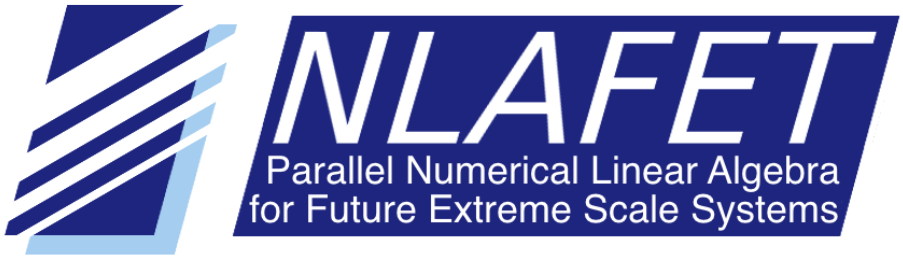


H2020–FETHPC–2014: GA 671633

D2.7

Eigenvalue solvers for nonsymmetric problems

April 2019

Document information

Scheduled delivery	2019-04-30
Actual delivery	2019-04-29
Version	1.0
Responsible partner	UMU

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2019-02-18	Bo Kågström	Draft	0.1	First layout of structure.
2019-03-11	Bo Kågström	Draft	0.2	Introduction and structure, more text added.
2019-04-15	Mirko Myllykoski	Draft	0.3	Results from computational experiments, reorganized structure, more text added. For internal review.
2019-04-29	Mirko Myllykoski	Final	1.0	Missing data points and GPU results added, internal review comments addressed. Final version.

Author(s)

Mirko Myllykoski (UMU)
Carl Christian Kjelgaard Mikkelsen (UMU)
Angelika Schwarz (UMU)
Bo Kågström (UMU)

Internal reviewers

Jan Papez (INRIA)
Srikara Pranesh (UNIMAN)

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

List of Figures

List of Tables

1 Introduction

The *Description of Action* document states for deliverable D2.7:

> "*D2.7 Eigenvalue solvers for nonsymmetric problems*
>
> Evaluation of new eigenvalue solvers for the nonsymmetric eigenvalue problem, including Krylov methods."

This deliverable is in the context of Task 2.3 (*Eigenvalue problem solvers*). The report builds and extends on the deliverables D2.5 *Eigenvalue problem solvers* and D2.6 *Prototype software for eigenvalue problem solvers*. Given $A \in \mathbb{R}^{n\times n}$ the standard eigenvalue problem (SEP) consists of computing eigenvalues $\lambda \in \mathbb{C}$ (possibly including zero) and eigenvectors $x \in \mathbb{C}^n$ such that

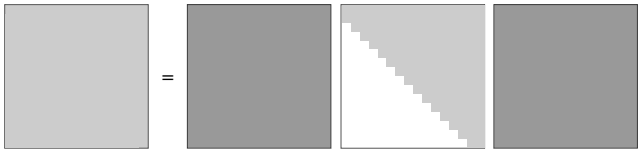
$$Ax = \lambda x. \tag{1}$$

Given $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times n}$ the generalized eigenvalue problem (GEP) consists of computing $\lambda \in \mathbb{C} \cup \{\infty\}$ and eigenvectors $x \in \mathbb{C}^{n\times n}$ such that

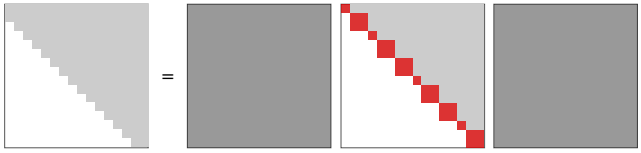
$$Ax = \lambda Bx. \tag{2}$$

In our case, A and B are dense and nonsymmetric matrices. Our goal is provide software which can compute all eigenvalues as well as invariant subspaces and/or eigenvectors associated with the user's selection of eigenvalues. We rely on an established approach which has been successful in the past: the application of two sided transformation algorithms acting on A or (A, B), respectively. We generalize and extend this to novel and effective task-based algorithms. The solution process for the above types of eigenvalue problems includes the following main steps:

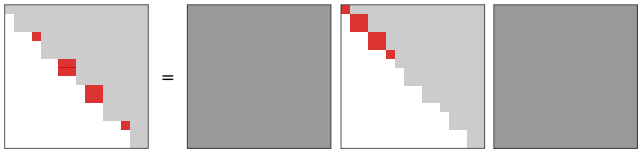
Step 1 Reduction to condensed forms (Hessenberg or Hessenberg-triangular forms)



Step 2 Reduction to (standard or generalized) Schur forms



Step 3 Eigenvalue reordering and subspaces (invariant or deflating subspaces)



Step 4 Computation of eigenvectors (associated with SEP or GEP)

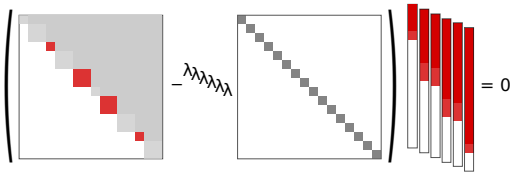


Table 1: Current status of the `StarNEig` library for standard eigenvalue problems.

Step	Shared memory	Distributed memory	GPUs
Hessenberg	Complete	ScaLAPACK	Single GPU
Schur	Complete	Experimental	Experimental
Reordering	Complete	Complete	Experimental
Eigenvectors	Complete	Integration ongoing	Not planned

Table 2: Current status of the `StarNEig` library for generalized eigenvalue problems.

Step	Shared memory	Distributed memory	GPUs
Hessenberg	LAPACK	ScaLAPACK	Not planned
Schur	Complete	Experimental	Experimental
Reordering	Complete	Complete	Experimental
Eigenvectors	Complete	Integration ongoing	Not planned

The algorithms for Steps 1, 2, and 3 are based on two-sided matrix transformations (i.e., multiplicative updates applied from both the left and the right on A or (A, B)). This approach often leads to complex data dependencies and limited concurrency. This is especially true for Step 2: the (iterative) multishift QR [5, 6, 8, 9] and QZ algorithms where the convergence is accelerated using Aggressive Early Deflation (AED) [3, 4, 10]. In contrast, the data dependencies in Step 4 are rather simple, but there are numerical issues which are nontrivial to address in a parallel setting. Altogether, the computations performed in Steps 1–4 can be challenging to run efficiently on today's and future extreme-scale HPC systems.

In this final deliverable of Task 2.3, we report progress on medium- to large-scale dense nonsymmetric eigenvalue problems, SEP and GEP. Relevant introductory material is covered in deliverables D2.5 and D2.6. The rest of the report is outlined as follows: Section 2 summarizes the current status of the software, including progress since D2.5 and D2.6. Section 3 describes the experimental setup as well as the hardware. Sections 4, 5, 6, and 7 cover the Steps 1, 2, 3, and 4, respectively. Summary and some final conclusions are given in Section 8.

2 Current status of the software

The new software evaluated in this report is available as the `StarNEig` library though the NLAFET GitHub platform `https://github.com/NLAFET/StarNEig`[1]. The `StarNEig` library is currently in a beta state and supports real arithmetic. Algorithmically, real arithmetic is more challenging than complex arithmetic due to the need to distinguish between real and pair of complex conjugate eigenvalues. See deliverable D7.8 *Release of the NLAFET library* for the `StarNEig` User Guide as well as algorithmic and implementation details.

The overall status of the library is summarized in Table 1 (SEP) and Table 2 (GEP). The *Experimental* status indicates that the software component has not been tested as extensively as those software components that are considered *Complete*. In addition, the GPU functionality requires some additional involvement from the user (performance

[1]The delivered software is available as version `0.1-beta.2` at `https://github.com/NLAFET/StarNEig/releases/tag/v0.1-beta.2`.

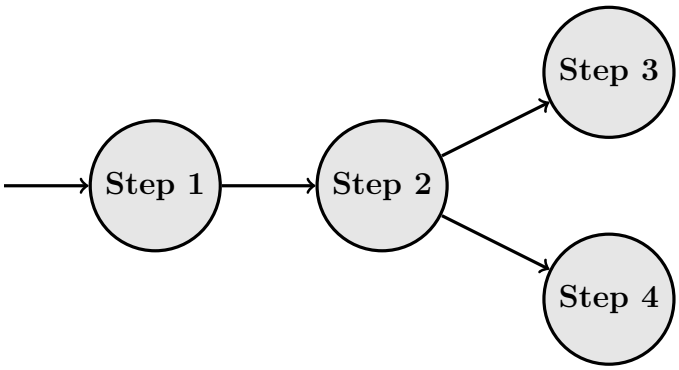


Figure 1: An illustration of the experimental data flow. The (generalized) Schur form is either reordered (Step 2 → Step 3) or the (generalized) eigenvectors are computed directly from it (Step 2 → Step 4).

model calibration). Known problems are listed in the `StarNEig` User Guide. Since D2.6, the `StarNEig` library has gained a new unified Application Programming Interface (API) and the functionality has been expanded. In particular, the library can now compute a generalized Schur decomposition using the QZ algorithm and supports parallel AED in QR and QZ. Functionality for computing both standard and generalized right-hand side eigenvectors in shared memory has also been integrated into the library. Furthermore, since D2.5, the library has gained the functionality to reorder the eigenvalues of a matrix in generalized real Schur form. Some interface functions are implemented as `LAPACK` and `ScaLAPACK` wrappers and the library thus now provides almost a full suite of functionality for dealing with non-symmetric eigenvalue problems in both shared and distributed memory, see Tables 1 and 2.

3 Experimental setting

Our main goal for this deliverable is to demonstrate that the `StarNEig` library can perform all four steps correctly and efficiently. For this purpose, we have designed experiments where we start from a set of matrices and matrix pairs and run them through all four steps. The overall data flow is illustrated in Figure 1. The entries of these matrices and matrix pairs were randomly generated and are uniformly distributed over the interval $[-1, 1]$. We have found that this class of matrices behaves very nicely in the standard case. In particular, the run-time is not sensitive to changes in random seed or to small changes in dimension of the problem.

A much more extensive set of experiments involving matrices and matrix pairs from real world applications would have provided a more comprehensive picture of the capabilities of the `StarNEig` library. However, since two matrices of similar size, but different origin, can exhibit drastically different convergence behaviour during Step 2, and Step 3 can actually fail in certain cases, we would have had to include significantly more matrices and matrix pairs in order to obtain a representative sample. This would have been too costly in terms of CPU core hours[2] and man-hours. Furthermore, we would like to point out that the generated eigenvalue problems are not trivial. In particular, the larger generalized eigenvalue problems can become very ill-conditioned and can thus be used to demonstrate that the `StarNEig` library can handle such problems.

[2]The production of this deliverable required in excess of 200 000 core hours.

We also compare our task-based software against state-of-the-art MPI-based software and demonstrate the scalability of our task-based software through scalability experiments. Most computational experiments are performed in distributed memory.

As the Steps 1, 2 and 3 consist of series of similarity transformations of the form

$$A = QXQ^T \tag{3}$$

or

$$A = QXZ^T \text{ and } B = QYZ^T, \tag{4}$$

we must show that each step retains the similarity/equivalence. For this purpose, we measure the following relative residuals after each step:

$$R_{sep,A}(X) = \frac{||QXQ^T - A||_F}{u||A||_F}, \tag{5}$$

$$R_{gep,A}(X) = \frac{||QXZ^T - A||_F}{u||A||_F}, \tag{6}$$

$$R_{gep,B}(Y) = \frac{||QYZ^T - B||_F}{u||B||_F}. \tag{7}$$

Here, $||\cdot||_F$ is the Frobenius norm and u is the double precision floating-point unit roundoff ($u = 2^{-52} \approx 2.22 \cdot 10^{-16}$). In addition, we measure the loss of orthogonality as follows:

$$R_{orth}(U) = \frac{||UU^T - I||_F}{u||I||_F}. \tag{8}$$

Note that the relative residuals are always computed with respect to the original random matrix A or the original random matrix pair (A, B). In addition, the orthogonal transformations Q and Z are accumulated across all steps.

For Step 4, we compute the following relative residual for each computed eigenpair (λ, x):

$$E_{sep}(\lambda, x) = \frac{||Ax - \lambda x||_2}{u(||A||_F + |\lambda|)||x||_2}. \tag{9}$$

In the generalized case, all generalized eigenvalues are expressed as $\lambda = \alpha/\beta$ and the computed relative residual is

$$\begin{aligned} E_{gep}(\lambda, x) &= \frac{||Ax - \lambda Bx||_2}{u(||A||_F + |\lambda|||B||_F)||x||_2} \\ &= \frac{||\beta Ax - \alpha Bx||_2}{u(|\beta|||A||_F + |\alpha|||B||_F)||x||_2}. \end{aligned} \tag{10}$$

If the eigenvalue is zero ($\alpha = 0$), we have

$$E_{sep}(\lambda, x) = E_{gep}(\lambda, x) = \frac{||Ax||_2}{u||A||_F||x||_2}. \tag{11}$$

In the case of infinite eigenvalue, i.e., $\beta = 0$, we compute the relative residual as

$$E_{gep}(\lambda, x) = \frac{||Bx||_2}{u||B||_F||x||_2}. \tag{12}$$

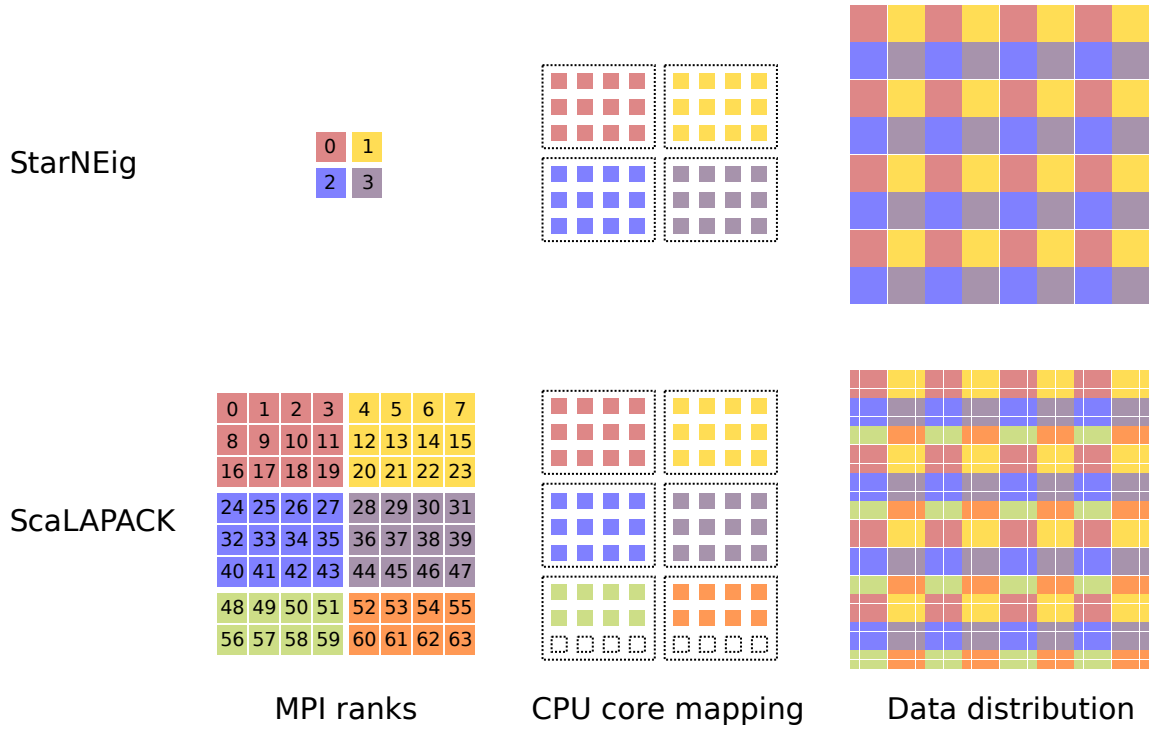


Figure 2: Illustrations of CPU core mappings and data distributions. With `StarNEig` each MPI process is mapped to a full node (12 cores per node in this example). With `ScaLAPACK` each MPI process is mapped to a single CPU core.

All computational experiments were performed on the Kebnekaise system, located at the High Performance Computing Center North (HPC2N), Umeå University. Kebnekaise is a heterogeneous systems consisting from many different types of compute nodes. The compute node types relevant for this deliverable are:

Regular compute node contains 28 Intel Xeon E5-2690v4 cores organized into 2 NUMA islands with 14 cores each and 128 GB memory. The nodes are connected with FDR Infiniband.

Large memory node contains 72 Intel Xeon E7-8860v4 cores organized into four NUMA islands with 18 cores each and 3072 GB memory.

V100 GPU node contains 28 Intel Xeon Gold 6132 cores organized into 2 NUMA islands with 14 cores each and 192 GB memory. Each node contains two NVIDIA Tesla V100 GPUs.

Most experiments were performed on the regular nodes but the eigenvector experiments were performed on the large memory nodes and the GPU experiments on were performed on the V100 GPU nodes. The software was compiled with GCC 7.3.0 compiler and linked against OpenMPI 3.1.3, OpenBLAS 0.3.2, ScaLAPACK 2.0.2, CUDA 9.2.88, and StarPU 1.2.8 [1] libraries.

All distributed memory experiments were performed using a square MPI process grid[3] and the matrices were distributed in two-dimensional block cyclic fashion as illustrated in Figure 2. One major difference between `StarNEig` interface functions and `ScaLAPACK`-style subroutines is that `StarNEig` is designed to run in a multi-threaded environment, while

[3] A square process grid usually leads to more consistent performance with `ScaLAPACK`-style subroutines.

many ScaLAPACK-style subroutines perform optimally in a single-threaded environment. This can create a conflict if we want to continue using a square MPI process grid since finding a CPU core allocation that would lead to a square MPI process grid in both approaches is not straightforward. Our solution is to always favour the ScaLAPACK-style subroutine in the comparisons as illustrated in Figure 2. We always map each StarNEig process to a full node (28 cores) and distributes the data in large blocks. We always map each ScaLAPACK-style process to a single CPU core and distributes the data in 160×160 blocks. The total number of CPU cores in each ScaLAPACK experiment is always equal or larger than the total number of CPU cores in the corresponding StarNEig experiment.

4 Reduction to Hessenberg or Hessenberg-triangular forms

Given a general matrix A, the StarNEig interface functions

- `starneig_SEP_SM_Hessenberg()` and
- `starneig_SEP_DM_Hessenberg()`

compute a Hessenberg decomposition

$$A = QHQ^T, \tag{13}$$

where H is upper Hessenberg and Q is orthogonal.

Given a general matrix pair (A, B), the StarNEig interface functions

- `starneig_GEP_SM_HessenbergTriangular()` and
- `starneig_GEP_DM_HessenbergTriangular()`

compute a Hessenberg-triangular decomposition

$$A = QHZ^T, \quad B = QRZ^T, \tag{14}$$

where H is upper Hessenberg, R is upper triangular, and Q and Z are orthogonal.

The shared memory variant `starneig_SEP_SM_Hessenberg` is implemented on top of StarPU and has already been discussed extensively in D2.6 (some of the reported GPU performance issues were related to StarPU and have since been solved). The other variants are implemented as LAPACK and ScaLAPACK wrapper functions. Thus, only the residuals after computing the Hessenberg and Hessenberg-triangular forms are reported here; see Tables 3 and 4. We note that the residuals from Step 1 are quite small and thus the generated Hessenberg and Hessenberg-triangular forms can be safely used as input for Step 2.

Table 3: Residuals after computing Hessenberg forms. Residuals are reported in multiples of the double precision floating-point unit roundoff; see (5) and (8).

n	$R_{sep,A}(H)$	$R_{orth}(Q)$
10 000	36	20
20 000	50	27
40 000	34	20
60 000	55	24
80 000	50	20
100 000	55	22
120 000	52	19

Table 4: Residuals after computing Hessenberg-triangular forms. Residuals are reported in multiples of the double precision floating-point unit roundoff; see (6), (7) and (8).

n	$R_{gep,A}(H)$	$R_{gep,B}(R)$	$R_{orth}(Q)$	$R_{orth}(Z)$
10 000	90	91	73	67
20 000	131	132	103	100
40 000	180	181	145	135
60 000	168	226	177	112
80 000	286	270	205	235

5 Reduction to standard or generalized Schur forms

We start by considering the standard eigenvalue problem. Given a Hessenberg decomposition $A = QHQ^T$, the `StarNEig` interface functions

- `starneig_SEP_SM_Schur()` and
- `starneig_SEP_DM_Schur()`

compute a Schur decomposition

$$A = Q(USU^T)Q^T, \tag{15}$$

where the Schur matrix S is upper quasi-triangular with 1×1 and 2×2 blocks on the block diagonal and U is orthogonal. On exit, Q is overwritten by $Q \leftarrow QU$.

Both interface functions are implemented on top of `StarPU`. The shared memory variant `starneig_SEP_SM_Schur` was already discussed in D2.6 but major improvements have been made since. In particular, D2.6 demonstrated that the sequential AED step (see Figure 3) forms a bottleneck for the scalability of the algorithm. The AED step is now implemented in a task-based manner and runs in parallel. The QR algorithm is recursive. In particular, every AED step applies the QR algorithm and Hessenberg reduction to the AED window. Both are handled by calls to `StarNEig`. The major new component is the parallel implementation of the intermediate deflation checks and the required reordering steps. The basic idea of the parallel task-based AED algorithm is visualized in Figure 4.

The `starneig_SEP_DM_Schur` interface function was compared against the state-of-the-art MPI-based `PDHSEQR` subroutine [9]. The results are presented in Table 5 and Figure 5. The `StarNEig` interface function outperforms the `PDHSEQR` subroutine (in terms of the speed) in all cases. In particular, `StarNEig` is almost three times faster for the

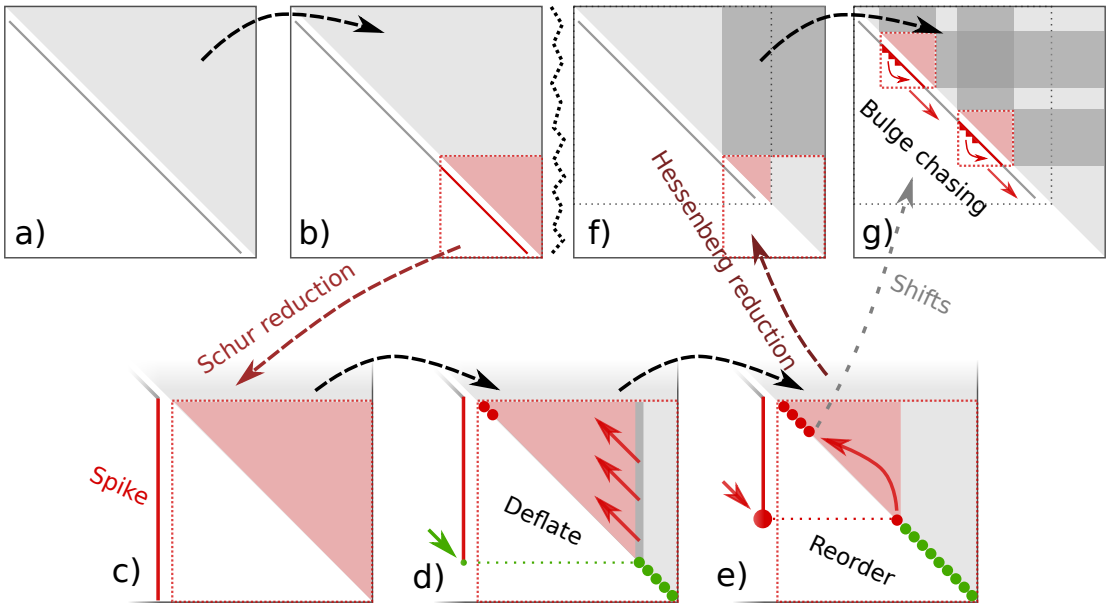


Figure 3: An illustration of a single iteration of the multishift QR algorithm with AED. Starting from the left: a) the original A in Hessenberg form H; b) an AED window is placed at the lower right corner (highlighted in red); c) the AED window is reduced to Schur form and a spike is formed; d) tiny element in the spike lead to a converged eigenvalue that is successfully deflated by setting the matching element in the spike to zero (highlighted in green); e) a failed eigenvalue candidate is reordered to the top left corner of the AED window (highlighted in red); f) the remaining AED window is reduced back to upper Hessenberg form; and g) pipelined QR iteration with two bulge-chasing windows. Note how the active region (dashed outline) shrinks after the AED process managed to deflate a set of eigenvalues. The similarity transformations from each diagonal computational window are accumulated and applied as matrix-matrix multiplications in associated submatrix updates.

largest considered problem ($n = 120\,000$). Based on the data gathered from the computational experiments, the larger problems seem to have a larger probability of deflating into several independent subproblems as the algorithm proceeds. As part its basic design, `StarNEig` is capable of detecting and processing several independent subproblem concurrently. Naturally this increases parallelism and reduces the run-time, as demonstrated in Table 5 and Figure 5. The basic ideas have been explained in D2.6.

The scalability of the `starneig_SEP_DM_Schur` interface function was investigated separately. The results are presented in Figure 6. When we move from a single node to four nodes the performance is roughly doubled. For problems that are larger than $n = 100\,000$, we observe good speedups for even larger number of nodes. It should be noted that the QR algorithm is an iterative method and the nature of the input matrix can have a huge impact on the convergence rate. In particular, based on the data gathered from the computational experiments, it appears that Hessenberg matrices computed from dense matrices with entries distributed uniformly typically require only a few bulge chasing steps. The QR algorithms ends up performing several consecutive AED steps between the bulge chasing steps thus leading to faster convergence. However, the bulge chasing steps are the main source of parallelism in the QR algorithm, which explains why this type of random test problems do not scale particularly well.

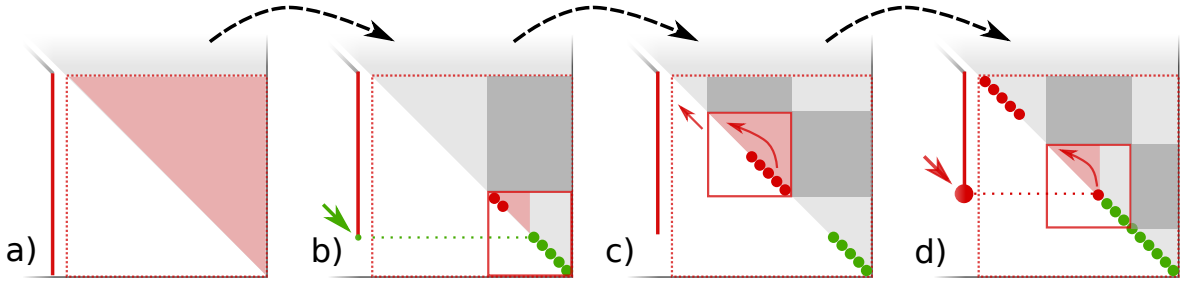


Figure 4: An illustration of a parallel AED step. From left to right: a) the AED window; b) a small deflation window is placed at the lower right corner and the involved eigenvalue candidates are either deflated or moved to the top left corner of the deflation window; c) a series of small reordering windows are placed on the diagonal (only one illustrated here) and the failed eigenvalues from the deflation window are moved together to the upper left corner of the AED window; then d) the next deflation window is placed on the diagonal. The similarity transformations from each diagonal computational window are accumulated and applied as matrix-matrix multiplications in associated submatrix updates. In practice, the deflation and reordering windows are relatively small and the deflation and reordering steps can overlap and thereby enhance the potential parallelism.

Table 5: A comparison between `PDHSEQR` (left in each column-pair) and `starneig_SEP_DM_Schur` (right in each column-pair). Residuals are reported in multiples of the double precision floating-point unit roundoff; see (5) and (8).

n	CPU cores		Run-time ($secs$)		$R_{sep,A}(S)$		$R_{orth}(Q)$	
10 000	36	1 · 28 (28)	38	18	145	189	100	130
20 000	36	1 · 28 (28)	158	85	191	248	130	169
40 000	36	1 · 28 (28)	708	431	120	314	161	217
60 000	121	4 · 28 (112)	992	563	285	369	191	249
80 000	121	4 · 28 (112)	1667	904	279	419	182	297
100 000	121	4 · 28 (112)	3319	1168	240	397	163	269
120 000	256	9 · 28 (252)	3268	1111	288	442	184	318

We now turn the attention to the generalized eigenvalue problem. Given a Hessenberg-triangular decomposition $(A, B) = Q(H, R)Z^T$, the `StarNEig` interface functions

- `starneig_GEP_SM_Schur()` and
- `starneig_GEP_DM_Schur()`

compute a generalized Schur decomposition

$$A = Q(U_1 S U_2^T)Z^T, \quad B = Q(U_1 T U_2^T)Z^T, \tag{16}$$

where S is upper quasi-triangular with 1×1 and 2×2 blocks on the diagonal, T is a upper triangular, and U_1 and U_2 are orthogonal. On exit, Q and Z are overwritten by $Q \leftarrow QU_1$ and $Z \leftarrow ZU_2$. Contrary to the standard case, each generalized eigenvalue is represented as a pair of numbers (α, β) such that the actual generalized eigenvalue λ is given by α/β for $\beta \neq 0$ (finite eigenvalue). The case $\beta = 0$ corresponds to an infinite eigenvalue, which appears if the matrix B is singular. If both $\alpha = 0$ and $\beta = 0$, then the eigenvalue is indefinite/undefined. In finite precision arithmetic, a highly ill-conditioned

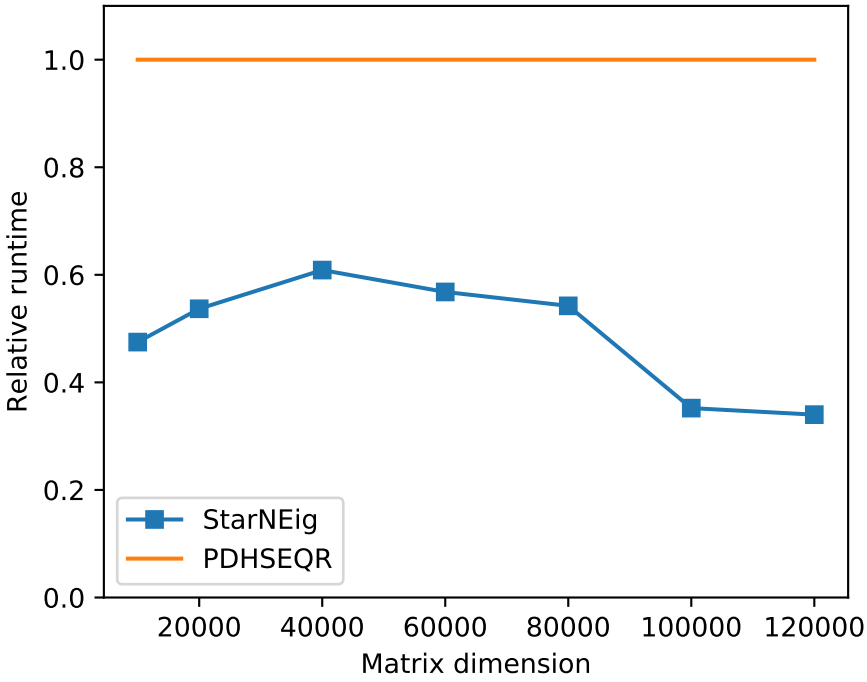


Figure 5: Relative run-time improvement of `starneig_SEP_DM_Schur` with respect to `PDHSEQR` (base at 1.0). The used CPU core count always favours `PDHSEQR` (see Table 5).

generalized eigenvalue problem can have both infinite and indefinite eigenvalues. The latter corresponds to a singular eigenvalue problem. In case of a real, an infinite or an indefinite eigenvalue, the α and β values can be read off directly from the diagonals of S and T, respectively.

Both interface functions are implemented on top of `StarPU`. The basic theory and the main computational steps for SEP are generalized and extended to GEP. One significant difference is that the GEP algorithm attempts to identify and isolate any infinite eigenvalues. The standard convention[4] is to classify a real eigenvalue as infinite if the corresponding diagonal entry of T has magnitude smaller than $u||T||_F$. Any infinite eigenvalues can be reordered to the upper left (or bottom right) corner of the Hessenberg-triangular form and deflated immediately. This deflation not only reduces the size of the remaining sub-problem, it also improves the rate of convergence of the algorithm. The detection of infinite eigenvalues is performed during the QZ bulge chasing stage and, if necessary, a set of special reordering tasks is inserted once a large enough portion of the bulge chasing tasks have completed. The detected infinite eigenvalues are ordered to the upper left corner of the matrix, which allows the next AED to be overlapped with the reordering stage.

The `starneig_GEP_DM_Schur` interface function was compared against the state-of-the-art MPI-based `PDHGEQZ` subroutine [4, 2]. The results are presented in Table 6 and Figure 7. Table 7 show the number of (nearly) infinite and (nearly) indefinite eigenvalues in the final generalized Schur form. We see that, `StarNEig` is between 1.4 and 7.9 times faster than `PDHGEQZ`. The matrix dimensions $n = 60\,000$ and $n = 80\,000$ are particularly interesting since in both cases infinite eigenvalues were detected by both algorithms but in

[4]The `DHGEQZ` LAPACK subroutine uses an identical condition.

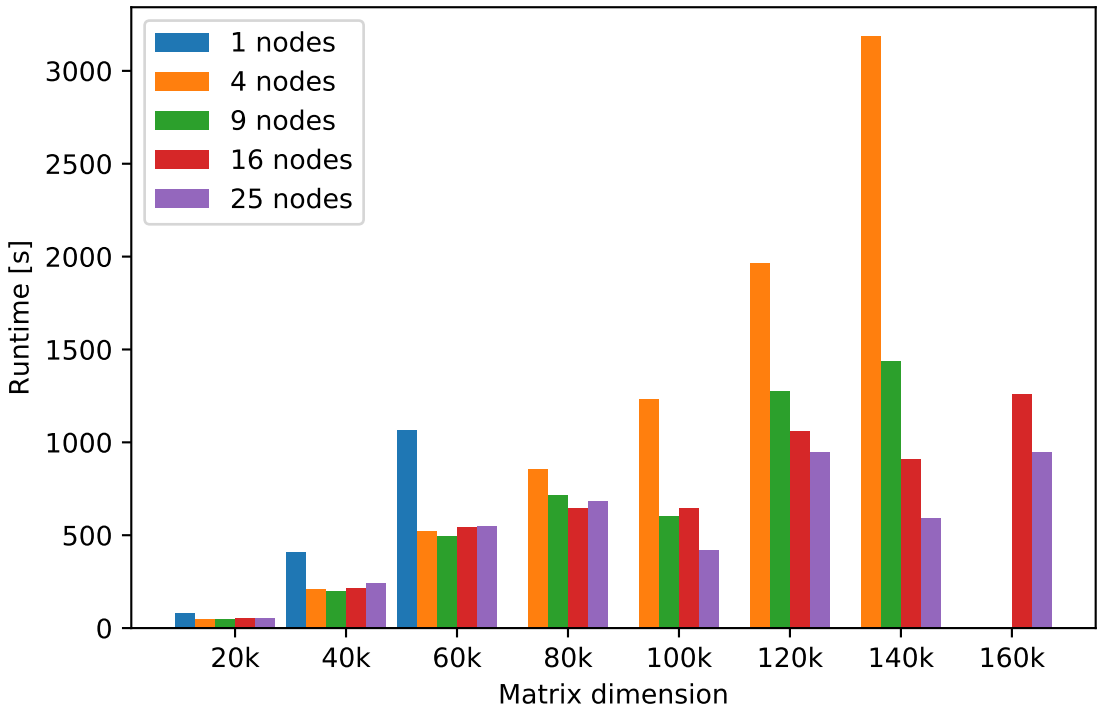


Figure 6: Scalability results for `starneig_SEP_DM_Schur`. Each node contains 28 cores.

Table 6: A comparison between `PDHGEQZ` (left in each column-pair) and `starneig_GEP_DM_Schur` (right in each column-pair). Residuals are reported in multiples of the double precision floating-point unit roundoff; see (6), (7) and (8).

n	CPU cores		Run-time *(secs)*		$R_{gep,A}(S)$		$R_{gep,B}(T)$		$R_{orth}(Q)$		$R_{orth}(Z)$	
10 000	36	28	57	23	356	363	354	350	117	117	325	339
20 000	36	28	199	64	523	542	519	526	149	154	480	515
40 000	36	28	903	308	917	803	913	781	209	199	856	772
60 000	121	112	746	94	178	175	248	234	187	178	127	118
80 000	121	112	3499	2527	4383	3570	5007	4068	291	320	3583	2891

the former case `StarNEig` was almost eight times faster and in the latter case `StarNEig` was only 1.4 times faster. In addition, `PDHGEQZ` managed to detect significantly more infinite eigenvalues[5] in both cases. The exact reason for this large difference in behavior is a topic for future analysis. However, we can make several educated guesses:

- The log-files clearly state that the case of $n = 60\,000$ decoupled into many independent sub-problems. This increased the level of parallelism and, perhaps more importantly, reduced the need for global communication. However, the $n = 80\,000$ case also deflated into many independent sub-problems.
- `PDHGEQZ` detected most of the infinite eigenvalues late in the $n = 60\,000$ case and thus benefited from the deflated infinite eigenvalues only to a limited extent.

[5]Some detected infinite eigenvalues are classified as nearly indefinite eigenvalues.

Table 7: The number of infinite and indefinite eigenvalues in the outputs of `PDHGEQZ` (left in each column-pair) and `starneig_GEP_DM_Schur` (right in each column-pair). The *Infinities* and *Nearly infinities* columns show the number of infinite eigenvalues ($t_{i,i} = 0$) and nearly infinite eigenvalues ($|t_{i,i}| < u||T||_F$), respectively. The *Indefinites* and *Nearly indefinites* columns show the number of indefinite ($|s_{i,i}| = 0$ and $|t_{i,i}| = 0$) and nearly indefinite eigenvalues ($|s_{i,i}| < u||S||_F$ and $|t_{i,i}| < u||T||_F$), respectively. The types of the classifications are disjoint.

n	Infinities		Nearly infinities		Indefinites		Nearly indefinites	
10 000	0	0	0	0	0	0	0	0
20 000	0	0	0	0	0	0	0	0
40 000	0	0	0	0	0	0	0	0
60 000	7758	4401	0	42	0	0	42632	15323
80 000	46145	23	0	2	0	0	10711	10717

- `PDHGEQZ` detected most of the infinite eigenvalues early in the $n = 80\,000$ case and thus benefited from the deflated infinite eigenvalues to a greater extent.

It should be noted that the fact that one code managed to detect more infinite eigenvalues does not, in general, imply that one computed result is more valid than the other. In particular, the computed residuals are essentially identical (see Table 6). Undetected infinite eigenvalues can cause the QZ algorithm to perform more iterations, thus adding more rounding errors to the final result, but this has not affected the residuals.

Since the behavior of the QZ algorithm appears to be less predictable compared to the QR algorithm, we decided to generate input matrices for the scalability experiment using the `Hessrand1` algorithm (see, e.g., [4, 5]) described below. Let $\mathcal{N}(\mu, \sigma)$ be the normal distribution with mean μ and variance σ, and let $\chi^2(\tau)$ be the chi-squared distribution with τ degrees of freedom. We generate the input matrix pair (H, R) as follows:

$$\begin{aligned}
h_{i,j} &\sim \mathcal{N}(0,1), && i = 1, \ldots, j, && j = 1, \ldots, n, \\
h^2_{i+1,i} &\sim \chi^2(n-i), && i = 1, \ldots, n-1, && \\
r_{i,j} &\sim \mathcal{N}(0,1), && i = 1, \ldots, j-1, && j = 1, \ldots, n, \\
r^2_{i,i} &\sim \chi^2(i-1), && i = 2, \ldots, n, && \\
r^2_{1,1} &\sim \chi^2(n). &&
\end{aligned} \tag{17}$$

The resulting matrix pairs (H, R) in Hessenberg-triangular form have reasonably well-conditioned eigenvalues and, thus, the convergence behaviour is expected to be more consistent. Scalability results for `starneig_GEP_DM_Schur` are presented in Figure 8. The results show that that larger problems benefit from using additional nodes, although with modest scalability for the largest problem considered ($n = 100\,000$) on 25 nodes. However, we expect improved scalability for larger problems. As with the standard case, only a few bulge chasing steps were required for convergence.

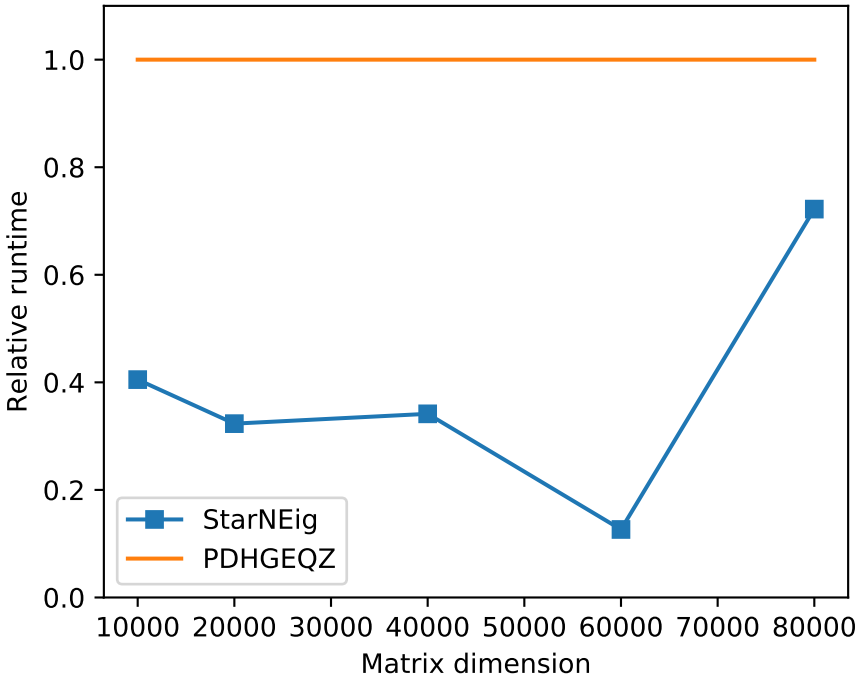


Figure 7: Relative run-time improvement of `starneig_GEP_DM_Schur` with respect to `PDHGEQZ` (base at 1.0). The used CPU core count always favours `PDHGEQZ` (see Table 6).

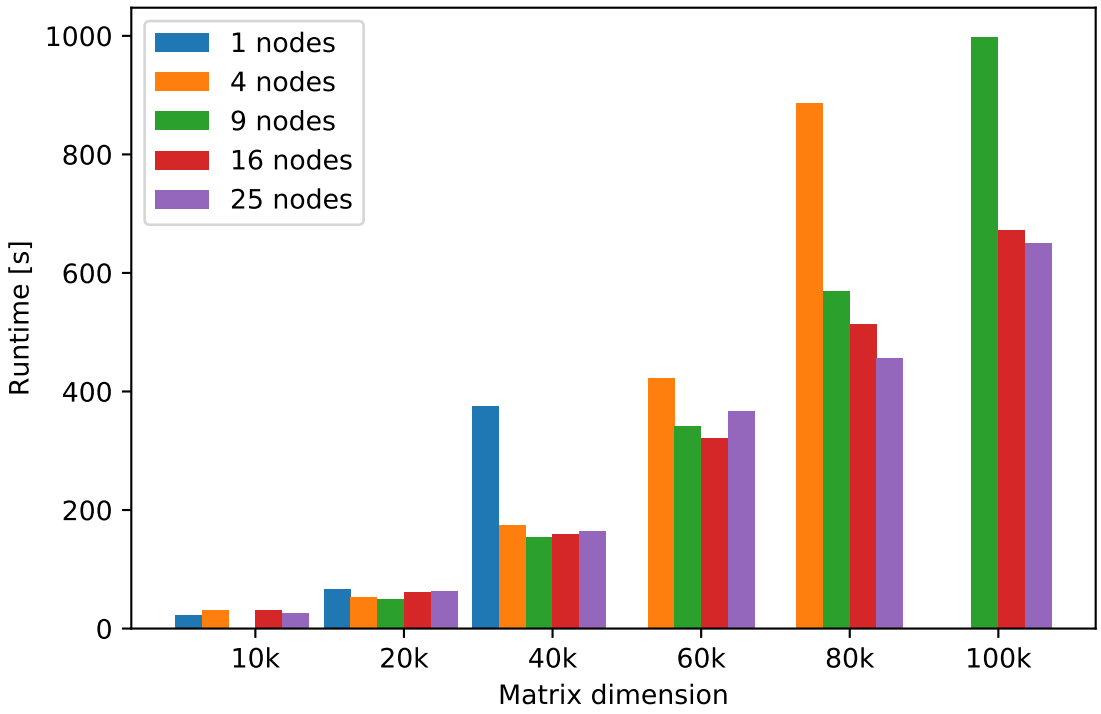


Figure 8: Scalability results for `starneig_GEP_DM_Schur`. Each node contains 28 cores.

6 Eigenvalue reordering and invariant subspaces

Given a Schur decomposition $A = QSQ^T$ and user's selection of eigenvalues, the `StarNEig` interface functions

- `starneig_SEP_SM_ReorderSchur()` and
- `starneig_SEP_DM_ReorderSchur()`

attempt to reorder the selected eigenvalues to the top left corner of an updated Schur matrix $\hat{S}$ by an orthogonal similarity transformation

$$A = Q(U\hat{S}U^T)Q^T. \tag{18}$$

On exit, Q is overwritten by $Q \leftarrow QU$. Both interface functions are implemented on top of `StarPU`. The shared memory variant `starneig_SEP_SM_ReorderSchur` was already discussed in D2.5 although major improvements have been made since (GPU support in particular).

The `starneig_SEP_DM_ReorderSchur` interface function was compared against the MPI-based `PDTRSEN` subroutine [7]. The results are presented in Table 8 and Figure 9. `StarNEig` outperforms `PDTRSEN` in all cases. In particular, `StarNEig` is almost five times faster for the largest considered problem ($n = 120\,000$). The results are very much in line with the shared memory results presented in [13, 14] and D2.5. We note that `StarNEig` generated a much larger $R_{sep,A}(\tilde{S})$ residual in the case $n = 40\,000$. At this point, we can only comment that the result is repeatable (we also repeated the `PDTRSEN` experiment) and is likely related to the numerical conditioning of the input matrix.

The scalability of the `starneig_SEP_DM_ReorderSchur` interface function was investigated separately. For time saving reasons, we constructed a set of well-conditioned input matrices[6] for the experiments. The results are presented in Figure 10. The results show that moving from a single node to four nodes typically almost quadruples the performance and a similar (but slightly weakening) trend continues all the way to 25 nodes. We find these results to be outstanding.

Now, we turn our attention to the generalized case. Given a generalized Schur decomposition $(A, B) = Q(S, T)Z^T$, the `StarNEig` interface functions

- `starneig_GEP_SM_ReorderSchur()` and
- `starneig_GEP_DM_ReorderSchur()`

attempt to reorder the selected generalized eigenvalues to the top left corner of an updated generalized Schur form $(\hat{S}, \hat{T})$ by an orthogonal similarity transformation

$$A = Q(U_1\hat{S}U_2^T)Z^T, \quad B = Q(U_1\hat{T}U_2^T)Z^T, \tag{19}$$

On exit, Q and Z are overwritten by $Q \leftarrow QU_1$ and $Z \leftarrow ZU_2$. Both interface functions are implemented on top of `StarPU`.

The `starneig_GEP_DM_ReorderSchur` interface function was compared against the MPI-based `PDTGSEN` subroutine [2]. The results are presented in Table 9. We performed two separate experiments as explained below:

[6] The separation between two eigenvalues is guaranteed to be at least 1.

Table 8: A comparison between `PDTRSEN` (left in each column-pair) and `starneig_SEP_DM_ReorderSchur` (right in each column-pair). 35% of the eigenvalues were randomly selected. Residuals are reported in multiples of the double precision floating-point unit roundoff; see (5) and (8).

n	CPU cores		Run-time (*secs*)		$R_{sep,A}(\tilde{S})$		$R_{orth}(Q)$	
10 000	36	1 · 28 (28)	12	3	214	214	146	146
20 000	36	1 · 28 (28)	72	25	287	286	195	194
40 000	36	1 · 28 (28)	512	180	332	3265	260	258
60 000	121	4 · 28 (112)	669	168	471	468	311	309
80 000	121	4 · 28 (112)	1709	391	554	551	374	372
100 000	121	4 · 28 (112)	3285	737	579	574	386	383
120 000	256	9 · 28 (252)	2902	581	662	660	446	443

Table 9: A comparison between `PDTGSEN` (left in each column-pair) and `starneig_GEP_DM_ReorderSchur` (right in each column-pair). 35% of the eigenvalues were randomly selected. The upper half shows the results for the main experiment and the lower half shows the results for a synthetic test problem. The large differences in run-times (upper half, $n = 60\,000$ and $n = 80\,000$) are due to differences how the two codes handle a failed swap of two diagonal blocks. Residuals are reported in multiples of the double precision floating-point unit roundoff; see (6), (7) and (8).

n	CPU cores		Run-time (*secs*)		$R_{gep,A}(\tilde{S})$		$R_{gep,B}(\tilde{T})$		$R_{orth}(Q)$		$R_{orth}(Z)$	
10 000	36	28	30	8	428	432	396	399	156	158	355	355
20 000	36	28	176	52	680	683	619	621	224	225	549	549
40 000	36	28	1176	364	1126	1125	1009	1009	383	382	856	856
60 000	121	112	7	324	175	186	234	254	178	187	118	128
80 000	121	112	64	729	3570	3672	4068	4178	320	381	2891	2923
10 000	36	28	35	8	72	70	97	95	67	66	67	66
20 000	36	28	141	51	86	99	117	133	82	92	80	92
40 000	36	28	594	375	113	144	125	196	99	135	80	135
60 000	121	112	1004	341	151	178	198	239	139	165	132	164
80 000	121	112	1417	778	163	210	188	285	132	195	121	195

1. The first experiment (see the upper half of Table 9) completes the upper branch of the main experiment (see Figure 1). The reason why we performed two separate experiments was that the larger matrices ($n = 60\,000$ and $n = 80\,000$) were so ill-conditioned that both algorithms failed to reorder the Schur form in its entirety. In particular, the nearly indefinite eigenvalues can cause many numerical problems. The *partially reordered* output matrices are still in generalized Schur form, as they should be, but the performance results are not comparable due to the differences in how the two codes handle a failed swap of two diagonal blocks. `PDTGSEN` aborts the reordering procedure after one MPI process detects a failed swap and the run-time can thus be just a few seconds. On the other hand, `StarNEig` attempts to continue the reordering procedure for those parts of the matrix that are not effected by the failed swap. The latter could thus potentially produce a more complete output. The answer to the question which approach is better depends on the application.

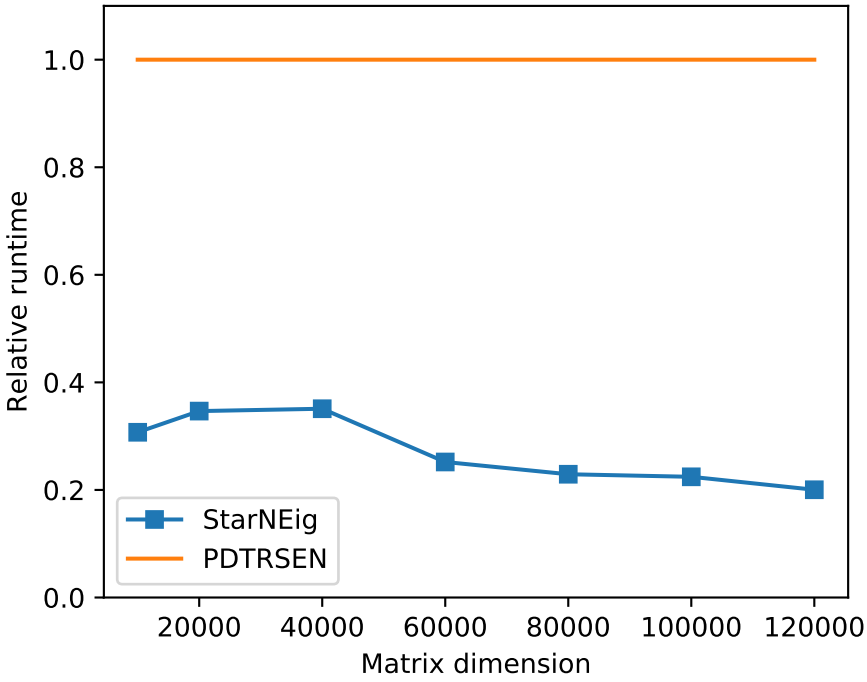


Figure 9: Relative run-time improvement of `starneig_SEP_DM_ReorderSchur` with respect to `PDTRSEN` (base at 1.0). 35% of the eigenvalues were randomly selected. The used CPU core count always favours `PDTRSEN` (see Table 8).

2. The second experiment (see the lower half of Table 9) attempts to correct for the shortcomings of the first experiment by constructing a set of better conditioned input matrices for which failure to swap is less likely. Figure 11 presents the results from the second experiment. Based on these numbers, `StarNEig` is between 1.6 to 4.4 times faster than `PDTGSEN`. The results of a scalability experiment are presented in Figure 12 and are inline with the standard case.

The GPU performance was investigated separately and the results are presented in Figure 13. We performed one set of experiments using a single CPU socket (14 cores) and a second set of experiments using two CPU sockets (28 cores). We note that the inclusion of a single GPU can improve the performance by a factor of 1.7 and the inclusion of two GPUs can improve the performance by a factor of 3.5. Surprisingly, a configuration consisting of one or two GPUs paired with a single CPU socket appears to give the best performance. A more complete understating need further analysis.

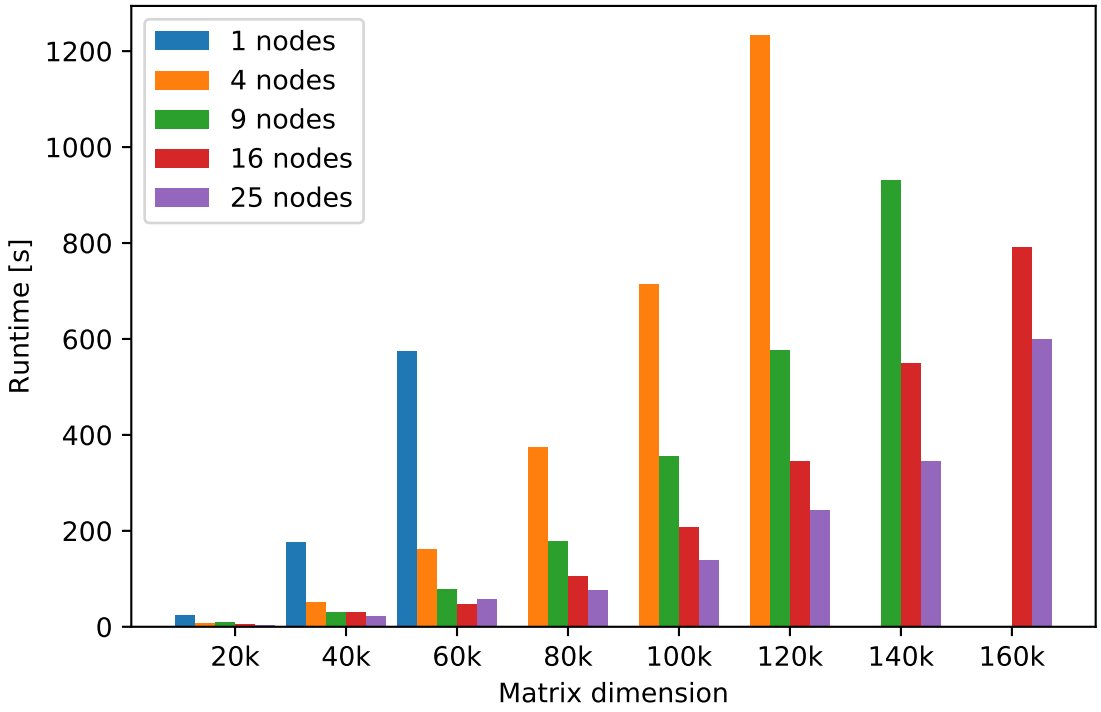


Figure 10: Scalability results for `starneig_SEP_DM_ReorderSchur`. 35% of the eigenvalues were randomly selected. Each node contains 28 cores.

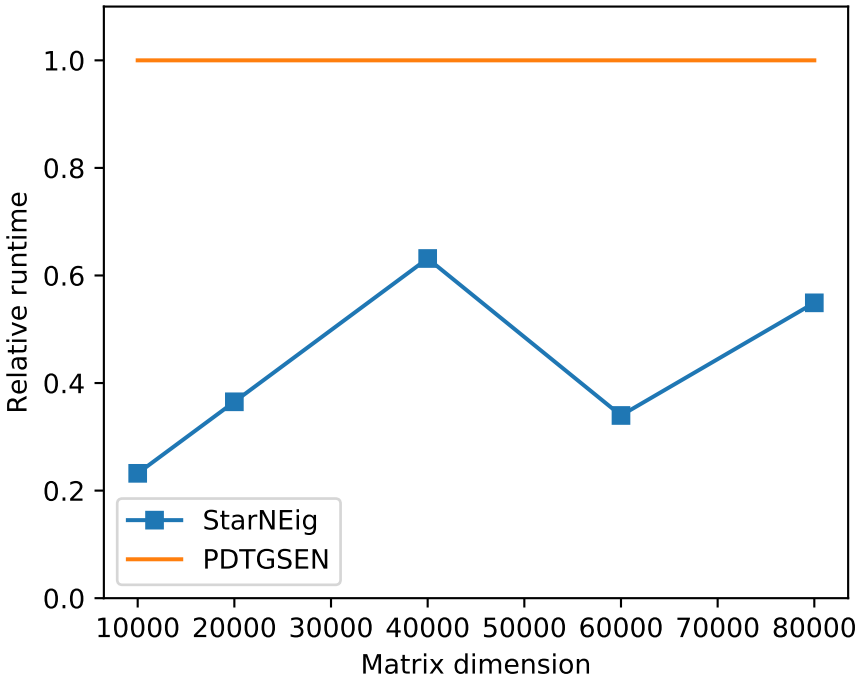


Figure 11: Relative run-time improvement of `starneig_GEP_DM_ReorderSchur` with respect to `PDTGSEN` (base at 1.0). 35% of the eigenvalues were randomly selected. The used CPU core count always favours `PDTGSEN` (see Table 9).

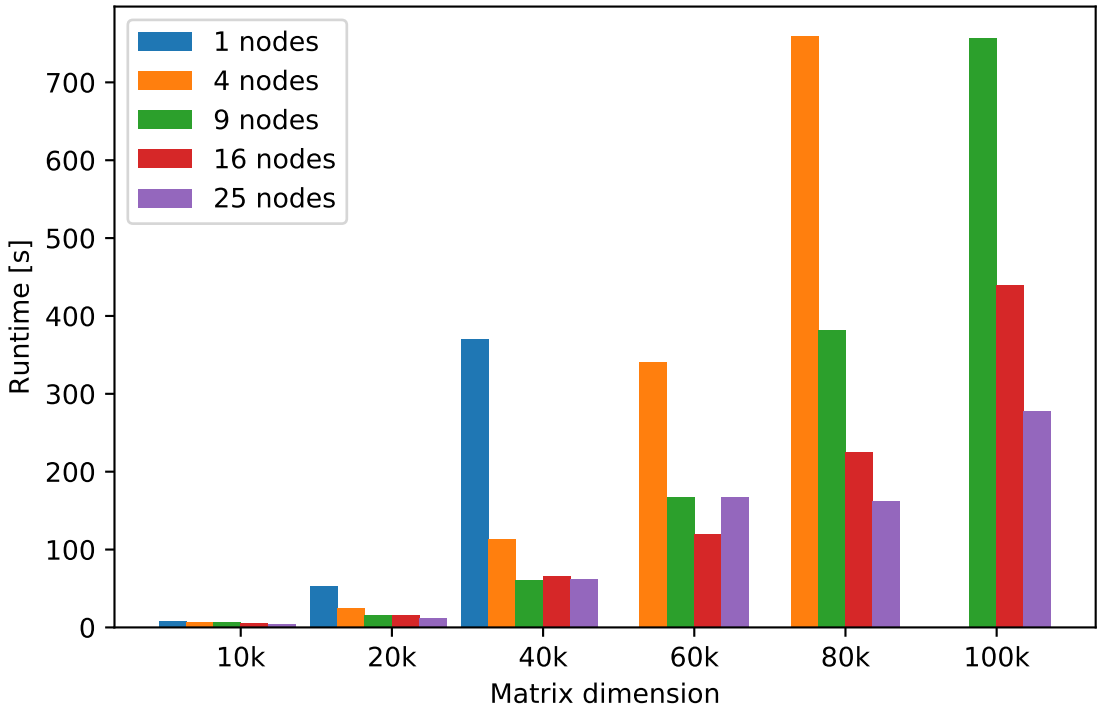


Figure 12: Scalability results for `starneig_GEP_DM_ReorderSchur`. 35% of the eigenvalues were randomly selected. Each node contains 28 cores.

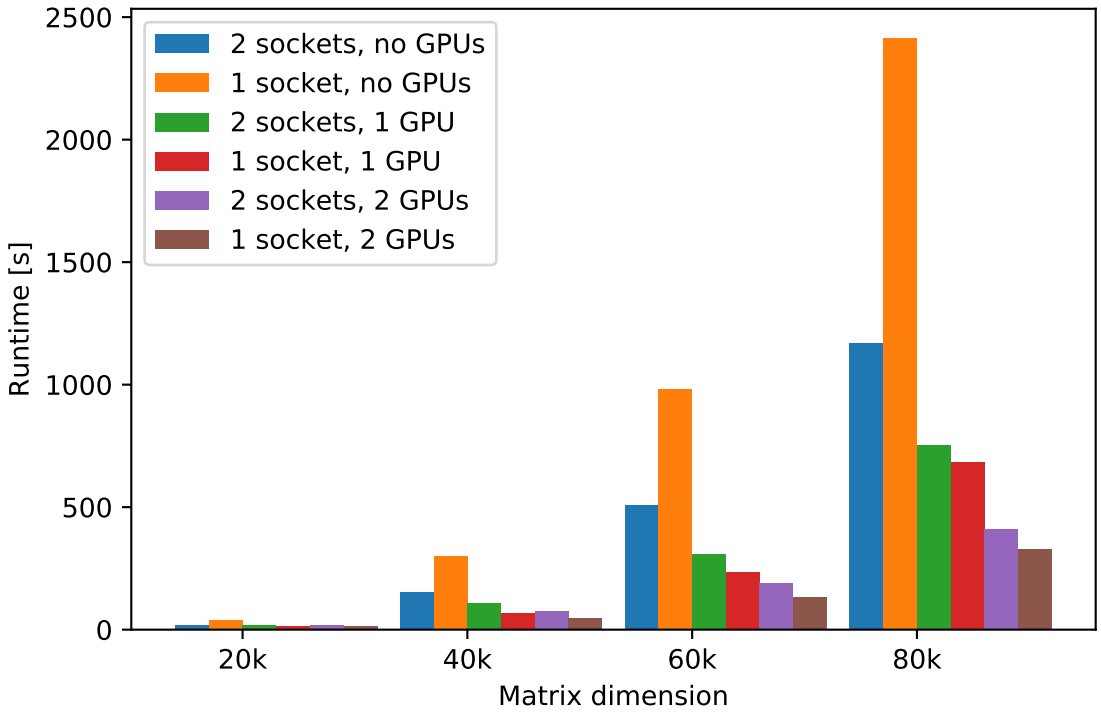


Figure 13: GPU performance results for `starneig_SEP_SM_ReorderSchur`. 35% of the eigenvalues were randomly selected. Each socket contains 14 cores.

7 Computation of selected eigenvectors

Given a Schur decomposition $A = QSQ^T$ and a user selection of eigenvalues, the `StarNEig` interface functions

- `starneig_SEP_SM_Eigenvectors()` and
- `starneig_SEP_DM_Eigenvectors()`

compute and return an eigenvector for each of the selected eigenvalues. The shared memory interface function `starneig_SEP_SM_Eigenvectors` is implemented on top of `StarPU`. The distributed memory variant `starneig_SEP_DM_Eigenvectors` still requires certain components before in can be integrated with the rest of the library. We emphasize that the selected eigenvectors are computed directly from the (generalized) real Schur forms and then backtransformed to the original basis. This is faster than first reordering the selected eigenvalues to the upper left corner and then computing then eigenvector from the upper left corner. This is illustrated in Figure 14.

Table 10 shows the run-time and residuals for `starneig_SEP_Eigenvectors` when the user has selected either 35% or 100% of all eigenvalues. The parallel scalability is illustrated in Figure 15 (35% selected) and Figure 16 (100% selected). We note that the relative residuals are always a modest multiple of the unit roundoff. Moreover, the code scales reasonably up to 64 cores.

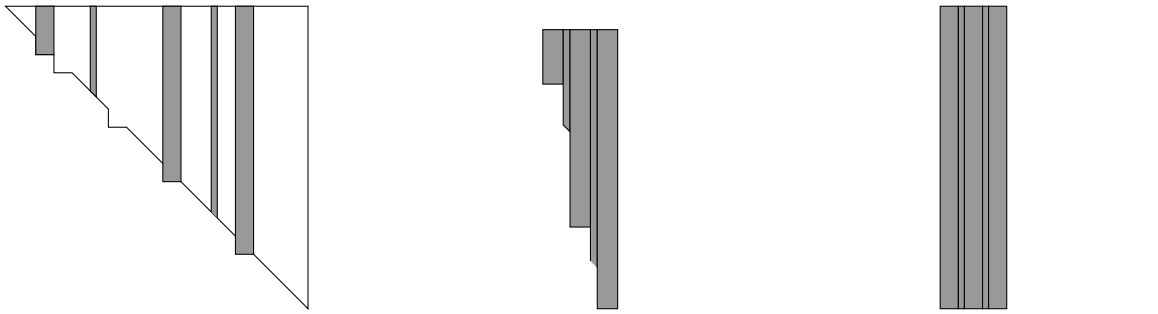
We now return to the generalized eigenvalue problem. Given a generalized Schur decomposition $(A, B) = Q(S, T)Z^T$ and the user's selection of eigenvalues, the `StarNEig` interface functions

- `starneig_GEP_SM_Eigenvectors()` and
- `starneig_GEP_DM_Eigenvectors()`

compute and return a generalized eigenvector for each of the selected generalized eigenvalues. The shared memory variant `starneig_GEP_SM_Eigenvectors` is implemented on top of `StarPU`. The distributed memory variant `starneig_GEP_DM_Eigenvectors` still requires certain components to be integrated with the rest of the library.

Table 11 shows the run-time and residuals for `starneig_GEP_Eigenvectors` when the user has selected either 35% or 100% of all eigenvalues. The parallel scalability is illustrated in Figure 17 (35% selected) and Figure 18 (100% selected). We note that the relative residuals are always a modest multiple of the unit roundoff. Moreover, the code scales reasonably up to 32 cores.

In exact arithmetic it is straightforward to compute eigenvectors in both the standard and the generalized case. Standard algorithms are variants of the forward/backward substitution algorithms used for Gaussian elimination. However, in floating point arithmetic it is entirely possible that the intermediate or the final result exceeds the representational range and thus no regular solver can succeed. The `StarNeig` solvers are based on the principles developed in [11] and [12] for regular backward substitution. The solvers dynamically scale the right hand side and the solution to prevent overflow. The scalings are accumulated within each tile and are only applied when the results from two tiles are combined. In reality, this is equivalent to different processing units using different units of measurement, say, SI units versus imperial units. `StarNeig` solvers examine every division and every linear update. If overflow is possible, then a suitable scaling factor is computed and applied. If no scaling is necessary, then the `StarNeig` solvers will run



(a) Selected eigenvectors in grey (b) The right hand side (c) After backtransformation

Figure 14: On the left: The users selection of eigenvalues both real and complex. Center: the compressed representation of the eigenvectors used internally. Right: the final result returned to the user after backtransformation.

slightly slower than a non-robust solver since the unnecessary checks must still be completed. If scaling is necessary, then the `StarNeig` solvers will produce a result which can be evaluated whereas the non-robust solver will fail due to floating point overflow. Applying the necessary scalings requires extra time. This is clearly demonstrated in Figure 18. Here, the cases of $n = 60\,000$ and $n = 80\,000$ required more time than suggested by the case of $n = 40\,000$ due to the presence of many (nearly) infinite and nearly indefinite eigenvalues (see Table 7) that dramatically increased the need for numerical scaling.

Table 10: Run-times (64 cores) and residuals for `starneig_SEP_SM_Eigenvectors`. Residuals are reported in multiples of the double precision floating-point unit roundoff; see (9).

n	Selected	Run-time ($secs$)	$E_{sep}(\lambda, x)$ mean	$E_{sep}(\lambda, x)$ min	$E_{sep}(\lambda, x)$ max
10 000	35 %	1	1.03	0.72	6.28
20 000	35 %	8	0.96	0.67	1.51
40 000	35 %	40	0.45	0.29	7.24
60 000	35 %	117	0.81	0.54	5.73
80 000	35 %	256	0.69	0.43	6.59
100 000	35 %	473	0.24	0.16	0.77
120 000	35 %	781	0.47	0.22	4.13
10 000	100 %	4	1.04	0.71	6.28
20 000	100 %	14	0.96	0.67	1.91
40 000	100 %	92	0.45	0.29	80.99
60 000	100 %	301	0.81	0.54	16.49
80 000	100 %	654	0.69	0.43	12.13
100 000	100 %	1210	0.24	0.16	0.85
120 000	100 %	1995	0.47	0.22	16.53

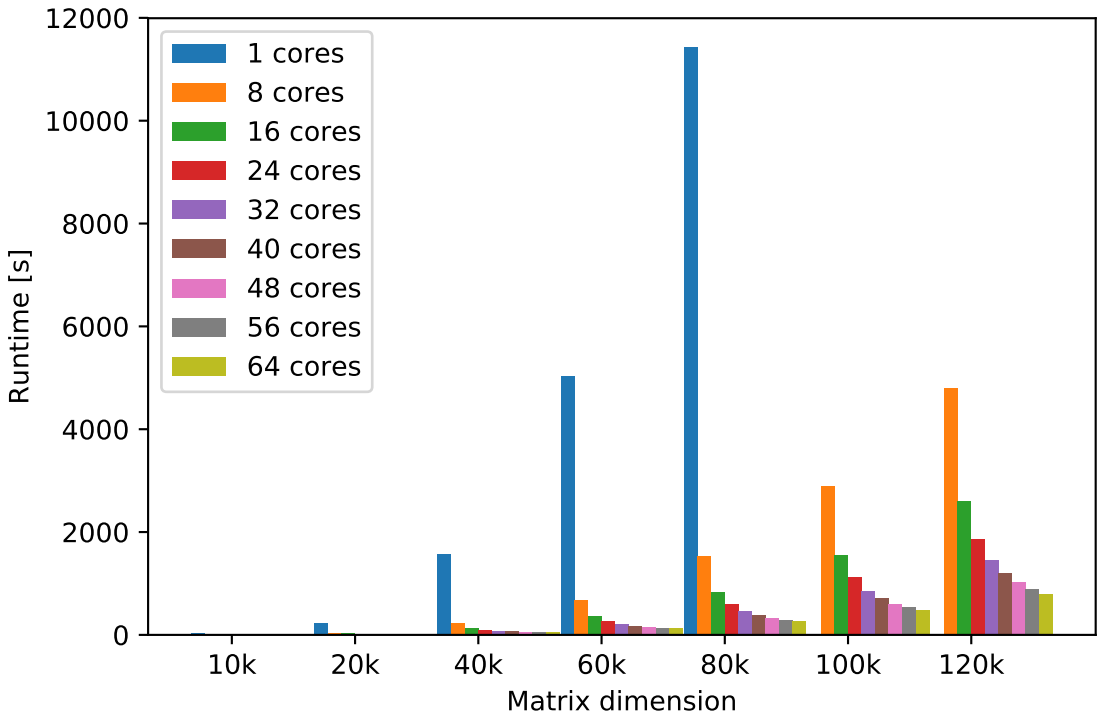


Figure 15: Scalability results for `starneig_SEP_SM_Eigenvectors`. 35% of the eigenvalues were randomly selected.

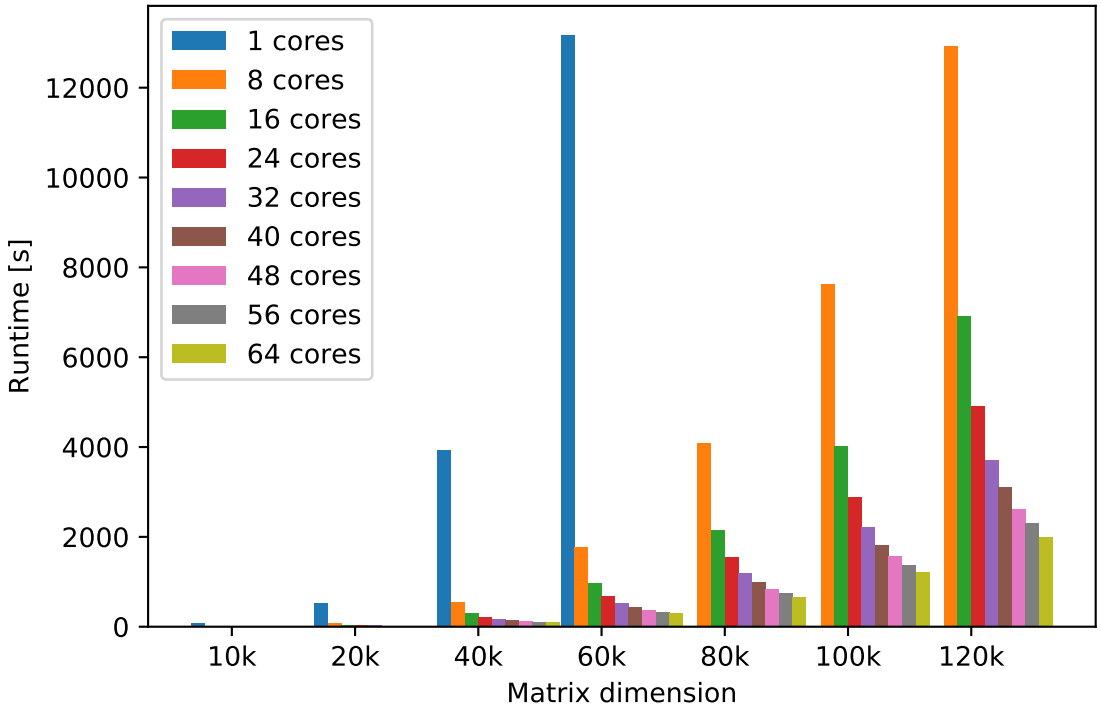


Figure 16: Scalability results for `starneig_SEP_SM_Eigenvectors`. 100% of the eigenvalues were selected.

Table 11: Run-times (64 cores) and residuals for `starneig_GEP_SM_Eigenvectors`. Residuals are reported in multiples of the double precision floating-point unit roundoff; see (10).

n	Selected	Run-time ($secs$)	$E_{gep}(\lambda, x)$ mean	$E_{gep}(\lambda, x)$ min	$E_{gep}(\lambda, x)$ max
10 000	35 %	3	0.40	0.15	0.62
20 000	35 %	16	0.41	0.14	0.81
40 000	35 %	118	0.37	0.14	0.92
60 000	35 %	1215	0.13	0.05	0.23
80 000	35 %	983	0.09	0.07	0.17
10 000	100 %	5	0.40	0.14	0.62
20 000	100 %	35	0.41	0.14	1.02
40 000	100 %	266	0.37	0.13	0.92
60 000	100 %	3133	0.13	0.05	0.23
80 000	100 %	2147	0.09	0.07	0.17

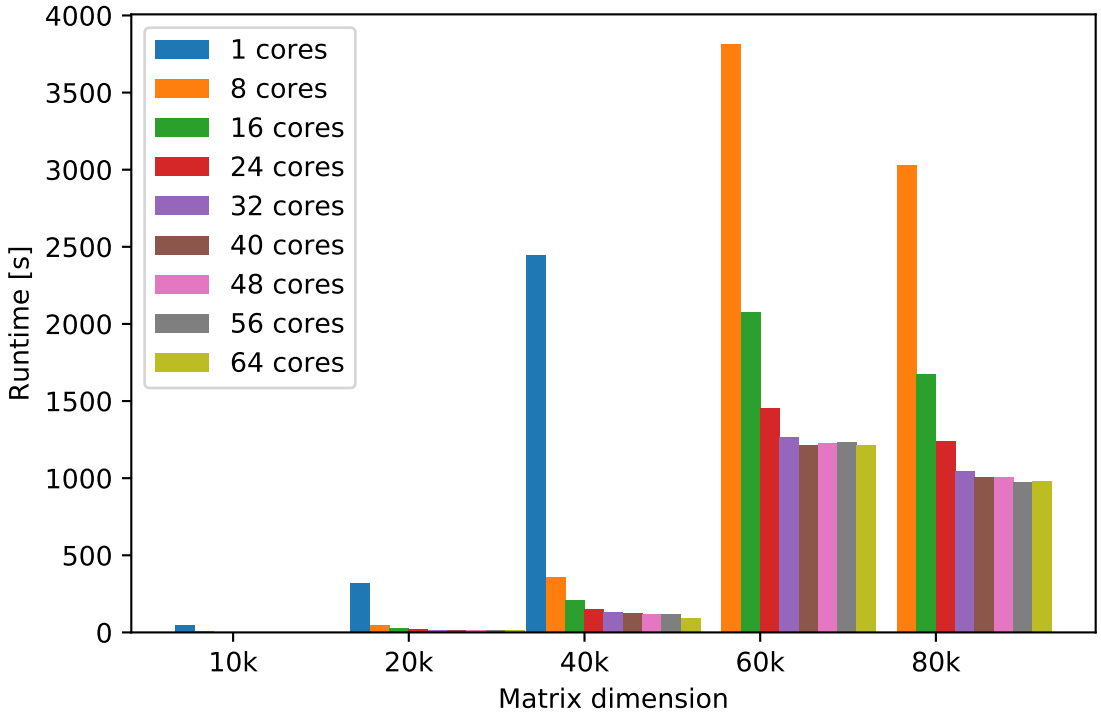


Figure 17: Scalability results for `starneig_GEP_SM_Eigenvectors`. 35% of the eigenvalues were randomly selected.

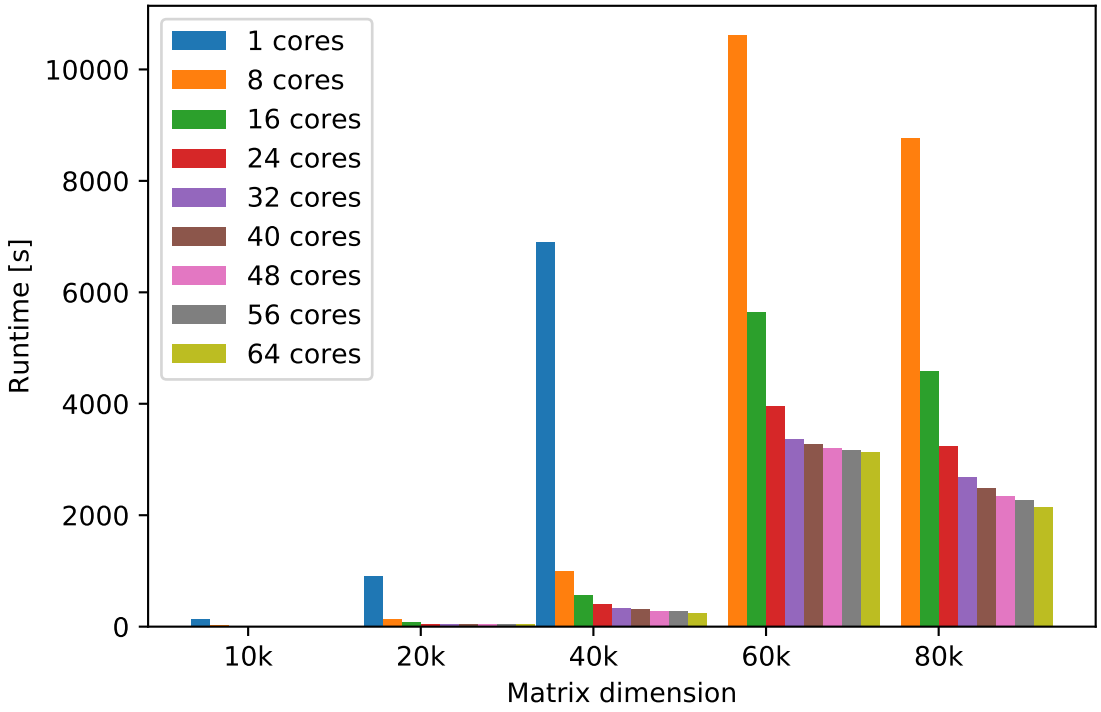


Figure 18: Scalability results for `starneig_GEP_SM_Eigenvectors`. 100% of the eigenvalues were selected.

8 Summary and some conclusions

Our long term goal has been to develop and implement a full suite of task-based algorithms for solving nonsymmetric (dense) eigenvalues problems. Within the NLAFET project, we have created a library called `StarNEig`. It is currently in a beta state, but already implements task-based algorithms for most required computational steps. Our main goal in this deliverable was to demonstrate that `StarNEig` library can perform the following steps correctly and efficiently:

Step 1 Reduction to condensed forms (Hessenberg or Hessenberg-triangular forms)

Step 2 Computation of the eigenvalues, i.e., reduction to (generalized) Schur forms

Step 3 Eigenvalue reordering

Step 4 Computation of eigenvectors

Our approach for accomplishing this was to construct an experiment where we pass a set of matrices and matrix pairs through all four steps. The relative residuals computed after each step indicate that the library can indeed perform all four steps correctly.

In most cases the task-based components of `StarNEig` are several times faster than `ScaLAPACK`-style subroutines. This is most clearly demonstrated in the case of the standard eigenvalue reordering problem (Step 3). Here we consistently obtain a speedups of 2.8 or higher. For the generalized eigenvalue problem, all `StarNEig` codes are faster than `ScaLAPACK`-style subroutines, but the improvement is less predictable. This may well be due to the fact that the largest test problems were very ill-conditioned. Furthermore, the `StarNEig` library produces results for which relative residuals are comparable to the considered `ScaLAPACK`-style subroutines. The presented results from the conducted scalability experiments show that the algorithm implementations are scalable to some extent in distributed memory. To conclude, our task-based algorithms and implementations contribute novel prototypes for solving dense nonsymmetric eigenvalue problems for future extreme scale (likely heterogeneous) systems. Future work includes the use of `StarNEig` software with Krylov-based methods for solving large-scale standard and generalized eigenvalue problems and provide functionalities also for complex datatypes.

Acknowledgements

We thank the High Performance Computing Center North (HPC2N) at Umeå University, which is part of the Swedish National Infrastructure for Computing (SNIC), for providing computational resources and valuable support during test and performance runs.

References

[1] StarPU — A Unified Runtime System for Heterogeneous Multicore Architectures. `http://starpu.gforge.inria.fr/`.

[2] B. Adlerborn, B. Kågström, and D. Kressner. PDHGEQZ User Guide. *NLAFET Working Note* WN-2, May, 2016. Also as Report UMINF 15.12, Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden.

[3] B. Adlerborn, B. Kågström, and D. Kressner. Parallel variants of the multishift QZ algorithm with advanced deflation techniques. In B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, editors, *Applied Parallel Computing, PARA 2006*, LNCS 4699, pages 117–126. Springer Berlin Heidelberg, 2006.

[4] B. Adlerborn, B. Kågström, and D. Kressner. A Parallel QZ Algorithm for distributed memory HPC-systems. *SIAM J. Sci. Comput.*, 36(5):C480–C503, 2014.

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

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

[7] R. Granat, B. Kågström, and D. Kressner. Parallel eigenvalue reordering in real Schur forms. *Concurrency and Computation: Practice and Experience*, 21(9):1225–1250, 2009.

[8] R. Granat, B. Kågström, D. Kressner, and M. Shao. ALGORITHM 953: Parallel Library Software for the Multishift QR Algorithm with Aggressive Early Deflation. *ACM Trans. Math. Software*, 41(4):Article 29:1–23, 2015.

[9] R. Granat, B. Kågström, D. Kressner, and M. Shao. ALGORITHM 953: Parallel Library Software for the Multishift QR Algorithm with Aggressive Early Deflation — Electronic Appendix: Derivation of the Performance Model. *ACM Trans. Math. Software*, 41(4), 2015. (Available online DOI `http://dx.doi.org/10.1145/2699471`).

[10] B. Kågström and D. Kressner. Multishift variants of the QZ algorithm with aggressive early deflation. *SIAM J. Matrix Anal. Appl.*, 29(1):199–227, 2006.

[11] C. C. Kjelgaard Mikkelsen and L. Karlsson. Blocked algorithms for robust solution of triangular linear systems. In Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 68–78, Cham, 2018. Springer International Publishing.

[12] C. C. Kjelgaard Mikkelsen, A. B. Schwarz, and L. Karlsson. Parallel robust solution of triangular linear systems. *Concurrency and Computation: Practice and Experience*, 0(0):1–19, 2018.

[13] M. Myllykoski. A Task-Based Algorithm for Reordering the Eigenvalues of a Matrix in Real Schur Form. In R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, editors, *Parallel Processing and Applied Mathematics*, volume 10777, pages 207–216, Cham, 2018. Springer International Publishing.

[14] M. Myllykoski, C. C. Kjelgaard Mikkelsen, L. Karlsson, and B. Kågström. Task-Based Parallel Algorithms for Reordering of Matrices in Real Schur Form. *NLAFET Working Note* WN-11, April, 2017. Also as Report UMINF 17.11, Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden.