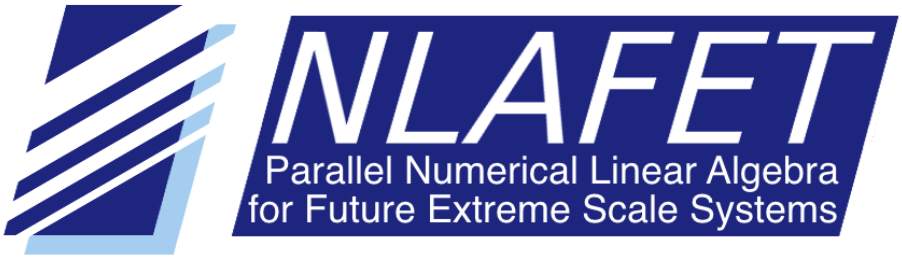


H2020–FETHPC–2014: GA 671633

D2.5

Eigenvalue problem solvers

April 2017

Document information

Scheduled delivery	2017-04-30
Actual delivery	2017-04-27
Version	1.0
Responsible partner	UMU

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2017-03-28	CCKM	Draft	0.1	Not applicable
2017-04-26	CCKM	Final	1.0	Typographical errors fixed, figure placement improved and a few new paragraphs added in response to review questions.

Author(s)

Carl Christian Kjelgaard Mikkelsen (UMU)
Mirko Myllykoski (UMU)
Björn Adlerborn (UMU)
Lars Karlsson (UMU)
Bo Kågström (UMU)

Internal reviewers

Alan Ayala (INRIA)
Negin Bagherpour (UNIMAN)
Sven Hammarling (UNIMAN)
Mawussi Zounon (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

1 Introduction

The *Description of Action* document states for deliverable D2.5:

> "*D2.5 Eigenvalue problem solvers*
>
> Report on computation of eigenvectors and reordering of eigenvalues in Schur and generalized Schur forms. Includes evaluation of the scalability and tunability of the prototype software developed."

This deliverable is in the context of Task 2.3 (Eigenvalue problem solvers).

1.1 Long term goals

The long term goal of the team at UMU is to develop and implement a full suite of task based algorithms for the standard and generalized eigenvalue problems. For the purpose of this deliverable, we have concentrated on the problem of reordering eigenvalues and computing eigenvectors in parallel.

1.2 Eigenvalue problems

Given an n by n matrix A, the standard eigenvalue problem for A consists of finding scalars (eigenvalues) λ_i and vectors $v_i \neq 0$ (eigenvectors), such that

$$Av_i = \lambda_i v_i. \tag{1}$$

Given two n by n matrices A and B the generalized eigenvalue problem for the matrix pencil $A - \lambda B$ consists of finding generalized eigenvalues λ_i and generalized eigenvectors v_i such that

$$Av_i = \lambda_i B v_i. \tag{2}$$

Dense eigenvalue problems are solved by reducing the matrices to standard form. A square complex matrix A has a Schur decomposition

$$A = QSQ^H \tag{3}$$

where Q is a unitary matrix and S is an upper triangular matrix. Similarly, if A and B are complex matrices, then there exist a generalized Schur decomposition

$$A = QSZ^H, \quad B = QTZ^H, \tag{4}$$

where Q and Z are unitary matrices, and S and T are upper triangular. The standard eigenvalues of A are the diagonal elements of S. The generalized eigenvalues of the pencil $A - \lambda B$ can be determined from the diagonal elements of the matrices S and T. In both cases, the eigenvectors can be computed by backward substitution and back-transformation to the original basis. Unitary transformations preserve the Euclidean norm (two-norm) of vectors and matrices and are numerically stable.

A square real matrix A has a real Schur decomposition

$$A = QSQ^T \tag{5}$$

where S is quasi upper triangular with 1 by 1 or 2 by 2 blocks on the diagonal and Q is orthonormal. Each 2 by 2 block on the diagonal of S corresponds to a pair of complex

conjugate eigenvalues of A. Similarly, if A and B are real matrices, then the pencil $A-\lambda B$ has a real Schur decomposition

$$A = QSZ^T, \quad B = QTZ^T \tag{6}$$

where Q and Z are orthonormal matrices, S is quasi upper triangular and T is triangular.

1.3 Practical considerations

In practice, the reduction from the general form to the relevant Schur form is carried out gradually using *two-sided* transformation algorithms. Here we sketch the procedure for the standard eigenvalue problem. The matrix A is first transformed to upper Hessenberg form

$$H = Q^H AQ. \tag{7}$$

This is a direct method which requires a finite number of operations. We then reduce H iteratively to upper triangular form S. The flow of the algorithm depends on the actual floating point values. The eigenvalue reordering problem for S now consists of moving the user's selection of eigenvalues to the upper left corner of S. This problem is also solved using a sequence of two-sided unitary transformations. They induce some very complex data dependencies which are common to all two-sided transformation algorithms.

1.4 Our current focus

Currently, there exist in ScaLAPACK parallel algorithms for many of the two-sided reductions needed to reduce standard eigenvalue problems to Schur form [6]. These algorithms are all based on message passing using MPI. There are algorithms for computing eigenvectors in ScaLAPACK, but with one very important limitation. During the backward substitution phase there is either no protection against overflow or the code is essentially sequential. Normally, we can solve triangular linear systems very accurately, but in the context of eigenvector computation, the solver is likely to overflow if the eigenvalue is part of a cluster of nearby eigenvalues. Eigensolvers fight overflow by scaling the solution dynamically. Until now, this has proven difficult to do efficiently in parallel.

In view of the current state of ScaLAPACK, we have concentrated on two topics:

1. Solving the reordering problem for the standard eigenvalue problem for matrices in real Schur form.

2. Computing (right) eigenvectors for the standard eigenvalue problem for matrices in complex Schur form.

We will apply the lessons learned from these special cases to the problem of developing other task based algorithms for both standard and generalized eigenvalue problems.

2 Eigenvalue reordering

Given a real Schur decomposition $A = QSQ^T$, the eigenvalue reordering problem for S consists of modifying the Schur basis Q such that selected eigenvalues of A appear in the upper left corner of the modified matrix S.

In this section we briefly review the previous work which is directly relevant for our current work. We begin by discussing the kernels which form the basis for most known algorithms. Moreover, we discuss the sequential algorithm which is implemented in LAPACK as DTRSEN, Kressner's blocked improvement BDTRSEN as well as the parallel algorithm implemented in ScaLAPACK as PBDTRSEN. We refer to the cited references and the references therein for a more thorough survey of the field.

2.1 The fundamental swapping kernels

The fundamental swapping kernels by Bai and Demmel (1993) [3] form the basis for most known algorithms for the reordering problem. Consider a matrix S in real Schur form which has only two diagonal blocks, i.e.

$$S = \begin{bmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{bmatrix},$$

where S_{11} and S_{22} have size at most 2×2. Then the swapping kernels can be used to compute an orthonormal matrix V such that

$$V^T S V = \begin{bmatrix} \tilde{S}_{11} & \tilde{S}_{12} \\ 0 & \tilde{S}_{22} \end{bmatrix},$$

where $\tilde{S}_{11}$ and $\tilde{S}_{22}$ have the same eigenvalues as S_{22} and S_{11}. We say that the blocks/eigenvalues have been swapped.

The swapping kernels are based on the robust solution of a tiny Sylvester equation by solving the equivalent linear system using Gaussian elimination with complete pivoting and scaling to avoid overflow. Backward stability is guaranteed by cheaply monitoring the perturbations introduced by each swap and rejecting those swaps that lead to unacceptably large errors. The authors were unable to find or construct an example were the method failed, indicating that even though failures are a theoretical possibility, they are extremely rare in practice.

Kågström and Poromaa (1996) [7] have extended the work by Bai and Demmel to the generalized eigenvalue reordering problem. Their algorithm is based on the robust solution of a tiny generalized Sylvester equation and similarly guarantees backward stability.

2.2 The sequential algorithm implemented in DTRSEN

The sequential algorithm implemented in LAPACK as DTRSEN is based on the swapping kernels by Bai and Demmel (1993) [3]. The user's selection of eigenvalues is systematically reordered to the top left corner of the matrix by repeatedly swapping *adjacent* blocks. The orthogonal transformation constructed from a small submatrix enclosing a pair of adjacent blocks affect all entries above and to the right of the submatrix. In principle, it is possible to swap any pair of blocks, but unless the blocks are adjacent, the real Schur form is destroyed by fill-in. One can readily solve the reordering problem by scanning the diagonal of S from the upper left to the lower right corner, moving any selected blocks to the top by a sequence of swaps. This is the approach taken in DTRSEN. The computation is memory bound because it contains many low-level BLAS operations.

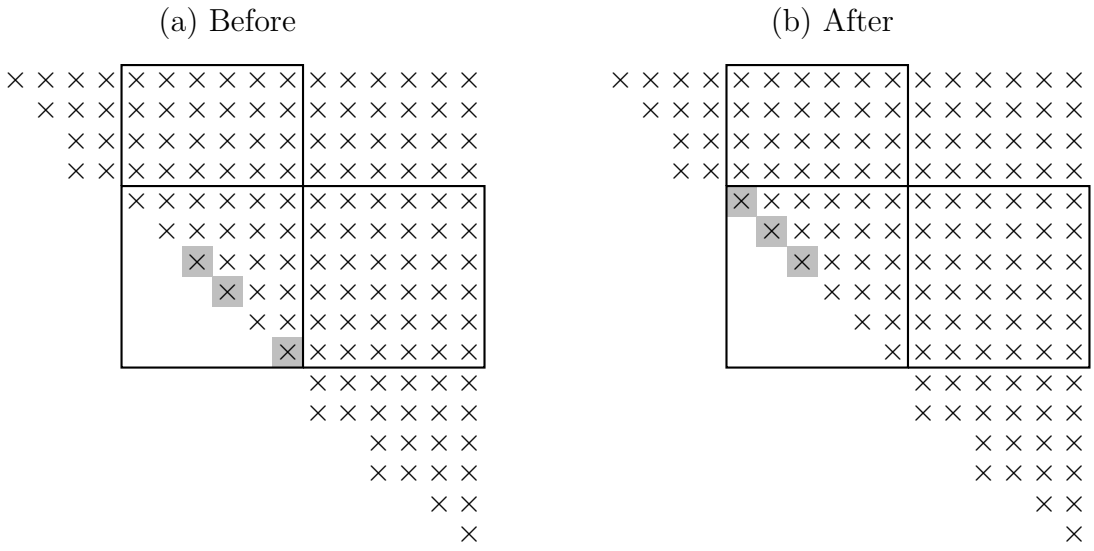


Figure 1: Illustration of a quasi-triangular matrix of dimension $n = 16$ with three selected 1×1 blocks within a window (submatrix on the diagonal) of dimension $w = 6$. The initial and final placements of the selected blocks are shown in (a) and (b), respectively. The transformation of the big matrix proceeds in three logical steps. The blocks are reordered by a sequence of swaps applied only within the window. The resulting transformations are then applied using level 3 BLAS to the indicated off-diagonal submatrices above and to the right of the window.

2.3 The blocked algorithm implemented in BDTRSEN

Kressner (2006) [8] improves the execution rate of the reordering algorithms in both [3] and [7] by reorganizing the computations for improved cache reuse. Specifically, a group of eigenvalues is reordered within a small window on the diagonal. Most of the off-diagonal updates are delayed and applied efficiently either in factored form or using the Level 3 BLAS matrix multiply and add routine DGEMM after explicitly accumulating the transformations. This is the approach take in BDTRSEN. The idea is best illustrated by a small example; see Figure 1. Three selected blocks (which all happen to be of size 1×1) are highlighted. A window of size $w = 6$ is placed on the diagonal such that the selected block furthest down the diagonal is located flush against the bottom right corner of the window. The selected blocks are reordered by a sequence of swaps (in this case, $2+2+3 = 7$ swaps) such that they end up as illustrated in Figure 1(b). The key is to only apply the transformations to the window at this stage. The window computation consists of an interleaved mix of updates from both the left and the right and is hard to implement efficiently. After processing the window, the resulting transformations are accumulated into a small orthogonal matrix of size $w \times w$. This new transformation is applied from the left to the block row to the right of the window and from the right to the block column above the window. These two operations are done efficiently using DGEMM operations.

2.4 The parallel algorithm implemented in PBDTRSEN

Granat, Kågström, and Kressner (2009) [5] present a parallel algorithm for both the standard and generalized reordering problems. The algorithms are based on those in [8] and the software is expressed in the ScaLAPACK style, i.e. the processes are arranged in a

mesh and the matrices are distributed in a two-dimensional block-cyclic fashion. Several sliding computational windows are introduced to increase the degree of concurrency. Global synchronization is needed to alternate between updates along block rows and along block columns. Eigenvalues are reordered across a process border by redundantly processing the window on the processors which own the two diagonal blocks.

3 Our progress on eigenvalue reordering

We have developed a task based parallel algorithm for reordering eigenvalues in real Schur forms. Our current implementation can be executed in parallel using StarPU[1]. Three factors contributed to our choice of run-time system. Firstly, StarPU is well documented and contains all the desirable features identified in Deliverable D6.1. Secondly, initial progression to distributed memory machines is straightforward as StarPU is able to deduce all necessary node-to-node communications from the task graph once the data distribution is provided. A high performance distributed memory implementation is likely to require additional work, but in principle, the same StarPU code can be executed in both shared and distributed memory machines. And finally, since the diagonal windows necessary overlap and a single sliding window is not usually enough to reorder the whole matrix, the resulting data dependencies become extremely complex. This effectively excludes less powerful run-time systems, such as OpenMP, from the list of run-time systems which can be used.

Parallelism is achieved by using multiple sliding windows in a manner similar to PBDTRSEN. Each sliding window manifests itself as a semi-independent entry point into the task graph. Different sliding windows are not necessarily independent as one may block another. The main advantage of the task based approach is that we have eliminated the need for global synchronization associated with the repeated updates along the rows and columns. The technical details are complicated and are discussed in the NLAFET Working Note 11 [10].

Our current implementation has been very successful on a shared memory machine, see Section 4. It is faster than PBDTRSEN by a factor of at least 3, but more importantly it makes excellent use of the available resources. The weak and strong scaling efficiencies are well above 50% for all but the smallest test problems. It routinely exceeds 50% of the peak flop rate. With the exception of the smallest problems tested there is hardly any idle time and the parallel overhead is tiny.

We are currently rewriting and extending our code to run efficiently on a distributed memory machine using StarPU. In addition, work to extend the algorithms to generalized Schur forms is underway.

Our current implementation is available for download from the NLAFET repository `Eigen-reorder`. The code can automatically generate test examples and verify the results. For more details see Deliverable D7.5.

4 The performance of reordering algorithms

In this section, we present results pertaining to reordering of eigenvalues for matrices in real Schur form, on a shared memory machine using StarPU, double precision arithmetic and accumulation of the transformations needed to process the windows. Our software

[1]The StarPU project is hosted at this address `http://starpu.gforge.inria.fr/`

is capable of assigning different priorities for different tasks. Thus, we decided to use the `prio` scheduler (a central task queue based scheduler that sorts tasks by priority specified by the programmer) in our experiments.

4.1 Computer system

All experiments were executed on the Kebnekaise system at HPC2N, Umeå University. Each compute node contains 28 Intel Xeon E5-2690v4 cores organized in 2 NUMA island with 14 cores each (Intel Broadwell). The nodes are connected with an FDR Infiniband Network. Each compute core has 32 KB L1 data cache, 32 KB L1 instruction cache and 256 KB L2 cache. Moreover, for every NUMA island there is 35 MB of shared L3 cache. There is 128 GB of RAM per compute node. A schematic representation of a compute node is shown in Figure 2.

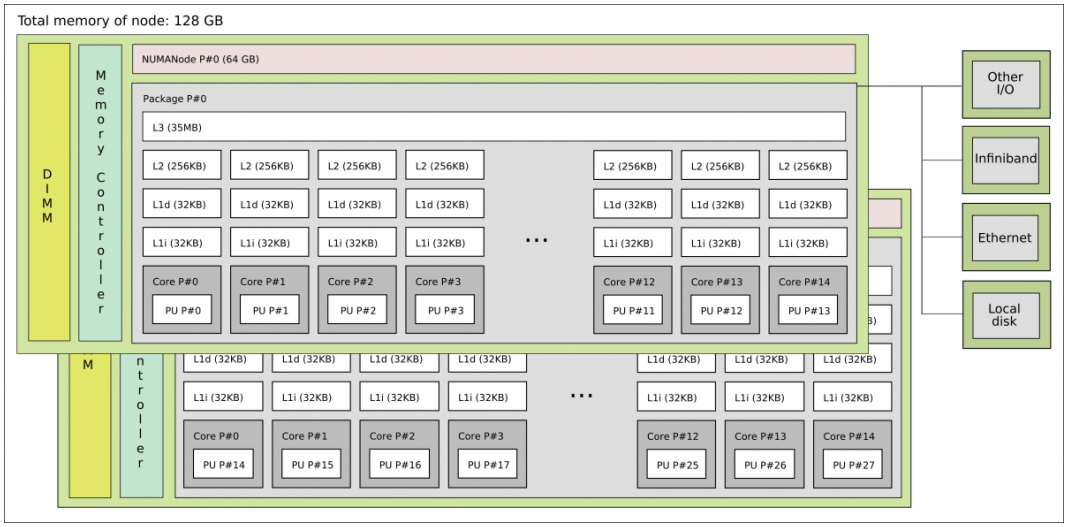


Figure 2: Schematic representation of a compute node of the Kebnekaise system at HPC2N, Umeå University.

4.2 Test matrices

We have developed a generator which can build a complete experiment from a single random seed and a small set of parameters (n, k, q). Here n is the dimension of the matrix A, k is the number of pairs of complex conjugate eigenvalues and q is the probability that the user selects any given diagonal block.

4.3 Experimental methodology

All timings were done using the `clock_gettime` function. Each experiment was repeated several times: once to validate the output and three times to measure the runtime. The runtimes were saved and the median was computed. Problems were generated from a random seed using the parameters listed here:

1. Matrix dimension $n \in \{10000, 20000, 30000, 40000\}$.

2. Number of pairs of complex conjugate eigenvalues $k = n/4$, that is, half of the eigenvalues belong to complex conjugate pairs.

3. The probability q of the user choosing a specific diagonal block,

$$q \in \{0.05, 0.015, 0.35, 0.50\}.$$

Parallel experiments with p cores were always executed using cores 0 though $p-1$, see Figure 2.

4.4 Accuracy

Results are worthless if they are not accurate. With respect to all experiments reported the following statements are true.

1. For each reordered eigenvalue $\hat{\lambda}$, we have the relative error bound

$$\frac{|\lambda - \hat{\lambda}|}{|\lambda|} \lesssim 900u, \tag{8}$$

where λ is the original value of the eigenvalue.

2. For each reordered Schur decomposition $\hat{A} = \hat{Q}\hat{S}\hat{Q}^T$ we have a relative backward error bound

$$\frac{\|A - \hat{A}\|_F}{\|A\|_F} \lesssim 190u, \tag{9}$$

where $A = QSQ^T$ is the original matrix.

3. For each new Schur basis $\hat{Q}$, we have

$$\frac{\|\hat{Q}^T\hat{Q} - I\|_F}{\|I\|_F} \lesssim 315u, \tag{10}$$

which shows that the orthogonality is very nearly preserved.

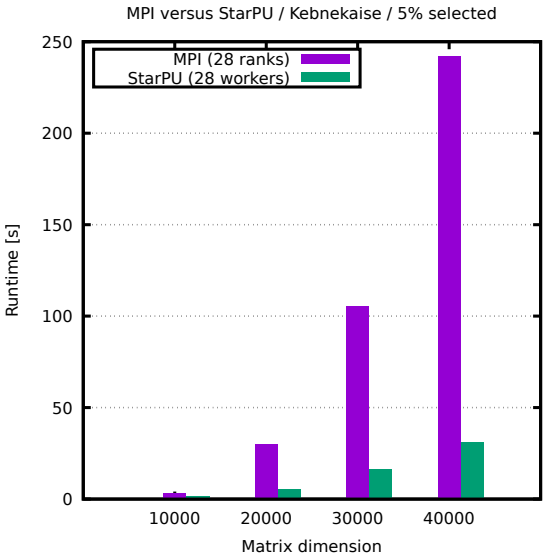
In exact arithmetic, the errors should be zero. The small values obtained merely serve to illustrate that our StarPU implementation does not commit any obvious errors for the well conditioned test problems considered.

4.5 Time to solve

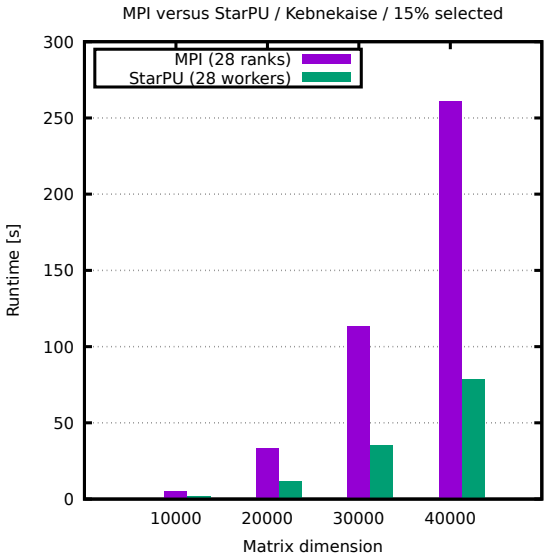
We report on the speed of our StarPU implementation relative to the existing ScaLAPACK routine PBDTRSEN. We compare the case of 28 MPI ranks to the case of 28 StarPU workers, i.e. full utilization of a single node on Kebnekaise. The results are illustrated in Figure 3. It is clear that our StarPU implementation is significantly faster than PBDTRSEN. In particular, for the largest problem size, we record a speedup well above 3 regardless of the number of selected eigenvalues.

4.6 Sequential execution

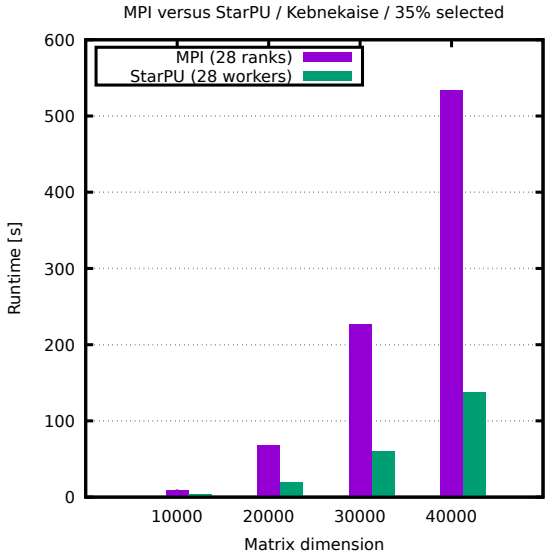
We report on the time T_s to solve problems using single threaded code and compare our code to the sequential blocked code BDTRSEN. This is important because it allows us to determine the best sequential code and establish a baseline against which all parallel speedups should be computed. Our results are shown in Figure 4. We note the paradoxical



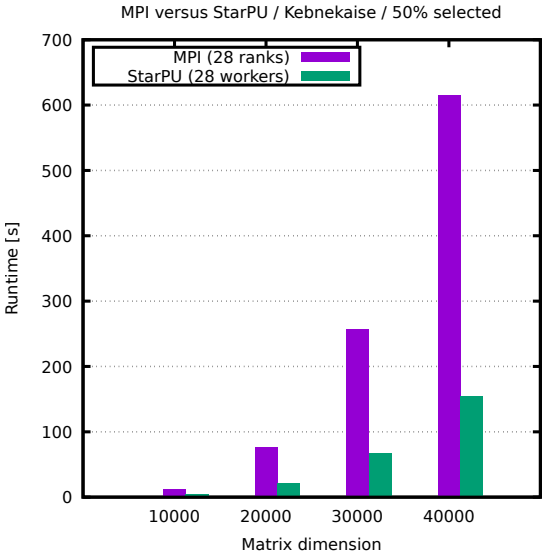
(a) 5% selected.



(b) 15% selected.



(c) 35% selected.

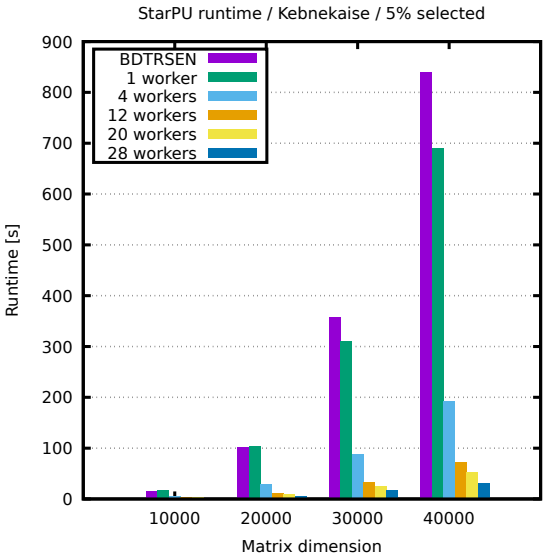


(d) 50% selected.

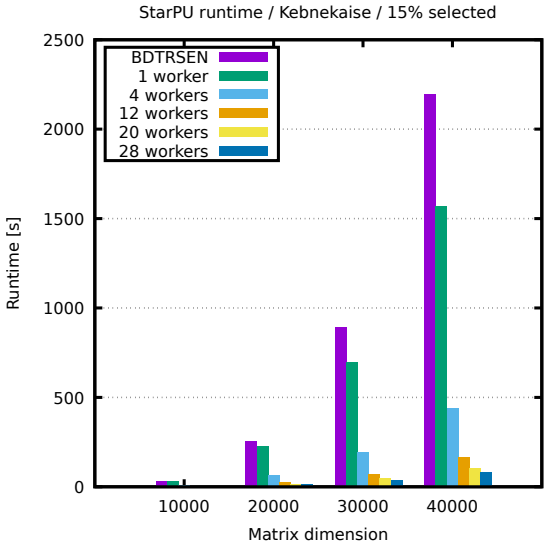
Figure 3: Comparison of the runtime for PBDTRSEN from ScaLAPACK and our new StarPU implementation. The number of selected eigenvalues relative to the matrix size runs through the values 5, 15, 35, and 50 percent. Observe that the scale on the y-axis varies between sub-figures.

result that our StarPU implementation which uses BDTRSEN as a kernel, is somewhat faster than BDTRSEN running alone.

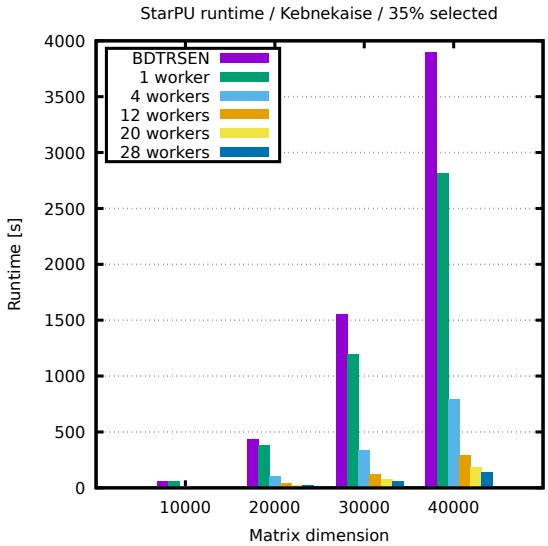
It remains an open problem to fully explain this situation, but the fact that the window size can chosen more freely in our implementations is probably a major factor. This allows us to have larger update task which implies fewer and larger DGEMM operations, hence a higher floprate.



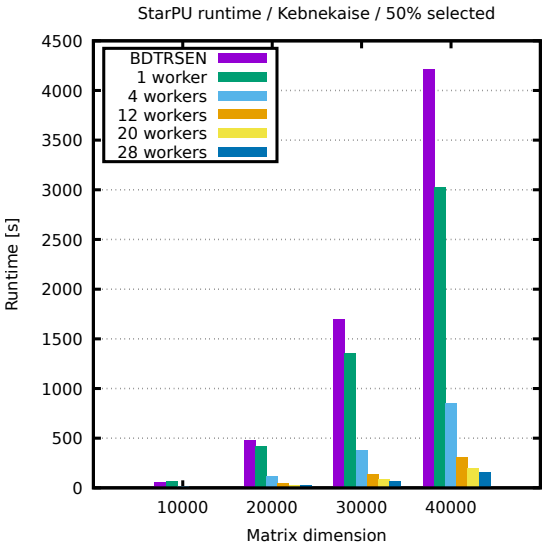
(a) 5% selected.



(b) 15% selected.



(c) 35% selected.



(d) 50% selected.

Figure 4: The runtime of the sequential code BDTRSEN and our StarPU implementation. The number of selected eigenvalues relative to the matrix size runs through the values 5, 15, 35, and 50 percent. Observe that the scale on the y-axis varies between sub-figures.

4.7 Scalability

The scalability of a program measures its response to an increase in the number of processing units. In the context of high performance computing, we are interested in weak and strong scalability. In both cases we report on the parallel efficiency ρ given by

$$\rho = \frac{T_s}{pT_p}, \tag{11}$$

where T_s is the serial execution time and T_p is the parallel execution time using p cores.

4.7.1 Weak scalability

Weak scalability refers to the situation where the problem size per processing unit is constant as the number of units is increased. Here we report on the weak scalability efficiency obtained by scaling our largest problem ($n = 40\,000$) from $p = 28$ down to $p = 1$ core. Our results are plotted in Figure 5. We are pleased to report that the efficiencies are well above 60%. Except for some minor bumps the curves decrease monotonically as expected.

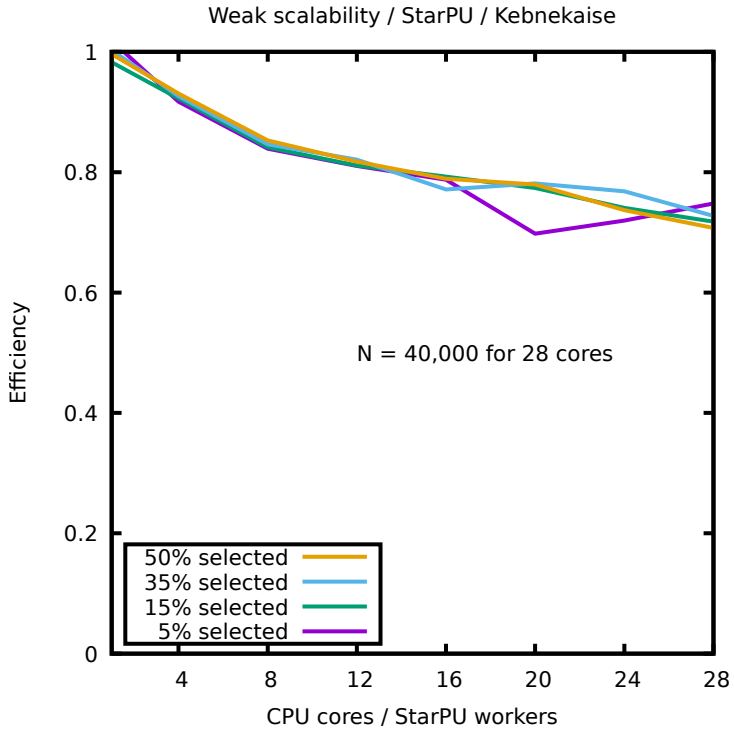
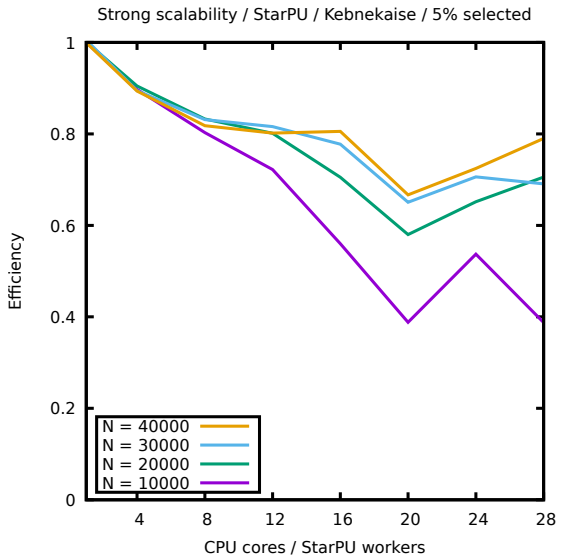


Figure 5: Weak scalability of our StarPU implementation.

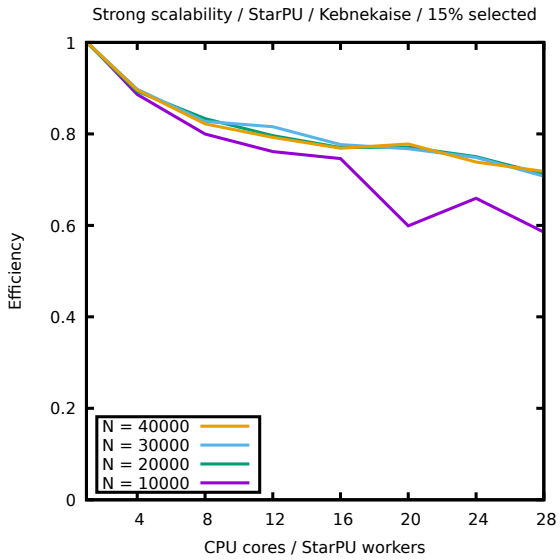
4.7.2 Strong scalability

Strong scalability refers to the situation where the problem size is constant as the number of processing units is increased. Here we report on strong scalability efficiency. Our results are shown in Figure 6. Ideally, one would like to obtain curves which are monotone and slowly decreasing. It is almost true for our experiments with the exception of the case of 5% and 15% percent selected eigenvalues. In general, shorter runs are more sensitive to

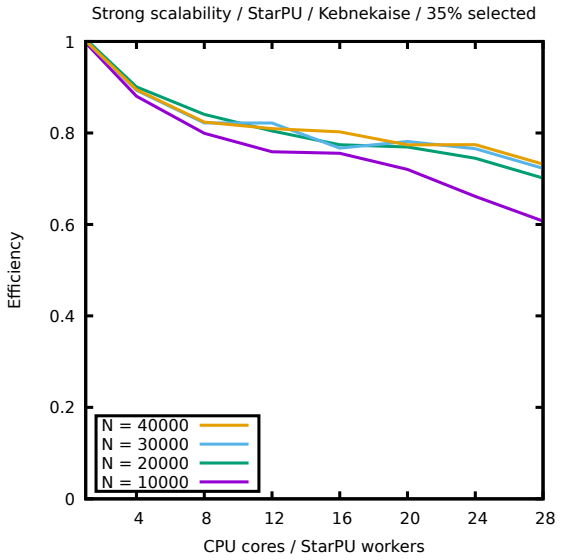
disruptions beyond our control, i.e. intervention by the operating system, so we are not surprised to record bumps when the computational load is rather light.



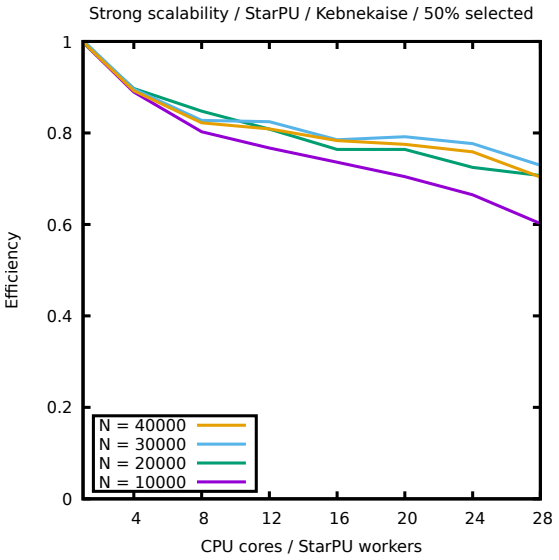
(a) 5% selected.



(b) 15% selected.



(c) 35% selected.



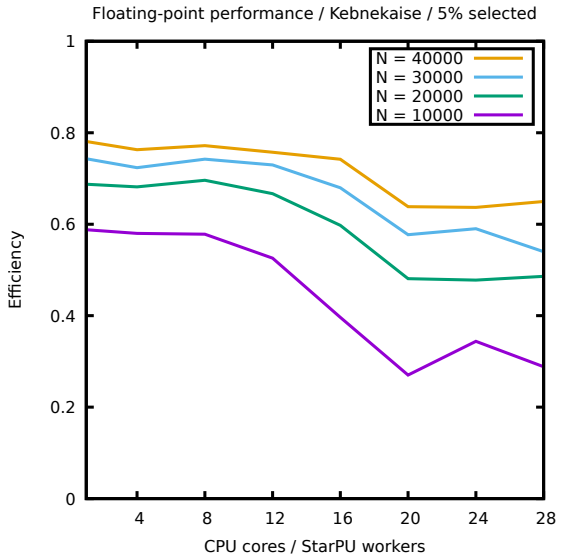
(d) 50% selected.

Figure 6: The strong scalability efficiency of our StarPU implementation. The number of selected eigenvalues relative to the matrix size runs through the values 5, 15, 35, and 50 percent.

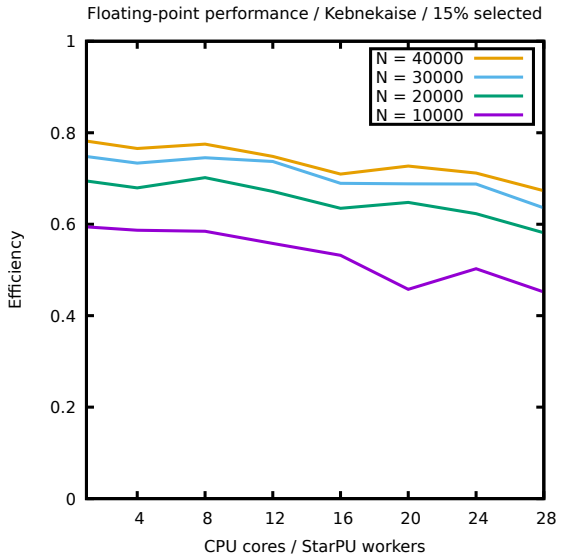
4.8 Flop rate

The flop rate is the number of floating point numbers per second. Our results are presented in Figure 7. Here we plot the relative flop rate, i.e. the flop rate relative to the peak flop rate. The curves represent *lower* bounds as we have only counted the flops performed during the row and column matrix–matrix multiply operations. The flops performed

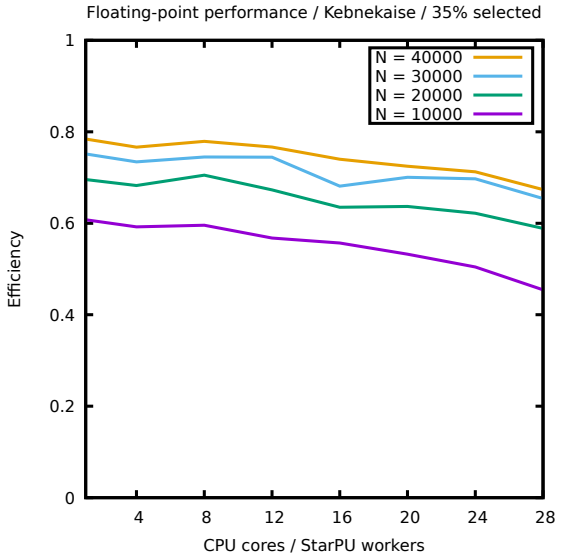
within each window are more difficult to count and have not been included. We are pleased to note that the lower bound for our relative flop rate is well above 50% for all but the lightest loads. Ideally, one would like to obtain curves which are decreasing monotonically. In practice, the occasional bumps are unavoidable and not necessarily reproducible from one experiment to the next.



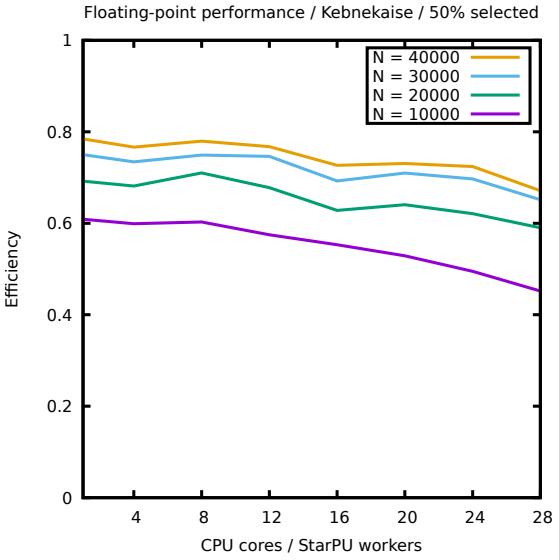
(a) 5% selected.



(b) 15% selected.



(c) 35% selected.



(d) 50% selected.

Figure 7: A lower bound for the flop rate of our StarPU implementation relative to the peak flop rate for the specific selection of cores. The number of selected eigenvalues relative to the matrix size runs through the values 5, 15, 35, and 50 percent.

4.9 Idle time and overhead

Worker threads executing code under a runtime system such as StarPU are either busy doing useful work, waiting for tasks to become available (idle time) or running the system

itself (parallel overhead). Ideally, we want all worker threads to advance the main calculation at all times, but this is of course not possible. StarPU has the ability to record the time spent in each activity. Here we report on the idle time and the overhead as a fraction of the total execution time. We include StarPU startup and shutdown times in the reported overhead. Our result are presented in Figure 8. In general, the impression is favorable and the workers are almost always moving the computation along, especially when the problem size is large and a large fraction of eigenvalues has been selected. If we were to omit the smallest problem size and the case of 5% selected eigenvalues, then all workers are devoting more than 95% of their time to advancing the calculations. There is significant amount of time lost to idling when only 5% are selected and the worker count is high. This is hardly surprising as we are trying to schedule a small load across a rather large number of workers.

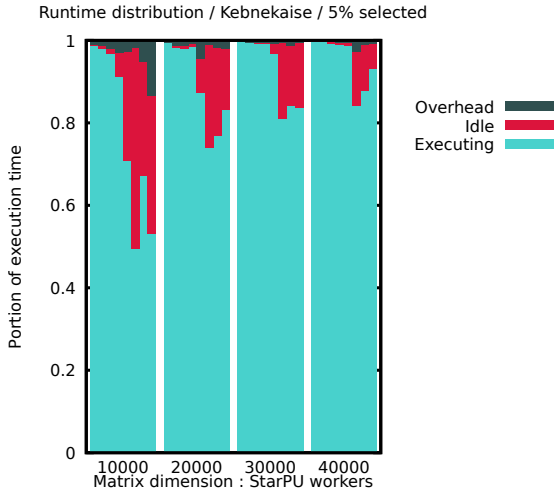
4.10 Tunability

The implementation has multiple tunable parameters. However, we have identified two parameters that have the largest impact on the performance. The matrix is initially partitioned into square blocks (tiles) of uniform size n_b. The block size has implications both for the data layout and the dependency tracking in StarPU. In addition, the block size also defines the task granularity. Based on our experiences, the block size appears to be the most important parameter. A second important parameter is the strategy that determines the order in which the tasks are inserted into StarPU as well as the task priorities, see [10]. Other parameters appear to be less important or a (close to) optimal values are already known. However, new tunable parameters are likely to appear as our implementation develops further.

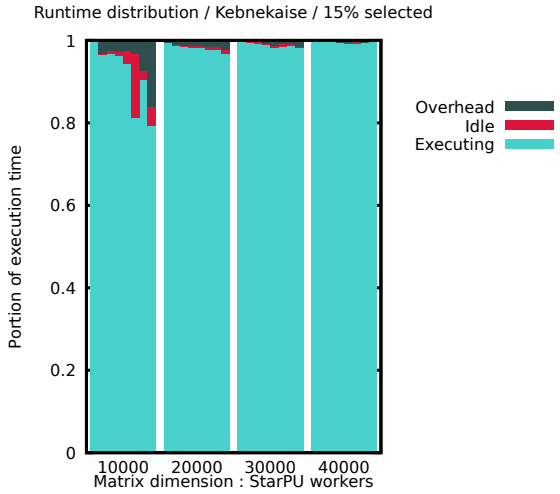
5 The next steps for eigenvalue reordering

We are currently rewriting and extending our code so that it can be executed efficiently by StarPU on a distributed memory machine. In addition, work to extend the algorithms to generalized Schur forms is underway.

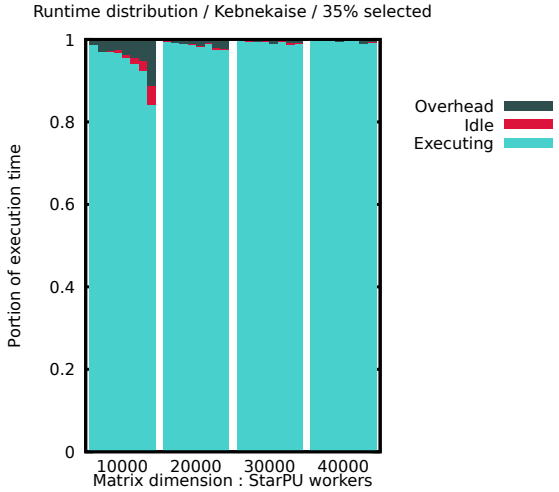
We fully expect to transfer the lessons learned from the problem of reordering eigenvalues to two-sided transformation algorithms needed for the reduction of dense standard and generalized eigenvalue problems to Schur and generalized Schur forms. This will include developing task-based algorithms for such reductions and implementing them for execution under StarPU.



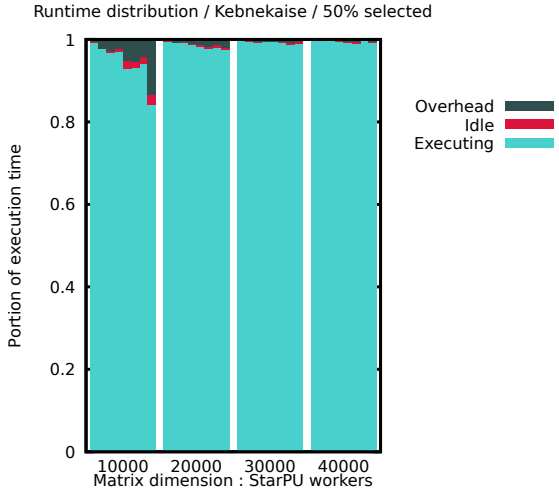
(a) 5% selected.



(b) 15% selected.



(c) 35% selected.



(d) 50% selected.

Figure 8: The idle time and overhead of our StarPU implementation relative to the total runtime. Each of the four sub-figures corresponds to the user selecting a specific fraction of the eigenvalues, specifically, 5, 15, 35, and 50 percent of the total. Within each figure, the results are grouped by matrix dimension. Within each group, the StarPU worker count runs through the numbers 1, 4, 12, 20 and 28 as we move from left to right.

6 Computation of eigenvectors

The problem of computing a single eigenvector reduces to the problem of solving a triangular linear system $(S - \lambda I)x = f$ and applying a unitary transformation $x \leftarrow Qx$. In practice there are at least two problems. Specifically,

1. The triangular solver may overflow.

2. Computing a single eigenvector is a memory bound operation.

The solver is likely to overflow if the eigenvalue is part of a cluster of nearby eigenvalues. In this case, the triangular matrix $S - \lambda I$ has a number of very small entries on the main diagonal and the corresponding divisions can trigger massive growth in the solution. For this reason, eigenvalue solvers attempt to compute a scaling α, such that the solution x of the *scaled* linear system

$$(S - \lambda I)x = \alpha b \tag{12}$$

does not overflow. This is done by protecting each scalar division and each linear update against overflow.

Solving a single triangular system of dimension n requires $O(n^2)$ flops on $O(n^2)$ words of memory. The arithmetic intensity (average number of flops per word) is $O(1)$. The problem of computing a single eigenvector is incompatible with modern computer architectures with deep memory hierarchies which favor algorithms with high arithmetic intensity. The eigenvector might be computed "rapidly", but the machine will be running at small fraction of its peak flop rate, wasting the majority of the processing power.

6.1 Robust scalar backward substitution

Since 1991 `xLATRS` [2] and its derivative `PxLATRS` [4] has fought overflow in triangular linear systems using an algorithm very similar to Algorithm 1 `RobustScalarBacksolve`. In fact, the only difference between `xLATRS` and `RobustScalarBacksolve` is that `xLATRS` contains an extra scaling which is harmless, but not necessary. Algorithm 1 addresses overflow by considering every single arithmetic operation, see Algorithm 2 `ProtectDivision` and Algorithm 3 `ProtectUpdate`. It is essentially impossible to parallelize Algorithm 1 efficiently. Recomputing the variable $x_{\max}$ at the end of every iteration requires global communication and badly affects the efficiency even if scalings are rarely needed.

7 Our progress on eigenvector computation

We have learned how to address overflow when solving triangular systems in parallel. The key is to do a blocked solve and use different scaling factors for each block row of the solution. We have replaced the standard kernels with robust variants. These new kernels operate on augmented vectors $\langle \alpha, x \rangle$ rather than regular vectors. An augmented vector consists of a scaling α and a vector x and represents the vector $\alpha^{-1}x$, which can lie far outside the representable range of floating point numbers. We explain this breakthrough in some detail, see Section 7.1 and refer to the NLAFET Working Notes 9 and 10 [9, 1] for a full discussion.

We have also learned to interleave the computation of several eigenvectors in order to improve the arithmetic intensity and reduce the solve time, see Section 7.2.

Algorithm 1: $(\alpha, x) = \texttt{RobustScalarBacksolve}(S, \lambda, b)$

Data: A non-singular upper triangular matrix $S - \lambda I \in \mathbb{C}^{n\times n}$ and $b \in \mathbb{C}^n$, such that

$$\|S - \lambda I\|_\infty \leq \Omega, \quad \|b\|_\infty \leq \Omega$$

and numbers $c(j)$ satisfying

$$\|S(1{:}j{-}1, j) - \lambda I\|_\infty \leq c(j), \quad j = 2, 3, \ldots, n.$$

Result: A scaling factor $\alpha \in (0, 1]$ and the solution of the scaled linear system

$$(S - \lambda I)x = \alpha b$$

where the scaling factor ensures that the computation of x cannot overflow.

```
1  α ← 1, x ← b, x_max ← ‖x‖_∞;
2  for j ← n, n − 1, ..., 1 do
3      β = ProtectDivision(x(j), λ, S(j, j));
4      if β ≠ 1 then
5          x ← βx;
6          α ← βα;
7      x(j) ← x(j) / (S(j,j) − λ);
8      if j > 1 then
9          β = ProtectUpdate(x_norm, c(j), |x(j)|);
10         if β ≠ 1 then
11             x ← βx;
12             α ← βα;
13         x(1:j−1) ← x(1:j−1) − S(1:j−1, j)x(j);
14         x_max ← ‖x(1:j−1)‖_∞;
15 return α, x;
```

Algorithm 2: $\alpha = \texttt{ProtectDivision}(b, s, \lambda)$

Data: Numbers b and t such that $|b| \leq \Omega$ and $s \neq \lambda$.

Result: A scaling factor $\alpha \in (0, 1]$ such that the result of the scaled division

$$x \leftarrow \frac{(\alpha b)}{s - \lambda} \tag{13}$$

cannot exceed the overflow threshold Ω.

```
α ← 1;
if |s − λ| < Ω^{-1} then
    if |b| > |s − λ|Ω then
        α ← |s−λ|Ω / |b|;
else
    if |s − λ| < 1 then
        if |b| > |s − λ|Ω then
            α ← |b|^{-1};
return α;
```

Algorithm 3: $\zeta = \texttt{ProtectUpdate}(y_{\text{norm}}, c_{\text{norm}}, x_{\text{norm}},)$

Data: Non-negative real numbers c_{norm}, x_{norm} and b_{norm} such that

$$\|Y\|_\infty \leq y_{\text{norm}} \leq \Omega, \quad \|C\|_\infty \leq c_{\text{norm}} \leq \Omega, \quad \|X\|_\infty \leq x_{\text{norm}} \leq \Omega.$$

Result: A scaling factor ζ such that

$$\zeta \left(y_{\text{norm}} + c_{\text{norm}} x_{\text{norm}} \right) \leq \Omega$$

which implies that the scaled linear update

$$Z \leftarrow \zeta Y - C(\zeta X) \tag{14}$$

cannot exceed the overflow threshold Ω.

```
ζ ← 1;
if x_norm ≤ 1 then
    if c_norm x_norm > Ω − b_norm then
        ζ ← 1/2;
else
    if c_norm > (Ω − b_norm)/x_norm then
        ζ ← 1/(2 x_norm);
return ζ;
```

We have developed a robust parallel algorithm for computing right eigenvectors for the standard eigenvalue problem for complex matrices. We have implemented it using MPI. The details of our current MPI implementation can be found in the NLAFET Working Note 10 [1].

We report on the performance of our current implementation in Section 8. The sequential version of our code is about twice as fast as its counterpart in LAPACK. However, we have recently realized that it is possible to improve on our parallel scalability and reduce our memory footprint.

Our current implementation can be downloaded from the NLAFET repository `Eigen vector`. For more details see Deliverable D7.5.

7.1 Robust block computation of eigenvectors

Consider the problem of solving a triangular linear system $S - \lambda I$ using a blocked algorithm. First partition the system conformally, and write

$$(S - \lambda I)x = \left(\begin{bmatrix} S_{11} & S_{12} & \dots & S_{1N} \\ & S_{22} & \dots & S_{2N} \\ & & \ddots & \vdots \\ & & & S_{NN} \end{bmatrix} - \lambda I \right) \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} = b. \tag{15}$$

This system can be solved using block backward substitution as in Algorithm 4.

Algorithm 4: $x = \texttt{BlockBackSolve}(S, \lambda, b)$

Data: A non-singular upper triangular matrix $S - \lambda I$ and a vector b partitioned conformally as in equation (15).

Result: The solution x of the linear system $(S - \lambda I)x = b$.

```
1 x = b;
2 for j = N, N − 1, . . . , 1 do
3 |   x_j ← f(S_jj, λ, b_j);
4 |   for i = 1, 2, . . . , N do
5 |   |   x_i ← g(x_i, S_ij, x_j);
6 return x;
```

In order to obtain a robust algorithm, one must replace the two kernels

$$f(S, \lambda, b) = (S - \lambda I)^{-1} b, \quad g(y, C, x) = y - Cx \tag{16}$$

with robust variants, which are given as Algorithm 5 `RobustBlockSolve`, and Algorithm 6 `RobustBlockUpdate`. It is critical to observe that they operate on augmented vectors $\langle \alpha, x \rangle$ representing vectors $\alpha^{-1} x$. The result is Algorithm 7 `RobustBlockBackSolve`. For a detailed discussion of the algorithms stated in this report we refer to NLAFET Working Notes 9 and 10 [9, 1]. Here we limit ourselves to making one critical observation. Any runtime system capable of executing a regular block triangular backward substitution can do so robustly. It is simply a matter of replacing the standard kernels with our robust kernels. Apart from the final re-scaling, the number and pattern of messages that have to be exchanged is unchanged. Moreover, the volume is increased marginally as we have to communicate the scaling factors.

Algorithm 5: $\langle \alpha, x \rangle = \texttt{RobustBlockSolve}(S, \lambda, \langle \beta, b \rangle)$

Data: A non-singular matrix $S - \lambda I$ and an augmented vector $\langle \beta, b \rangle$.
Result: An augmented vector $\langle \alpha, x \rangle$ such that

$$(S - \lambda I)(\alpha^{-1} x) = \beta^{-1} b$$

where the scaling α ensures that the computation of x cannot overflow.

$\langle \alpha, x \rangle = \texttt{RobustScalarBackSolve}(S, \lambda, b)$;
$\alpha \leftarrow \beta\alpha$;
return $\langle \alpha, x \rangle$;

Algorithm 6: $\langle \nu, z \rangle = \texttt{RobustBlockUpdate}(\langle \alpha, x \rangle, C, \langle \beta, y \rangle)$

Data: Augmented vectors $\langle \alpha, x \rangle$, $\langle \beta, y \rangle$ and a matrix C such that the linear transformation

$$z = y - Cx$$

is defined.
Result: An augmented vector $\langle \nu, z \rangle$ such that

$$\nu^{-1} z = \beta^{-1} y - C(\alpha^{-1} x)$$

where the scaling ν ensures that the computation of z cannot overflow.

$\gamma = \min\{\alpha, \beta\}$;
$x \leftarrow (\gamma/\alpha)x$;
$y \leftarrow (\gamma/\beta)y$;
$\xi \leftarrow \texttt{ProtectUpdate}(\|y\|_\infty, \|C\|_\infty, \|x\|_\infty)$;
$z \leftarrow \xi y - C(\xi x)$;
$\nu = \xi\gamma$;
return $\langle \nu, z \rangle$

Algorithm 7: $\langle \alpha, x \rangle = \texttt{RobustBlockBackSolve}(S, \lambda, b)$

Data: A partitioned, non-singular, upper triangular linear system $(S - \lambda I)x = b$.
Result: An augmented vector $\langle \alpha, x \rangle$ such that $(S - \lambda I)x = \alpha b$.

for $i = 1, \ldots, N$ **do**
  $\langle \alpha_i, x_i \rangle \leftarrow \langle 1, b_i \rangle$;
for $j = N, N-1, \ldots, 1$ **do**
  $\langle \alpha_j, x_j \rangle \leftarrow \texttt{RobustBlockSolve}(S_{jj}, \lambda, \langle \alpha_j, x_j \rangle)$;
  for $i = 1, 2, \ldots, j-1$ **do**
    $\langle \alpha_i, x_i \rangle \leftarrow \texttt{RobustBlockUpdate}(\langle \alpha_i, x_i \rangle, S_{ij}, \langle \alpha_j, x_j \rangle)$
$\alpha = \min\{\alpha_1, \alpha_2, \ldots, \alpha_N\}$;
for $i = 1, 2, \ldots, N$ **do**
  $x_i \leftarrow (\alpha/\alpha_i)x_i$;
return $\langle \alpha, x \rangle$

7.2 Simultaneous computation of multiple eigenvectors

Consider a matrix A for which a triangular Schur form $S = Q^H AQ$ has been computed. We now seek to determine the eigenvectors of S corresponding to a subset of the eigenvalues of A. We are also interested in back-transforming the computed eigenvectors of S to eigenvectors of A.

It is entirely possible to consider a general subset of the eigenvalues, but there is a certain clarity associated with the special case where all eigenvectors are sought. Strictly speaking, the general case can be reduced to this special case by reordering the eigenvalues of S, but this is beside the point. For the sake of simplicity, please assume that all eigenvalues are distinct. In this case Algorithm 8 can be used to simultaneously compute all eigenvectors.

Algorithm 8: X=`ScalarSimEigenvector`(S)

Data: An m by m upper triangular matrix S with distinct diagonal entries.
Result: A matrix $X = \begin{bmatrix} x_1 & x_2 & \dots & x_m \end{bmatrix}$ of eigenvectors of S such that $Sx_j = s_{jj}x_j$.

```
1 X ← I_m for j = m, m − 1, ..., 1 do
2 |   for k = j + 1 : m do
3 |   |   X(j,k) ← X(j,k) / (S(j,j) − S(k,k));
4 |   S(1:j−1, j:m) ← X(1:j−1, j:m) − S(1:j−1, j)X(j, j:m);
5 return X;
```

Algorithm 8 loops backwards through the m systems which are to be solved. The inner iteration computes a single row of X and the linear update has rank 1. As a result, the arithmetic intensity is very low.

However, it is straight forward to transform this algorithm into a block algorithm, see Algorithm 9 `BlockSimEigenvector`. It uses a pair of arrays `first` and `last` to keep track of the partitioning of S. Specifically, the first (last) row of the Jth diagonal block has global row index `first`(J) (`last`(J)). The algorithm loops backwards over the systems which are to be solved. It computes one block row of X per iteration. The variable k is always the dimension of the next linear system which will be solved. It is worth stressing the fact that while each of the diagonal blocks of S are read many times from cache memory, they are only read once from main memory. The linear update at the end of each iteration has the same rank as the dimension of the current diagonal block.

Algorithm 9: X=`BlockSimEigenvector`(S)

Data: An m by m upper triangular matrix S partitioned as an M by M block matrix, arrays `first` and `last` describing the partitioning.

Result: A matrix X of eigenvectors of X such that $SX = X\mathrm{diag}(T)$.

$X \leftarrow I_m$;
for $J = M, M-1, \ldots, 1$ **do**
  $f \leftarrow$ `first`(J);
  $l \leftarrow$ `last`(J);
  for $j = f+1 : l$ **do**
    $k \leftarrow j - f$;
    $X_{f:j-1,j} \leftarrow -(S_{f:j-1,f:j-1} - s_{jj}I_k)^{-1}S_{f:j-1,j}$;
  $k \leftarrow l - f + 1$;
  for $j = l+1 : m$ **do**
    $X_{f:l,j} \leftarrow (S_{f:l,f:l} - s_{jj}I_k)^{-1}X_{f:l,j}$;
  $X_{1:f-1,f:m} \leftarrow X_{1:f-1,f:m} - S_{1:f-1,f:l}X_{f:l,f:m}$;
return X

8 The performance of eigenvector computations

We report on a sequence of experiments testing our new parallel algorithm. It offers improved protection against overflow, uses augmented vectors, interleaves the computation of multiple eigenvectors and merges the backward substitution phase with the back-transformation. We give a brief description of the hardware, the test matrices, and the experimental methodology before presenting the results. We report on the single threaded execution time, the weak and the strong scaling of our current implementation.

8.1 Computer system

All experiments were executed on the Kebnekaise system at HPC2N, Umeå University. The relevant details are given in Section 4.1.

8.2 Test matrices

The test matrices were randomly generated from a seed with the real and complex parts being uniformly distributed on the interval $[0, 1]$. The generator is available on demand.

8.3 Experimental methodology

The high-resolution function `GetTimeOfDay` was used to measure runtimes, and the median execution time of three runs yielded reproducible numbers.

8.4 Software requirements

Our parallel algorithm is built using level 2 and 3 BLAS, as well as MPI calls. The user must provide libraries for both of those. We have linked all our tests against Intel MKL 2017.1.132 and Intel MPI 2017.1.132.

8.5 Sequential execution

In Figure 9 our MPI based algorithm is compared against the LAPACK `ZTREVC3` solver for

$$n \in \{10000, 20000, 30000, 40000\}. \tag{17}$$

All right eigenvectors are computed from a random upper triangular S, and are back-transformed using a dense unitary transformation matrix as input.

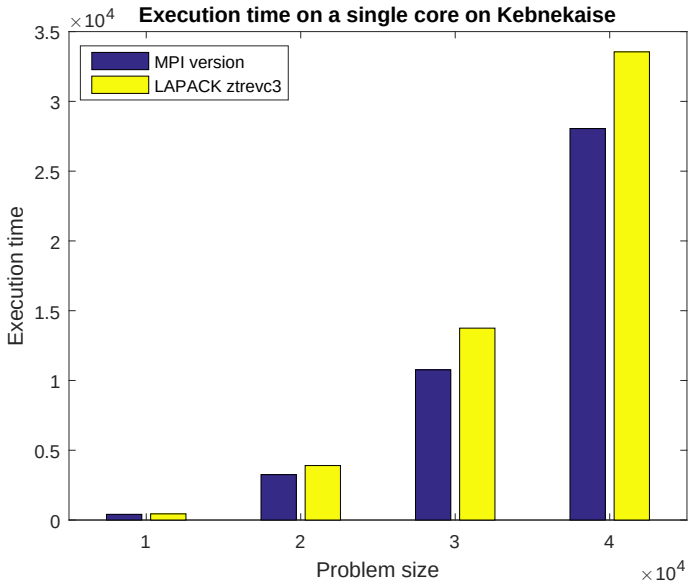


Figure 9: Execution time in seconds for our MPI based algorithm and ZTREVC3 from LAPACK for $n \in \{10000, 20000, 30000, 40000\}$.

It is clear that the runtimes are quite similar and our MPI version is marginally better. However, the MPI version has significant overhead, even when using a single MPI rank. A serial implementation of the algorithm, without the overhead gives a runtime of 13835 seconds for $n = 40000$, roughly half of what the MPI based implementation requires for completion. The LAPACK routine `ZTREVC3` deals with potential overflow by working with one column of S and right-hand-sides completely before moving on to next. The blocking strategy we propose is clearly beneficial as it allows for cache reuse.

8.6 Scalability

As for the eigenvalue reordering in Section 4.7, we have measured weak and strong scalability of our eigenvalue computation.

8.6.1 Weak scalability

The weak scalability was measured by scaling up the problem size n with the number of cores to keep the memory required per core constant. Specifically, for a problem of size n_1 on P_1 MPI ranks, the problem size n_P on P_P cores was set to $n_1\sqrt{P_P/P_1}$.

The results of the weak scalability experiments, with $n_1 = 15000$ and $P_1 = 16$ are shown in Figure 10.

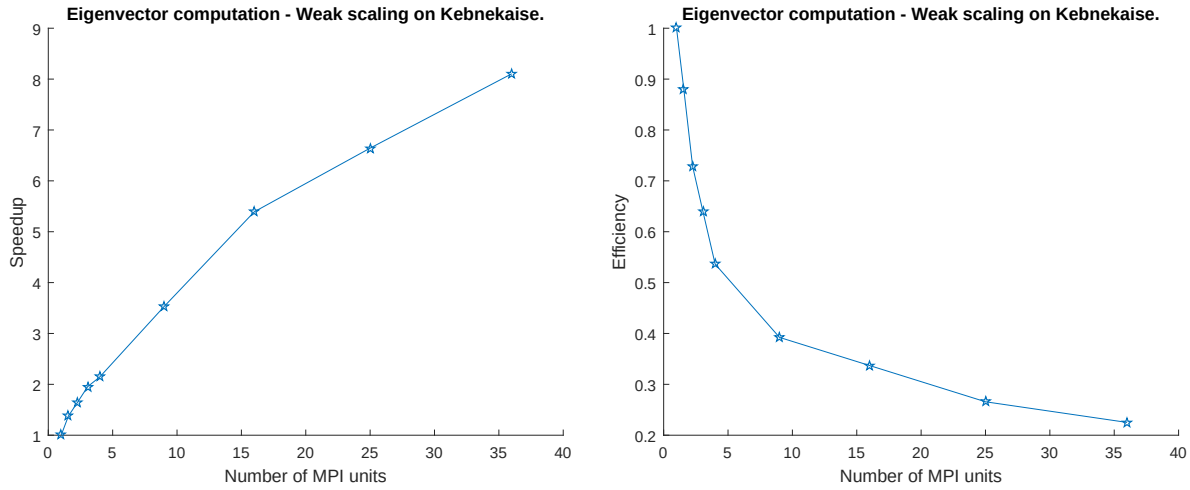


Figure 10: Speedup (left) and efficiency (right) achieved when running weak scalability performance test using up to 36 MPI units. One MPI unit consists of 16 MPI ranks.

8.6.2 Strong scalability

The strong scalability of the parallel algorithm was measured in terms of speedup relative to the parallel implementation running on one core for problems of size

$$n \in \{10000, 20000, 30000, 40000\}, \tag{18}$$

and meshes of size $P_r \times P_c$ for

$$P_r = P_c \in \{1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 28, 32\}. \tag{19}$$

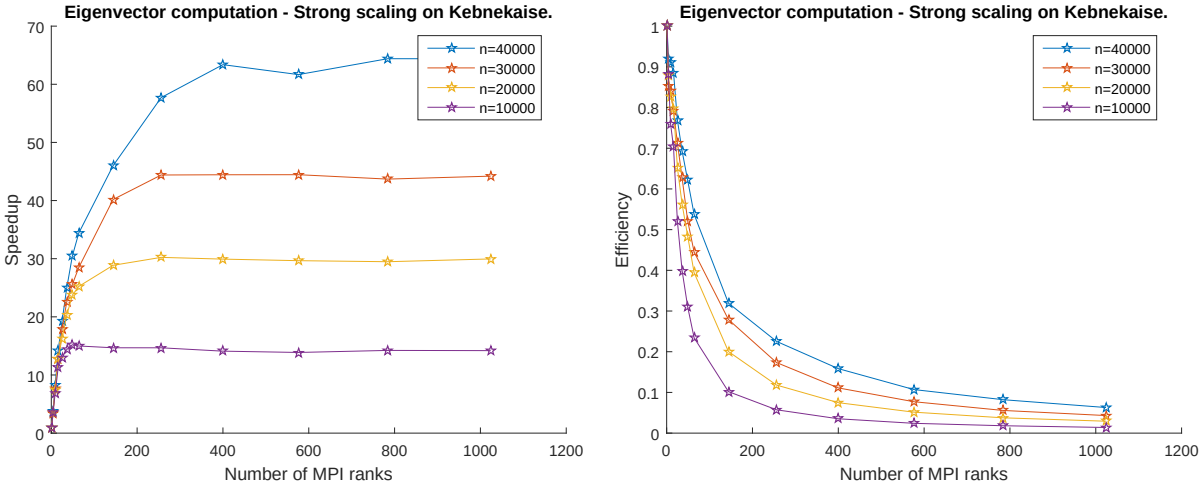


Figure 11: Speedup (left) and efficiency (right) achieved when running strong scalability performance test using up to 1024 MPI ranks for four problem sizes.

The results of the strong scalability experiments are shown in Figure 11. The strong scalability increases with larger problem sizes. The parallel efficiency drops dramatically when the number of cores is increased. This is hardly surprising as even the largest problem size ($n = 40,000$) is too small compared with the number of cores used.

8.7 Tunability

There is only one parameter which can be adjusted, namely the block size used to define the 2D block cyclic distribution of the matrices. It is not practical to change it dynamically during the execution of the code.

9 The next steps for eigenvector computation

There are several natural steps which will be taken in the immediate future. A standard error analysis for partitioned back-substitution using a unique scaling factor for each block row must be completed. The basic idea must be extended to a partitioned algorithm for Sylvester equations, so that eigenvectors can be computed in parallel and robustly from real Schur forms as well.

We will also improve our MPI implementation of the eigenvector computation. The most viable strategy is to maintain the block cyclic distribution used by ScaLAPACK, but begin the calculation by broadcasting all diagonal blocks of the Schur matrix along each process row. Moreover, all eigenvalues should be broadcast to all ranks. This will eliminate the rather elaborate scheme currently used to load balance the backward substitution.

In view of the success of our StarPU implementation for the eigenvalue reordering problem, it is realistic to develop a task based algorithm for computing eigenvectors in parallel while addressing overflow using augmented vectors.

10 Conclusion

Our work has focused on two topics, eigenvalue reordering and eigenvector computation. We are pleased to report progress on these topics. We now understand how to fight overflow when solving a triangular linear system in parallel. The key is to assign a unique scaling factor to each block row and replace all necessary kernels with robust counterparts. We understand how it is possible to interleave the robust computation of multiple eigenvectors in order to improve the arithmetic intensity.

In terms of software, we have developed and implemented two new algorithms. We can solve the problem of reordering eigenvalues in real Schur form using our task based algorithm. Our implementation can be executed using StarPU. We routinely exceed 50% of the peak flop rate. We have recorded weak and strong scalability efficiencies well above 50%. We are currently rewriting and extending our code to run efficiently on a distributed memory machine using StarPU.

We have developed a parallel and robust algorithm for computing eigenvectors from complex Schur forms. Our current implementation uses MPI. The numerical scaling works as it should. We anticipate some changes to the memory layout in the immediate future, but then we will turn to the task of producing task based software.

The two-sided transformation algorithm used for reordering eigenvalues encapsulates many of the aspects found in the other algorithms used in eigenvalue computations. We are confident that the lessons learned will allow us to develop task based algorithms and codes for these problems.

Having cracked the problem represented by the parallel bottleneck associated with robust backward substitution, we are also confident that task based algorithms for eigenvector computation will be developed and implemented successfully.

References

[1] Björn Adlerborn, Carl Christian Kjelgaard Mikkelsen, Lars Karlsson, and Bo Kågström. Towards Highly Parallel and Compute-Bound Computation of Eigenvectors fo Matrices in Schur Form. *NLAFET Working Note* WN-10, April, 2017. Also as Report UMINF 17.10, Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden.

[2] Edward Anderson. Robust Triangular Solves for Use in Condition Estimation. LAWN 36, Cray Research Inc., August 1991.

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

[4] Mark R. Fahey. New Complex Parallel Eigenvalue and Eigenvector Routines. Technical report, Computational Migration Group, Computer Science Corporation, 2001.

[5] 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.

[6] 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.

[7] B. Kågström and P. Poromaa. Computing eigenspaces with specified eigenvalues of a regular matrix pair (A, B) and condition estimation: theory, algorithms and software. *Numer. Algorithms*, 12(3-4):369–407, 1996.

[8] D. Kressner. Block Algorithms for Reordering Standard and Generalized Schur Forms. *ACM Transactions on Mathematical Software*, 32(4):521–532, December 2006.

[9] Carl Christian Kjelgaard Mikkelsen and Lars Karlsson. Robust solution of triangular linear systems. *NLAFET Working Note* WN-9, April, 2017. Also as Report UMINF 17.9, Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden.

[10] Mirko Myllykoski, Carl Christian Kjelgaard Mikkelsen, Lars Karlsson, and Bo 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.