsymbol{z}_2$ is used in a linear update, which in factored form reads

$$\boldsymbol{y}_1^{(0)} \leftarrow \boldsymbol{y}_1^{(0)} - \begin{bmatrix} \boldsymbol{h}_{12} - \lambda \boldsymbol{e}_{j-1} & \boldsymbol{H}_{13} \end{bmatrix} \boldsymbol{F}\boldsymbol{G}_n^T \dots \boldsymbol{G}_j^T \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{z}_2 \end{bmatrix}. \tag{6}$$

Here, $\boldsymbol{F} = \begin{bmatrix} \boldsymbol{0}_{n-j+2 \times j-2} & \boldsymbol{I}_{n-j+2} \end{bmatrix}$ ensures matching matrix dimensions by reducing the n-by-n Givens matrices to the submatrix relevant to the update. The factored update introduces options to add parentheses. If the Givens rotations are applied to the left, (6) yields the familiar update

$$\boldsymbol{y}_1^{(0)} \leftarrow \boldsymbol{y}_1^{(0)} - \begin{bmatrix} \boldsymbol{0} & \tilde{\boldsymbol{r}} & \boldsymbol{R}_{13} \end{bmatrix} \begin{bmatrix} \boldsymbol{0} \\ 0 \\ \boldsymbol{z}_2 \end{bmatrix},$$

i.e., $\boldsymbol{y}_1^{(0)} \leftarrow \boldsymbol{y}_1^{(0)} - \boldsymbol{R}_{13}\boldsymbol{z}_2$. If the Givens rotations are applied to the right

$$\boldsymbol{F}\boldsymbol{G}_n^T \dots \boldsymbol{G}_j^T \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{z}_2 \end{bmatrix} = \begin{bmatrix} \rho \\ \boldsymbol{u} \end{bmatrix},$$

(6) can be realized with a block matrix–vector multiplication

$$\boldsymbol{y}_1^{(0)} \leftarrow \boldsymbol{y}_1^{(0)} - \begin{bmatrix} \boldsymbol{h}_{12} - \lambda \boldsymbol{e}_{j-1} & \boldsymbol{H}_{13} \end{bmatrix} \begin{bmatrix} \rho \\ \boldsymbol{u} \end{bmatrix}. \tag{7}$$

If many right-hand sides are computed simultaneously, the block multiplication with $\boldsymbol{H}_{13}$ corresponds to a wide matrix–matrix multiplication. Then the majority of the flops of the backward substitution corresponds to level-3 BLAS operations. The remaining matrix in (5), $\begin{bmatrix} \boldsymbol{H}_{11} & \tilde{\boldsymbol{r}} \end{bmatrix} - \lambda \operatorname{diag}(1, ..., 1, 0)$, is again upper Hessenberg. Provided that the cross-over column $\tilde{\boldsymbol{r}}$ is available, the same technique can be applied repeatedly to solve the remaining system.

The dependence on the availability of the cross-over columns leads to a computation in three phases, the reduction phase, the backward substitution phase and the backtransform phase. It is the purpose of the reduction phase to compute and record the necessary cross-over columns so that the backward substitution phase can benefit from level-3 BLAS. A Matlab prototype demonstrating the procedure can be found at `https://github.com/angsch/block-algorithms/blob/master/blockShiftedHessenbergSolve.m`.

In the following, we will focus on improvements to the reduction phase.

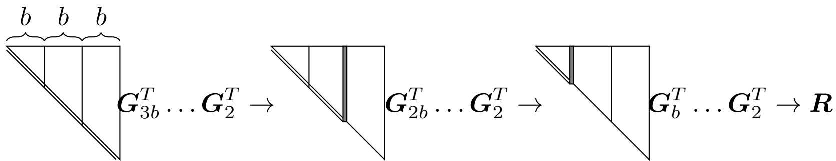


Figure 1: Cross-over columns (gray) surface when the Givens rotations are applied in batches. For 3 tile columns 2 cross-over columns have to be recorded. Columns to the right of the cross-over column correspond to columns of $\boldsymbol{R}$.

3.2 Improvements to the Reduction Phase

The reduction phase computes the Givens rotations and the cross-over columns required by the backward substitution phase. It is currently a bottleneck of `HSRQ3IN`. Numerical experiments show that the reduction phase comprises 42% of the flops, but frequently constitutes more than 60% of the total runtime, see [10, Fig. 6] or the left column in Figure 3. This section summarizes the reduction phase and presents the improvements that are realized in `HSRQ3INv2`, the improved version of `HSRQ3IN`.

The computation of cross-over columns The application of Givens rotations in batches of a fixed size b reduces the shifted Hessenberg system successively to triangular form

$$(\boldsymbol{H} - \lambda \boldsymbol{I}) \underbrace{\boldsymbol{G}_n^T \dots \boldsymbol{G}_{n-b+1}^T}_{b \text{ rotations}} \underbrace{\boldsymbol{G}_{n-b}^T \dots \boldsymbol{G}_{n-2b+1}^T}_{b \text{ rotations}} \dots \boldsymbol{G}_2^T = \boldsymbol{R}.$$

The batch size b induces a partitioning into tile columns of size b. To simplify the presentation of the algorithms, we assume that the matrix size n is an integer multiple of the batch size $n = Nb$. The application of the Givens rotations in batches yields $N-1$ cross-over columns. An algorithm for computing the cross-over columns in listed in Algorithm 1. Line 6 of Algorithm 1 yields $\boldsymbol{G}_j^T = \boldsymbol{I}_{j-2} \oplus \boldsymbol{\Phi}_j^T \oplus \boldsymbol{I}_{n-j}$, where $\oplus$ is the direct sum. Algorithm 1 is essentially Henry's algorithm [6, Algorithm 2] reduced to the computation of the triangular R factor. An example for the batched reduction for the case $N = 3$ is illustrated in Figure 1. The flop count of Algorithm 1 is $1.5n^2 + \mathcal{O}(n)$.

Algorithm 1 and, thus, the kernels of a tiled reduction algorithm can trivially be extended to multiple shifts by adding a loop over the shifts. The Givens rotations and the cross-over columns are distinct for distinct shifts. Shifts can be processed in batches in parallel without any dependences. Further parallelism can be introduced by partitioning the tile columns further into tile rows. This corresponds to splitting line 7 of Algorithm 1 into smaller updates, see [10, Algorithm 4].

Algorithm 1: $\tilde{\boldsymbol{R}} \leftarrow$REDUCE($\boldsymbol{H}$, λ)

Data: Upper Hessenberg matrix $\boldsymbol{H} \in \mathbb{R}^{n\times n}$, shift $\lambda \in \mathbb{R}$, tile size b with $n = Nb$

Result: Cross-over columns $\tilde{\boldsymbol{R}} \in \mathbb{R}^{n\times N}$

1 $\boldsymbol{v} \leftarrow \boldsymbol{h}(1:n, n)$; $v(n) \leftarrow v(n) - \lambda$;

2 $\tilde{\boldsymbol{R}} \leftarrow$ ZEROS(n, N);

3 $t \leftarrow N$; // Tile column index

4 $\tilde{\boldsymbol{r}}(:, t) \leftarrow \boldsymbol{v}$;

5 **for** $j \leftarrow n : -1 : 2$ **do**

6 Find a rotation $\boldsymbol{\Phi}_j^T = \begin{bmatrix} c_j & -s_j \\ s_j & c_j \end{bmatrix}$ s. t. $\begin{bmatrix} h(j, j-1) & v(j) \end{bmatrix} \boldsymbol{\Phi}_j^T = \begin{bmatrix} 0 & \star \end{bmatrix}$;

7 $\boldsymbol{v}(1:j-2) \leftarrow c_j \boldsymbol{h}(1:j-2, j-1) - s_j \boldsymbol{v}(1:j-2)$;

8 $v(j-1) \leftarrow c_j(h(j-1, j-1) - \lambda) + s_j v(j-1)$;

9 **if** $j \mod b = 0$ **then**

10 $\tilde{\boldsymbol{r}}(1:j, t) \leftarrow \boldsymbol{v}(1:j)$;

11 $t \leftarrow t - 1$;

12 **return** $\tilde{\boldsymbol{R}}$;

Loop blocking Henry [6, p. 8] notes for the original RQ approach that a small number of Givens rotations can be aggregated and applied jointly. Beattie et al. [2, Sec. 5.2.1] give the full code for a blocking factor of 2. Applied to Algorithm 1, this corresponds to loop blocking of line 5. Products of the components of the Givens rotations c_k, s_k can be aggregated before transforming the column vectors $\boldsymbol{h}$ and $\boldsymbol{v}$. Hence, loop blocking lowers the flop count of the reduction. The theoretically best though unattainable flop reduction is by $1/2n^2 + \mathcal{O}(n)$ flops. What is not written in the above references, is a practical choice of the blocking factor. Our implementation chooses a blocking factor of 4, which corresponds to a flop reduction by $3/8n^2 + \mathcal{O}(n)$. This choice is motivated by the number of available registers to store the products. On our test machine, higher blocking factors slow down the computation despite (slightly) lower flop counts due to register spilling. The added delay by storing and loading of quantities that are not stored in registers across loop iterations is higher than the gain from flop savings.

Data locality To improve the spatial locality, a mix between column major and tile layout is introduced. The Hessenberg matrix and the eigenvector matrix are stored in column major layout for compatibility with libraries such as LAPACK and StarNEig [8]. The storage layout of the internal workspaces for the Givens rotations and the cross-over columns is changed into tile layout.

To improve the temporal locality, the storage order of the cross-over columns is revised. Three indices identify a cross-over column, a tile row index, a tile column index and a shift index. The cross-over columns are stored in contiguous

segments for batches of shifts. In other words, the tile column index is the outermost index. Hence, a cache-friendly processing order is to reduce a tile column first before other tile columns to the left are reduced. This order promises improved temporal locality for accesses to the Givens rotations.

Next we address domain locality. It is well understood that remote accesses incur a lower bandwidth and a higher latency in shared memory NUMA systems. Most operating systems support a first touch policy. When memory is allocated, the mapping onto a physical memory address is deferred. Only when touched for the first time, the mapping is established. The physical page maps into the locality domain of the core executing the read/write instruction. The first touch policy and dynamic scheduling of OpenMP tasks can lead to non-local memory accesses. The impact of the first touch policy on scheduling in locality domains has been evaluated in [9]. We aim at computing all tasks of a batch of shifts in the same locality domain. Then, ideally the only non-local accesses are accesses to shared data such as $\boldsymbol{H}$.

Support for domain locality has been introduced with the `affinity` clause in OpenMP 5.0. So far compilers have only limited support of the OpenMP `affinity` clause[1]. To evaluate the *potential* of the `affinity` clause for the inverse iteration solver, we mimic the effect of task affinity through the MPI shared memory support introduced with MPI 3 and guarantee that all tasks to be scheduled in the local locality domain.

We assign one MPI rank per locality domain (`I_MPI_DOMAIN=socket`). One rank allocates the Hessenberg matrix, the shifts and the eigenvector matrix as shared memory regions (`MPI_Win_allocate_shared`). The other ranks acquire access to it (`MPI_Win_shared_query`). That way the allocated memory is contiguous and identical to an allocation in shared memory. All threads spawned within an MPI process can access the shared memory region through RMA operations. Our locality domain-aware work distribution partitions the eigenvector matrix into tile columns. All tasks created for the solution of one tile column spawn in the same locality domain. As a result, every locality domain has its own task graph handled by the OpenMP runtime. Then the first touch policy allocates the Givens rotations and the cross-over columns in the locality domain the computation is executed in. All shared matrices are initialized such that the physical placement is distributed across locality domains.

4 Execution Environment

This section presents the test environment, the test problems and summarizes the routines used in the numerical experiments.

[1] The affinity clause is not supported by the Intel compiler version used in the experiments and not fully implemented in clang 13, see `https://clang.llvm.org/docs/OpenMPSupport.html` [July 2021]

Name	Purpose	Flops
`dhseqr` (MKL)	compute eigenvalues (no accumulation of orthogonal transformations)	$10n^3 + \mathcal{O}(n^2)$ [5, p. 391]
`HSRQ3IN` ([10])	eigenvectors by inverse iteration	$3.5n^2 + \mathcal{O}(n)$ per eigenvector column
`HSRQ3INv2` ([10], this work)	eigenvectors by inverse iteration	$3.125n^2 + \mathcal{O}(n)$ per eigenvector column

Table 1: Overview of the routines used for the computation of eigenvectors from the Hessenberg matrix.

Hardware The experiments are executed on an Intel Xeon Gold 6132 (Skylake) node. A node has two NUMA islands with 14 cores each. Its theoretical peak performance is 83.2 Gflops/s per core and 2329.6 Gflops/s per node. The memory bandwidth for domain-local memory accesses was measured using the STREAM triad benchmark at 162 GB/s for a full node. The latency matrix of memory accesses was measured with the Intel Memory Latency Checker version 3.9 at 139 ns between NUMA domains and 87 ns in the same NUMA domain. The node has dynamic frequency scaling enabled.

Software The software is built with the Intel compiler 19.0.1.144 at optimization level `-O2` using AVX-512 instructions. It is linked against MKL 2019.1.144. The domain locality through MPI was realized by linking against Intel MPI. Threads are pinned by setting `KMP_AFFINITY` to `compact`.

The numerical experiments use two library implementations of the QR algorithm. The first library implementation is the LAPACK routine `dhseqr` available in MKL 2019.1.144. The routine can be configured to either compute the Schur decomposition accumulating the orthogonal transformations or to solely compute the eigenvalues without accumulating the orthogonal transformations. The parallel experiments use the multi-threaded version of MKL.

The second library implementation is the shared memory implementation `starneig_SEP_SM_Schur` available in StarNEig v0.1.7 [8] built upon StarPU 1.3.7 [1]. The routine `starneig_SEP_SM_Schur` computes the Schur decomposition in a tiled, task-parallel fashion. The numerical experiments report the best runtime using at most the core count specified in the experiments. All runs use the default tile size. StarNEig v0.1.7 does not support the option to solely compute the eigenvalues without accumulating the orthogonal transformations.

Further, two routines for the computation of eigenvectors are employed. First, `EigVecs` [11, Algorithm 9] computes the eigenvectors from the Schur form by a robust backward substitution and a backtransformation. The flop count depends on the position of the associated 1-by-1 or 2-by-2 block on the diagonal of the Schur matrix. An upper bound is $4n^2 + \mathcal{O}(n)$ per eigenvector

Name	Purpose	Flops
`dhseqr` (MKL)	compute Schur decomposition	$25n^3 + \mathcal{O}(n^2)$ [5, p. 391]
`starneig_SEP_SM_Schur` ([8])	compute Schur decomposition	$25n^3 + \mathcal{O}(n^2)$ [5, p. 391]
`EigVecs` ([11])	eigenvectors by backsolve and backtransform	$4n^2 + \mathcal{O}(n)$ per eigenvector column

Table 2: Overview of the routines used for the computation of eigenvectors from the Schur decomposition.

Name	n	Nonzeros		Origin of problem
sherman3	5005	20033	0.08%	computational fluid dynamics
rajat03	7602	32653	0.06%	circuit simulation matrix
graham1	9035	335472	0.41%	computational fluid dynamics
sinc15	11532	551184	0.41%	materials problem
ex11	16614	1096948	0.40%	computational fluid dynamics

Table 3: Non-symmetric sparse test matrices from the SuiteSparse Matrix Collection used in the numerical experiments.

column. Second, the eigenvectors are computed from the Hessenberg matrix. This work added improvements to the inverse iteration routine `HSRQ3IN` [10], yielding `HSRQ3INv2`. `HSRQ3INv2` uses loop blocking, stores workspaces in tile layout and realizes domain locality. A summary of the two routes towards obtaining the eigenvectors is given in Table 1 and Table 2.

Test Problems The first test set aims at quantifying the improvements to the reduction phase of `HSRQ3IN`. It constructs an upper triangular $\boldsymbol{T} \in \mathbb{R}^{n \times n}$, where $t_{jj} = j$ and the superdiagonal entries are random in $(0, 1]$. The eigenvalues t_{jj} are all real and well-separated. The matrix $\boldsymbol{T}$ is transformed with a random Householder matrix $\boldsymbol{Q}_0 = \boldsymbol{I} - 2/(\boldsymbol{v}^T\boldsymbol{v})\boldsymbol{v}\boldsymbol{v}^T$, where $\boldsymbol{v}$ is a random vector. The resulting matrix $\boldsymbol{A} \leftarrow \boldsymbol{Q}_0\boldsymbol{T}\boldsymbol{Q}_0$ is reduced to Hessenberg form with the routine `starneig_SEP_SM_Hessenberg` available in StarNEig v0.1.7 $\boldsymbol{H}_2 \leftarrow \boldsymbol{Q}_1^T\boldsymbol{A}\boldsymbol{Q}_1$. The inverse iteration solvers receive the exact eigenvalues as input. Convergence is reached in a single iteration.

The second test set targets the question which approach to computing eigenvectors is fastest. Since both approaches share the Hessenberg reduction, the experiments start from Hessenberg matrices. The computed eigenvectors are eigenvectors of these Hessenberg matrices. Both synthetic and real-world matrices are used. A synthetic test matrix is created from a dense, random ma-

trix $\boldsymbol{A}$ with entries in $(0,1]$. The matrix $\boldsymbol{A}$ is reduced to Hessenberg form $\boldsymbol{H}_1 \leftarrow \boldsymbol{Q}^T\boldsymbol{A}\boldsymbol{Q}$ with `starneig_SEP_SM_Hessenberg`. The selected real-world matrices are listed in Table 3 and aim at representing the behavior of real-world problems. It is known that the convergence speed of the QR algorithm depends on the properties of the matrix. The experiments use sparse matrices because there is no dense counterpart to the SuiteSparse Matrix Collection.

Reproducability The experiments evaluate the wall time when a fraction $p \in (0,1]$ of the eigenvectors are sought. Since the eigenvalues can appear in any order in the Schur matrix, the experiments randomly select diagonal blocks until the desired eigenvector count is reached. We count a 2-by-2 block representing a complex conjugate pair of eigenvalues as two eigenvectors although only one is computed. That way, the computational cost per column is comparable between real and complex eigenvalues.

Since the position of the selected blocks influences the flop count and thereby also the runtime, the tests with synthetic matrices report the average of 5 samples. Each sample has a different input matrix $\boldsymbol{H}_1$. To minimize outliers in the measurements, each each sample is executed 3 times and the median runtime is used in the calculation of the average runtime.

Source code The shared memory implementation including loop blocking and usage of tile layout is available at `https://github.com/angsch/hsrq3in`.

5 Results of Numerical Experiments

This section presents the results of three sets of experiments. The first set concerns the inverse iteration routine `HSRQ3IN` and demonstrates the improvements added in this work. The second and third set compare the two approaches to computing eigenvectors.

5.1 Improvements to the Reduction Phase

The first experiment aims at quantifying the effectiveness of loop blocking, storing workspaces in tile layout and domain locality across NUMA domains. Figure 2 (left) shows the runtime relative to `HSRQ3IN`; Figure 2 (right) displays the performance relative to the machine capabilities. Loop blocking lowers the runtime approximately proportional to the fraction of flops saved, see Table 1 for the flop approximations. Since the runtime decreases in accordance to the flop savings, `HSRQ3IN` and `HSRQ3INv2` attain approximately the same fraction of the peak performance.

The usage of tile layout improves the runtime slightly. A computation with locality domains yields a significant improvement. Results for 2,..., 27 cores are omitted to avoid a biased comparison. Non-domain-local and domain-local

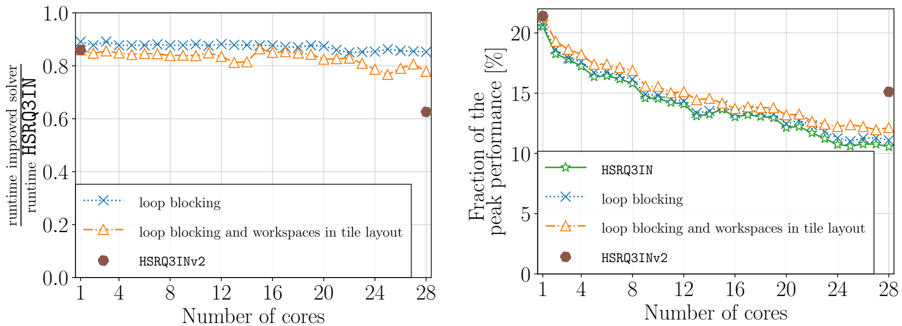


Figure 2: Relative improvements over HSRQ3IN (left) and attained fraction of the theoretical peak performance (right) on $\boldsymbol{H}_2$ where $n = 10000$ and 2500 real eigenvectors are computed.

Inverse iteration routine	1 core	28 cores
DHSEIN (LAPACK/oneMKL)	1470	n/a
RQ approach ([6, Algorithm 2])	169.4	n/a
HSRQ3IN	62.3	3.73
HSRQ3INv2	45.3	2.29

Table 4: Computational cost in seconds for computing 2500 real eigenvectors from $\boldsymbol{H}_2$ with size $n = 10000$.

runs favor distinct thread placements. Table 4 relates the improvements to existing inverse iteration solvers.

The next experiment quantifies how the computational phases contribute to the total runtime, see Figure 3. The reduction phase dominates the runtime of HSRQ3IN. Loop blocking only affects the reduction phase and reduces the runtime approximately by the amount of flops saved. The storage of the internal workspaces in tile layout and the execution of all tasks of a tile column within a locality domain improves all computational phases alike.

5.2 Race Between Eigenvectors Routines

The next experiments evaluate which approach to obtaining eigenvectors has the lowest computational cost.

Synthetic matrices Figure 4 compares the routes to eigenvectors on the Hessenberg matrix $\boldsymbol{H}_1$ with dimension 10000. The computation of eigenvectors by inverse iteration is advantageous for sequential runs when less than 50% of the eigenvectors are sought. In a parallel run, the fasted approach depends

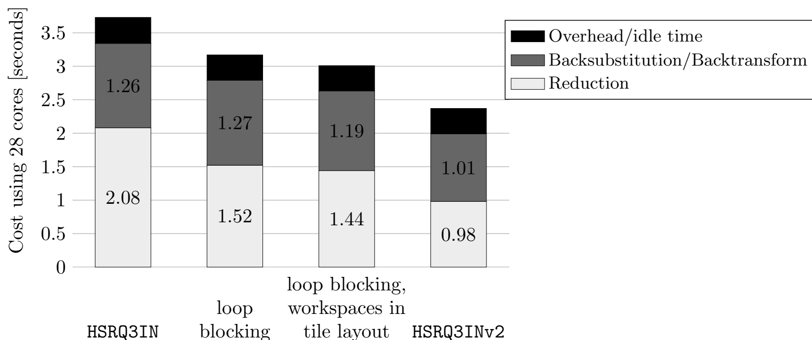


Figure 3: Decomposition of the computational cost into the computational phases on $\boldsymbol{H}_2$, $n = 10000$ and 2500 real eigenvectors.

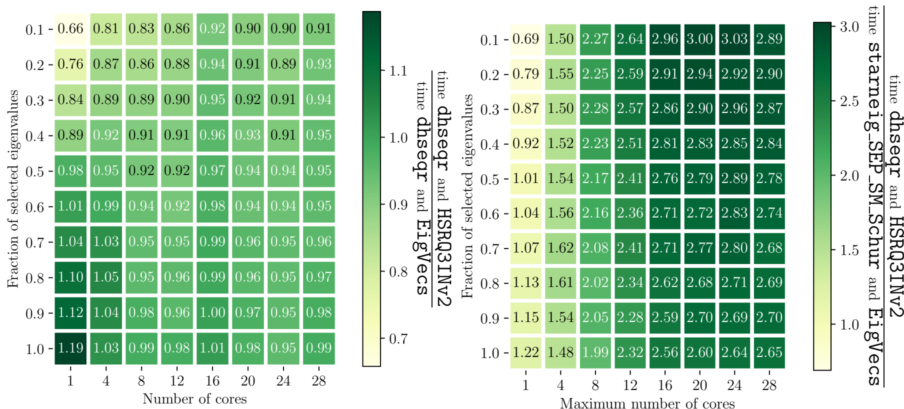
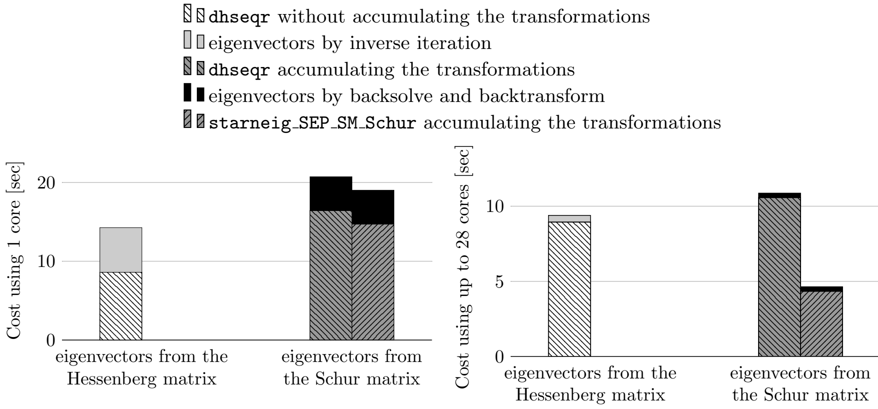


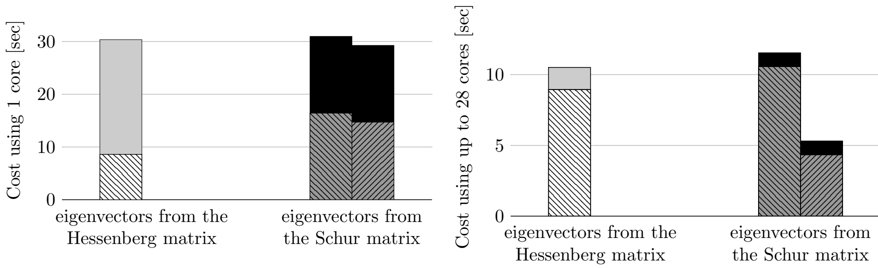
Figure 4: Relative wall times on $\boldsymbol{H}_1$ of size 10000.

on which implementation of the QR algorithm is used. If `dhseqr` is used, inverse iteration is the method of choice when up to 50% of the eigenvectors are sought. For larger fractions, the two approaches have a comparable computational cost. If `starneig_SEP_SM_Schur` is used, parallel computations favor the computation of eigenvectors from the Schur decomposition.

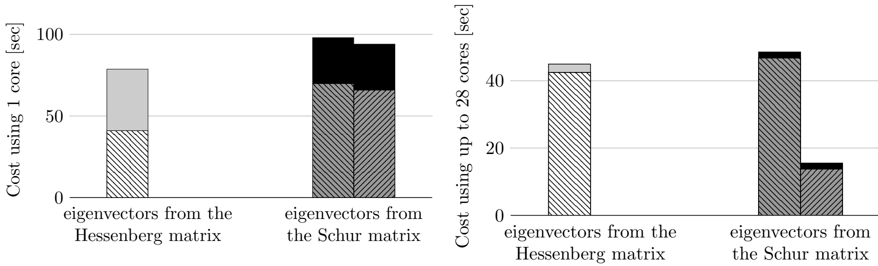
The next experiment aims at quantifying the time spent in each phase. For this purpose, Figure 5 decomposes the runtime into the fraction spent on the QR algorithm and the fraction spent on the eigenvector computation. The experiments (a) and (c) compute 25% of the eigenvectors, which corresponds to the cutoff value suggested by Golub et al. [5, p. 400]. The experiments (b) and (d) compute one eigenvector for each block on the diagonal of the Schur decomposition and serve as an upper bound for the fraction of the wall time that is spent in an eigenvector routine.



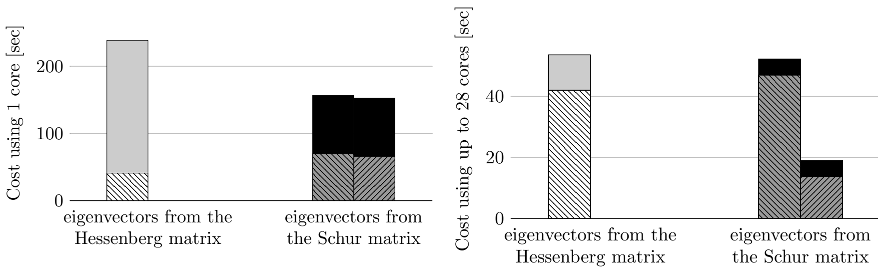
(a) Computation of 1250 eigenvector columns for $\boldsymbol{H}_1$ with size 5000.



(b) Computation of 5000 eigenvector columns for $\boldsymbol{H}_1$ with size 5000.



(c) Computation of 2500 eigenvector columns for $\boldsymbol{H}_1$ with size 10000.



(d) Computation of 10000 eigenvector columns for $\boldsymbol{H}_1$ with size 10000.

Figure 5: Cost comparison of the two approaches to computing eigenvectors.

	sherman3	rajat03	graham1	sinc15	ex11
`dhseqr` accumulating the transformations	9.42 5.56	20.43 10.73	57.63 32.57	33.67 19.48	279.51 111.15
`dhseqr` without accumulating the transformations	3.91 4.16	12.22 10.06	31.11 29.01	19.44 17.81	112.52 90.13
`starneig_SEP_SM_Schur` accumulating the transformations	11.31 8.83	9.16 7.30	70.31 34.74	22.28 45.83	330.09 77.21

Table 5: Runtimes in seconds (top: sequential, bottom: using up to 28 cores) of computing the Schur form from the Hessenberg form.

In the serial experiments, the flop savings during the QR algorithm pay off if a small number of eigenvectors is sought. For larger computations, the eigenvector routines contribute significantly to the runtime. Since `EigVecs` runs at a higher fraction of the peak performance than `HSRQ3INv2`, the computation is faster in spite of executing more flops.

In the parallel experiments. both eigenvector routines attain a comparable speedup. The runtime is dominated by the QR algorithm. A runtime comparison between the two modes of operation of `dhseqr` suggests that the additional flops for accumulating the orthogonal transformations have a minor impact on the wall time. This explains that the two approaches to computing eigenvectors yield similar runtimes when built upon `dhseqr`, see Figure 4 (left). Since `starneig_SEP_SM_Schur` outperforms `dhseqr` without accumulating the transformations, the computation of eigenvector from the Schur form is often significantly faster, see Figure 4 (right). The experiments do not couple inverse iteration with `starneig_SEP_SM_Schur` (StarNEig v.0.1.7) because the latter only support a mode where the orthogonal transformations are accumulated.

Real-world matrices It is well known that the convergence speed of the QR algorithm depends on properties of the matrix. This motivates the usage of real-world matrices as test cases. The next experiment uses the matrices listed in Table 3 from the SuiteSparse Matrix Collection. Table 5 reports the results of the QR algorithm. The computational cost tends to be lower than for synthetic matrices of similar size.

Table 6 compares the eigenvector routines for 25% and 50% sought eigenvectors. For matrices where the QR algorithm converges quickly, inverse iteration can be fastest way to eigenvectors for both sequential and parallel computations. This is, for example, the case for ex11 (serial) or sherman3 (parallel) when 25% of the eigenvectors are sought. The computation of eigenvectors from the Schur form is typically advantageous for parallel and larger computations.

	sherman3		rajat03		graham1		sinc15		ex11	
	25%	50%	25%	50%	25%	50%	25%	50%	25%	50%
eigenvectors by inverse iteration	6.25 0.57	12.18 0.87	19.92 1.56	39.03 2.74	31.91 2.39	64.32 4.39	75.69 4.79	148.7 8.89	146.7 10.8	291.6 20.24
eigenvectors by backsolve and back-transform	2.87 0.22	4.80 0.37	9.04 0.64	14.32 0.99	14.77 1.05	24.0 1.66	67.7 3.54	125.8 6.39	94.6 5.82	155.7 9.08

Table 6: Runtimes in seconds (top: sequential, bottom: using 28 cores) of computing 25% and 50% of the eigenvectors, respectively.

6 Conclusion

Eigenvectors of a non-symmetric matrix can be computed directly from the Schur matrix or from the Hessenberg matrix by inverse iteration. Both approaches have been expressed as tiled task-parallel algorithms that are rich in level-3 BLAS operations. The eigenvector computation from the Schur matrix requires more flops than inverse iteration, but the underlying operations benefit from calls to highly-optimized standard BLAS routines, in particular `GEMM`. Furthermore, the approach benefits from a high degree of parallelism. The computation of eigenvectors by inverse iteration, by contrast, is cheap in terms of flops, but not all computational steps are available in the BLAS.

This work added improvements to the inverse iteration route and evaluated under which circumstances which approach toward eigenvectors is advantageous. Experiments suggest that the preferred approach mostly depends on the performance of the QR algorithm. The inverse iteration route can be advantageous for small, serial problems. For larger problems, the computation of eigenvector from the Schur form is typically fastest, even when executed serially. In these cases the flop savings of the inverse iteration route cannot compensate that the operations executed during the computation of eigenvector from the Schur form run at a higher fraction of the peak performance. In the experiments conducted in this work, the QR algorithm with accumulations via `starneig_SEP_SM_Schur` (StarNEig v0.1.7) followed by the eigenvector computation from the Schur form is in most cases the method of choice.

Acknowledgments The author thanks Lars Karlsson for supporting this project. The author is grateful to Carl Christian Kjelgaard Mikkelsen for providing comments on this manuscript. 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 2020/5-286 and SNIC 2020/5-29.

Bibliography

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[2] Christopher Beattie, Zlatko Drmavč, and Serkan Gugercin. A note on shifted Hessenberg systems and frequency response computation. *ACM Trans. Math. Softw.*, 38(2), January 2012.

[3] Mark R Fahey. New Complex Parallel Eigenvalue and Eigenvector Routines. LAPACK Working Note 153, Computational Migration Group, Computer Science Corporation, August 2001.

[4] Mark Gates, Azzam Haidar, and Jack Dongarra. Accelerating computation of eigenvectors in the dense nonsymmetric eigenvalue problem. In *International Conference on High Performance Computing for Computational Science*, pages 182–191. Springer, 2014.

[5] Gene H Golub and Charles F Van Loan. *Matrix Computations.* John Hopkins University Press, 4rd edition, 2013.

[6] Greg Henry. The Shifted Hessenberg System Solve Computation. Technical report, Cornell University, NY, USA, 1994.

[7] Greg Henry. A Parallel Unsymmetric Inverse Iteration Solver. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 546–551, 1995.

[8] Mirko Myllykoski and Carl Christian Kjelgaard Mikkelsen. Task-based, GPU-accelerated and robust library for solving dense nonsymmetric eigenvalue problems. *Concurrency and Computation: Practice and Experience*, page e5915, 2020.

[9] Stephen L Olivier, Bronis R De Supinski, Martin Schulz, and Jan F Prins. Characterizing and mitigating work time inflation in task parallel programs. *Scientific Programming*, 21(3-4):123–136, 2013.

[10] Angelika Schwarz. Robust level-3 BLAS Inverse Iteration from the Hessenberg Matrix. *arXiv preprint arXiv:2101.05063*, 2021.

[11] Angelika Schwarz, Carl Christian Kjelgaard Mikkelsen, and Lars Karlsson. Robust parallel eigenvector computation for the non-symmetric eigenvalue problem. *Parallel Computing*, 100:102707, 2020.

[12] JM Varah. The Calculation of the Eigenvectors of a General Complex Matrix by Inverse Iteration. *Mathematics of Computation*, 22(104):785–791, 1968.