Towards Highly Parallel and Compute-Bound Computation of Eigenvectors of Matrices in Schur Form *

Björn Adlerborn,
Carl Christian Kjelgaard Mikkelsen,
Lars Karlsson,
Bo Kågström

May 30, 2017

Abstract

In this paper we discuss the problem of computing eigenvectors for matrices in Schur form using parallel computing. We develop a new parallel algorithm and report on the performance of our MPI based implementation. We have also implemented a new parallel algorithm for scaling during the backsubstitution phase. We have increased the arithmetic intensity by interleaving the compution of several eigenvectors and by merging the backward substitution and the back-transformation of the eigenvector computation.

Contents

*NLAFET Working Note 10. Report UMINF 17.10, Dept. Computing Science, Umeå University, SE 901 87 Umeå, Sweden.

1 Introduction

Given a dense, non-Hermitian, matrix $A \in \mathbb{C}^{n\times n}$ and a Schur decomposition $A = UTU^H$ of A, we seek to compute the eigenvectors of A corresponding to a user-defined subset Λ containing m of the n eigenvalues $\lambda(A)$ of A. For a given eigenvalue λ, a non-trivial solution x of the singular system $(T-\lambda I)x = 0$ is computed. This vector is an eigenvector of T corresponding to the eigenvalue λ. The transformation $x \leftarrow Ux$ converts x into an eigenvector of A.

In the latest version of `LAPACK` (version 3.7.0), the entire procedure is accomplished by the routines labeled `xTREVC3`. There is support for both real and complex flavors. In the latest version of `ScaLAPACK` (version 2.0.2) the complex case is implemented as `PCTREV` and `PZTREV`, but with no protection against overflow as this is nontrivial to accomplish in a parallel context. There is no support for real flavors in `ScaLAPACK`.

The ultimate goal our work is develop a novel task based algorithm which guards against overflow and is available in both real and complex flavors.

We focus on the following challenges:

1. *Robustness.* The matrix $T - \lambda I$ can be ill-conditioned and therefore the solver for the linear system needs to be robust against overflow.

2. *Cache reuse.* Computing just a single eigenvector is a memory-bound operation. As soon as multiple eigenvectors are computed, there is potential to increase the cache reuse and transform the operation into a compute-bound one.

In the immediate future we will focus on the following problem

3. *Scalability.* The algorithm needs to be expressed in terms of a fine-grained task graph and scheduled according to the data flow in order for the computation to scale to a potentially heterogeneous distributed memory system.

The rest of the paper is organized as follows. A formal description of the problem of computing eigenvectors is given in Section 2. The problems associated with the robust computation of a *single* eigenvector is discussed in 3. We develop sequential, blocked and parallel algorithms for this problem. It is also possible to interleave the computation of several eigenvectors in order to improve the arithmetic intensity. This and related issues are discussed in Section 4. In Section 5 we describe relevant aspects of our current MPI implementation. The results of our numerical experiments are presented in Section 6. Finally, in Section 7 we summaries our findings and outline our future plans.

2 Formal problem description

Let $A \in \mathbb{C}^{n\times n}$ be a non-symmetric matrix and let $A = UTU^H$ be a Schur decomposition of A, i.e. $U \in \mathbb{C}^{n\times n}$ is a unitary matrix and $T \in \mathbb{C}^{n\times n}$ is an upper triangular matrix. Let $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$ be a strictly increasing subset of $\{1, 2, \ldots, n\}$ which identifies the user's selection of eigenvalues by their position on the diagonal of T. The eigenvalue associated with the index s_i is $\lambda_i = t_{s_i,s_i}$. The goal is to compute the columns of a matrix $X = \begin{bmatrix} x_1, x_2, \ldots, x_m \end{bmatrix} \in \mathbb{C}^{n\times m}$ such that $x_i \in \mathbb{C}^n$ is an eigenvector for A associated with the eigenvalue λ_i, i.e. $Ax_i = \lambda_i x_i$ for $i = 1, 2, \ldots, m$.

There are several variations of this problem that are important to support in library software. For example:

- *No back transformation.* The user may only require the eigenvectors of T and chooses not to execute the back transform.
- *Left eigenvectors.* Instead of or (in addition to) computing a subset of the right eigenvectors, the user may also want to compute a subset of the left eigenvectors.
- *Real Schur form.* If the matrix A is real instead of complex, then it is usually reduced to real Schur form. Here U is a real orthogonal matrix and T is a real upper quasi-triangular matrix with 1×1 and 2×2 blocks on the diagonal. The 2×2 blocks represent complex conjugate pair of eigenvalues and require special treatment.
- *Generalized eigenvalue problem.* The goal is to compute a generalized eigenvector x for a generalized eigenvalue pair (λ, β) such that $\lambda Ax = \beta Bx$.

In this paper we focus on the standard eigenvalue problem for complex matrices. Next, we will extend our new results to real matrices and further extend the results to counterparts of the generalized eigenvalue problem.

3 Computation of a single eigenvector

Consider a selected eigenvalue λ at position s along the diagonal of the upper triangular T of size $n \times n$. We seek to find a non-zero solution x to the singular linear system

$$(T - \lambda I)x = 0. \tag{1}$$

After the subsequent back-transformation $x \leftarrow Ux$ we get an eigenvector x for the eigenvalue λ such that $Ax = \lambda x$.

Let us find a non-trivial solution to (1). Partition the system into a 3×3 block system such that the eigenvalue λ is exposed at position (s, s):

$$\left(\begin{bmatrix} T_{11} & t_{12} & T_{13} \\ 0 & \lambda & t_{23}^T \\ 0 & 0 & T_{33} \end{bmatrix} - \lambda \begin{bmatrix} I & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & I \end{bmatrix} \right) \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

The third block equation reads $(T_{33} - \lambda I)x_3 = 0$ and has the trivial solution $x_3 = 0$. The choice of $x_3 = 0$ ensures that the second block equation is satisfied. Substituting $x_3 = 0$ into the first block equation gives

$$(T_{11} - \lambda I)\, x_1 = -x_2 t_{12}. \tag{2}$$

Note that there are s degrees of freedom (the $s-1$ components of x_1 plus the scalar x_2) but only $s-1$ equations. If we assume that the eigenvalues of T are simple, then $T_{11} - \lambda I$ is guaranteed to be non-singular and the solution x_1 is therefore unique and computable once we have chosen a value for x_2. Setting $x_2 = 0$ implies that $x_1 = 0$ and by extension $x = 0$, which results in the undesirable trivial solution. We must therefore choose $x_2 \neq 0$, which is sufficient to ensure that $x \neq 0$, and then solve (2) for x_1.

Algorithm 1: $x = \texttt{ComputeEigenvector}(U, T, s)$

Data: A Schur decomposition $A = UTU^H$ and the position s along the diagonal of T of a selected eigenvalue.

Result: An eigenvector x of A associated with the simple eigenvalue λ such that $Ax = \lambda x$.

`// Set up the linear system` $(R - \lambda I)x_1 = x_2 b$

1 $\lambda \leftarrow T(s, s)$;

2 $b \leftarrow -T(1 : s-1, s)$;

3 $R \leftarrow T(1 : s-1, 1 : s-1)$;

4 $x_2 \leftarrow$ any non-zero value;

`// Solve the linear system`

5 Solve for x_1 in $(R - \lambda I)x_1 = x_2 b$;

`// Assemble the eigenvector`

6 $x \leftarrow \begin{bmatrix} x_1 \\ x_2 \\ 0_{n-s} \end{bmatrix}$;

`// Back-transform the eigenvector`

7 $x \leftarrow Ux$;

8 **return** x;

In summary, Algorithm 1 computes an eigenvector x of A for a given eigenvalue λ at position s along the diagonal of T. The algorithm spends $\Theta(s^2)$ flops on solving the linear system and $\Theta(ns)$ flops on the back-transformation provided that the structure of x is utilized. By simply applying the algorithm to each $s \in \mathcal{S}$, we completely solve the problem. However, this approach is inefficient for the following reasons:

1. The solve can overflow. To resolve this issue, we gradually develop robust solvers in Section 3.1 (scalar), Section 3.2 (block), and Section 3.3 (parallel).

2. The solve is memory-bound. To resolve this issue, we develop a compute-bound solver that simultaneously solves for several selected eigenvalues in Section 4.1.

3. The back-transformation is memory-bound. To resolve this issue, we review a technique to obtain a compute-bound back-transformation by simultaneously back-transforming eigenvectors for several selected eigenvalues in Section 4.2.

3.1 A scalar algorithm

In this section we present a robust algorithm for solving the scaled upper triangular linear system

$$(T - \lambda I)x = \alpha b \tag{3}$$

Our algorithm is almost identical to the algorithm implemented in `LAPACK` as the functions `xLATRS` [1]. Our contribution is to simplify and extend the analysis. The majority of this work is contained in the technical report [2] which deals exclusively with special case of $\lambda = 0$. In this note we simply adapt our earlier results to the general case.

The upper triangular linear system

$$(T - \lambda I)x = b \tag{4}$$

can be solved using scalar back-substitution; see Algorithm 2 `ScalarBackSolve`.

Algorithm 2: $x = \texttt{ScalarBackSolve}(T, \lambda, b)$

Data: A non-singular upper triangular matrix $T - \lambda I$ of size $n \times n$, a vector b of size $n \times 1$.

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

```
x = b;
for j = n, n − 1, ..., 1 do
    x_j ← x_j / (t_jj − λ);
    for i = 1, 2, ..., j − 1 do
        x_i ← x_i − t_ij x_j;
return x
```

We observe that `ScalarBackSolve` consists of a sequence of scalar divisions

$$x \leftarrow b/(t - \lambda), \quad t - \lambda \neq 0 \tag{5}$$

and scalar updates

$$y \leftarrow y - tx. \tag{6}$$

In the following sections, Ω represents a large positive number smaller than the largest representable floating point number. Our purpose it to prevent all intermediate results from exceeding this threshold. A word of caution is inserted here. It is entirely possible to engineer a situation where the scaling factors *underflow* in floating point arithmetic. If we limit ourselves to powers of 2 and use 64 bits integers, then we can represent scalings as small as $2^{-(2^{64}-1)} \approx 10^{-5.55\times10^{18}}$. This dramatically reduces the chance of the scaling underflowing. The details are omitted here.

3.1.1 Robust scalar divisions

In general, the scalar division (5) cannot exceed the overflow threshold Ω if

$$|b| \leq |t - \lambda|\Omega.$$

Algorithm 3 shows how to compute a scalar α, such that the scaled division

$$x \leftarrow \frac{\alpha b}{t - \lambda} \tag{7}$$

cannot exceed the overflow threshold Ω. Moreover, all inequalities can be evaluated without exceeding the overflow threshold. We refer to [2] for an analysis of this algorithm.

Algorithm 3: $\alpha = \texttt{ProtectDivision}(b, t, \lambda)$

Data: Numbers b and t such that $|b| \leq \Omega$ and $t \neq \lambda$.
Result: A scaling factor $\alpha \in (0, 1]$ such that the scaled division (7) cannot exceed the overflow threshold.

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

3.1.2 Robust linear updates

In this section we consider the more general problem of computing a scalar ξ, such that the scaled linear matrix transformation

$$Z \leftarrow \xi Y - C(\xi X) \tag{8}$$

cannot exceed the overflow threshold.

The following theorem gives a condition which ensures that the linear transformation

$$Z \leftarrow Y - CX. \tag{9}$$

cannot exceed the overflow threshold.

Theorem 1. *Let Y, C, X be matrices such that $Z = Y - CX$ is defined. If*

$$\|Y\|_\infty + \|C\|_\infty \|X\|_\infty \leq \Omega, \tag{10}$$

then overflow is impossible in the calculation of Z regardless of the order of the necessary arithmetic operations.

Proof. The proof consists of an elementary application of the triangle inequality. The details are given in [2]. □

Algorithm 4 `ProtectUpdate` shows how to compute a scalar ξ, such that the scaled linear matrix transformation

$$z \leftarrow \zeta Y - C(\zeta X) \tag{11}$$

can not exceed the overflow threshold. We refer to [2] for an analysis of this algorithm.

Algorithm 4: $\zeta = \texttt{ProtectUpdate}(y_{\text{norm}}, c_{\text{norm}}, x_{\text{norm}})$

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 (11) cannot exceed the overflow threshold.

```
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;
```

3.1.3 Robust scalar backward substitution

It is now straightforward to formulate a robust algorithm for solving the scaled upper triangular linear system (3); see Algorithm 5. At this point the reader is encouraged to briefly pause and consider the problem of parallelizing Algorithm 5 `RobustScalarBacksolve`.

Algorithm 5: $(\alpha, x) = \texttt{RobustScalarBacksolve}(T, \lambda, b)$

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

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

and numbers c_j satisfying

$$\|T_{1:j-1,j} - \lambda I\|_\infty \leq c_j, \quad j = 2, 3, \ldots, n.$$

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

$$(T - \lambda I)x = \alpha b,$$

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

$\alpha \leftarrow 1$, $x \leftarrow b$, $x_{\max} \leftarrow \|x\|_\infty$;
for $j \leftarrow n, n-1, \ldots, 1$ **do**
  $\beta = \texttt{ProtectDivision}(x_j, \lambda, t_{jj})$;
  if $\beta \neq 1$ **then**
    $x \leftarrow \beta x$;
    $\alpha \leftarrow \beta\alpha$;
  $x_j \leftarrow \frac{x_j}{t_{jj}-\lambda}$;
  if $j > 1$ **then**
    $\beta = \texttt{ProtectUpdate}(x_{\text{norm}}, c_j, |x_j|)$;
    if $\beta \neq 1$ **then**
      $x \leftarrow \beta x$;
      $\alpha \leftarrow \beta\alpha$;
    $x_{1:j-1} \leftarrow x_{1:j-1} - T_{1:j-1,j}x_j$;
    $x_{\max} \leftarrow \|x_{1:j-1}\|_\infty$;
return α, x;

3.2 A block algorithm

It is difficult to parallelize the sequential Algorithm 5 `RobustScalarBackSolve`. The fundamental problem is the need for global communication when recomputing the value of $x_{\max}$ at the end of every iteration. Moreover, every re-scaling of x also requires global communication. It is clear that another approach is required.

To this end we consider the problem of solving the linear system (4) using a blocked algorithm. First we partition the system conformally, and write

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

This system can be solved using block back-substitution as in Algorithm 6 `BlockBackSolve`. In order to obtain a robust algorithm we must replace the back solve and update kernels

$$f(T, \lambda, b) = (T - \lambda I)^{-1} b, \quad g(y, C, x) = y - Cx \tag{13}$$

with robust variants.

Algorithm 6: $x =$ `BlockBackSolve`(T, λ, b)

Data: A non-singular upper triangular matrix $T - \lambda I$ and a vector b partitioned conformally as in equation (12).

Result: The solution x of the linear system $T - \lambda I x = b$

```
x = b;
for j = N, N − 1, ..., 1 do
    x_j ← f(T_jj, λ, b_j);
    for i = 1, 2, ..., N do
        x_i ← g(x_i, T_ij, x_j);
return x;
```

We require the following definition.

Definition 1. *An augmented vector $\langle \alpha, x \rangle$ consists of a scalar $\alpha \in (0, 1]$ and a vector $x \in \mathbb{C}^n$ and represents the vector $y = \alpha^{-1} x$. Two augmented vectors $\langle \alpha, x \rangle$ and $\langle \beta, y \rangle$ are equivalent if they represent the same vector, i.e.*

$$\langle \alpha, x \rangle \sim \langle \beta, y \rangle \quad \Leftrightarrow \quad \alpha^{-1} x = \beta^{-1} y.$$

Two augmented vectors $\langle \alpha, x \rangle$ and $\langle \beta, y \rangle$ are consistently scaled if $\alpha = \beta$.

We will use augmented vectors to represent vectors which would otherwise exceed the overflow threshold. Augmented vectors which are consistently scaled can be added, although their sum might exceed the overflow threshold, unless an additional scaling is applied. In general, vectors should be consistently scaled before addition are attempted.

3.2.1 Robust block solve

Consider the problem of solving a linear shifted linear system

$$(T - \lambda I)(\alpha^{-1}x) = \beta^{-1}b, \tag{14}$$

where the right hand side is not given explicitly, but is represented by an augmented vector $\langle \beta, b \rangle$. We seek an augmented vector $\langle \alpha, x \rangle$, where the scaling factor α must be chosen to ensure that the computation of x does not overflow.

Algorithm 7: $\langle \alpha, x \rangle = \texttt{RobustBlockSolve}(T, \lambda, \langle \beta, b \rangle)$

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

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

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

1 $\langle \alpha, x \rangle = \texttt{RobustScalarBackSolve}(T, \lambda, b)$;
2 $\alpha \leftarrow \beta\alpha$;
3 **return** $\langle \alpha, x \rangle$;

It is straightforward to verify that Algorithm 7 `RobustBlockSolve` accomplishes its task. The call to Algorithm 7 `RobustBlockSolve` in line 1 returns an augmented vector $\langle \alpha, x \rangle$ such that

$$(T - \lambda I)x = \alpha b. \tag{15}$$

It follows immediately that

$$(T - \lambda I)(\alpha\beta)^{-1} = \beta^{-1}b \tag{16}$$

which explains why the line 2 is correct.

3.2.2 Robust block update

Now consider a pair of augmented vectors $\langle \alpha, x \rangle$ and $\langle \beta, y \rangle$, representing the vectors $\alpha^{-1}x$ and $\beta^{-1}y$, and a matrix C such that the transformation

$$z \leftarrow y - Cx$$

is defined. We seek an augmented vector $\langle \nu, z \rangle$ such that

$$\nu^{-1}z = \beta^{-1}y - C(\alpha^{-1}x) \tag{17}$$

and where the scaling ν prevents overflow during the computation of z. This problem is solved by Algorithm 8. While the central calculation is completed by Algorithm 4 `ProtectUpdate`, it is still important to appreciate the effect of the first three steps. These instructions replace $\langle \alpha, x \rangle$ and $\langle \beta, y \rangle$ with a pair of augmented vectors which are consistently scaled and represent the vectors $\alpha^{-1}x$ and $\beta^{-1}y$.

3.2.3 Robust block backward substitution

We can now given a robust block algorithm for solving a shifted partitioned linear system. This is Algorithm 9 `RobustBlockBackSolve`. It is virtually identical to Algorithm 6. We have simply replaced the central kernels which operate on vectors with robust kernels which perform equivalent operations on augmented vectors.

Algorithm 8: $\langle \alpha, x \rangle = \texttt{RobustBlockUpdate}(\langle \alpha, x \rangle, C, \langle \beta, y \rangle)$

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 9: $\langle \alpha, x \rangle = \texttt{RobustBlockBackSolve}(T, \lambda, b)$

Data: A block partitioned, non-singular, upper triangular linear system $(T - \lambda I)x = b$.

Result: An augmented vector $\langle \alpha, x \rangle$ such that $(T - \lambda I)x = \alpha b$.

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

3.3 Parallel algorithms

Any run-time system such as `StarPU` which can execute the partitioned Algorithm 6 `BlockBackSolve` can also execute the robust variant Algorithm 9 `RobustBackSolve`. It is simply a matter of replacing the standard kernels with robust kernels operating on augmented vectors. The communication patterns will be identical, save for a single global communication needed to enforce a consistent scaling at the end the of the computation. The messages exchanged during the backward substitution must each be augmented with a single number, the scaling factor, but this is a modest price to pay.

4 Computation of multiple eigenvectors

In this section we discuss how to accelerate the computation of eigenvectors. The fundamental problem is that the computation of a single eigenvector is memory bound. Specifically, if we seek an eigenvector x of the upper triangular matrix T corresponding to the eigenvalue $\lambda = t_{ss}$, then we have to solve a triangular linear system of dimension $s-1$. We have to complete $O(s^2)$ arithmetic operations on $O(s^2)$ data. If we wish to back-transform the computed eigenvector x, i.e apply the transformation $x \leftarrow Ux$, then a further $O(ns)$ arithmetic operations must be performed on $O(ns)$ data. We observe that the arithmetic intensity is $O(1)$. It is therefore impossible to execute this problem efficiently on modern computer architectures with deep memory hierarchies, which favor codes with high arithmetic intensities. However, if multiple eigenvectors are required, then it possible to organize the computations in an efficient manner.

4.1 Simultaneous backward substitution

Consider the matrix A for which a triangular Schur form T has been computed. We now seek to determine the eigenvectors of T corresponding to a subset of the eigenvalues of A. We are also interested in back-transforming the computed eigenvectors of T 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 T, but this is beside the point. For the sake of simplicity we now assume that all eigenvalues are distinct. In this case Algorithm 10 can be used to simultaneously compute all eigenvectors.

Algorithm 10: X=`ScalarSimEigenvector`(T)

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

```
X ← I_m for j = m, m−1, ..., 1 do
    for k = j+1 : m do
        X(j,k) ← X(j,k) / (T(j,j) − T(k,k))
    X(1:j−1, j:m) ← X(1:j−1, j:m) − T(1:j−1, j) X(j, j:m)
return X;
```

Algorithm 10 loops backwards through the m triangular systems 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.

It is straight forward to transform this algorithm into a block algorithm; see Algorithm 11 `BlockSimEigenvector`. This algorithm utilizes a pair of arrays `first` and `last` to keep track of the partitioning of T. Specifically, the first (last) row of the Jth block has global row index `first`(J) (`last`(J)). The algorithm loops backwards over the systems which are to be solved and it computes one block row of X per iteration. The variable k is always the dimension of the next linear system to be solved. It is worth stressing that each of the diagonal blocks of T are read several times from cache memory, but only read once from main memory. The linear update at the end of each iteration has the same rank k as the dimension of the current diagonal block.

Algorithm 11: X=`BlockSimEigenvector`(T)

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

Result: A matrix X whose columns are eigenvectors of T such that $TX = X\text{diag}(T)$.

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

4.2 Simultaneous back-transformation

The matrix X of eigenvectors of T can be be transformed into eigenvectors of $A = UTU^T$ by the transformation $X \leftarrow UX$. This can either be done efficiently, several columns at a time, or inefficiently, one column at a time. There is however, an third option which we explore in the next section.

4.3 Simultaneous substitution and gradual transformation

Algorithm 11 `BlockSimEigenvector` computes the components of the matrix X of eigenvectors of T one block row at a time. It is therefore possible to execute the back-transform $X \leftarrow UX$ gradually. This is done in Algorithm 12. The advantage of this approach is that freshly computed entries of X are utilized at once and do not have to be read from memory until X has been fully computed. We stress that the transformation matrix is read gradually one block column at a time and only once.

Algorithm 12: X=`BlockSimEigenvectorBackTransform`(T,U)

Data: An m by m upper triangular matrix T partitioned as an M by M block matrix, arrays `first` and `last` describing the partitioning. An orthonormal matrix U representing the back-transformation.

Result: A matrix X of eigenvectors of $A = UTU^T$ such that $AX = X\text{diag}(T)$.

`// Initialize` X `as the identity and zero out` Y`.`

$X \leftarrow I_m$;
$Y \leftarrow O_{m \times m}$;
for $J = M, M-1, \ldots, 1$ **do**
  $f \leftarrow \texttt{first}(J)$;
  $l \leftarrow \texttt{last}(J)$;
  for $j = f+1 : l$ **do**
    $k \leftarrow j - f$;
    $X_{f:j-1,j} \leftarrow -(T_{f:j-1,f:j-1} - I_k)^{-1} T_{f:j-1,j}$;
  $k \leftarrow l - f + 1$;
  for $j = l+1 : m$ **do**
    $X_{f:l,j} \leftarrow (T_{f:l,f:l} - t_{jj} I_k)^{-1} X_{f:l,j}$;
  $X_{1:f-1,f:m} \leftarrow X_{1:f-1,f:m} - T_{1:f-1,f:l} X_{f:l,f:m}$;
  $Y_{1:m,f:m} \leftarrow Y_{1:m,f:m} + Y_{1:m,f:l} X_{f:l,f:m}$;
$X \leftarrow Y$;
return X;

4.4 Robust simultaneous computation of several eigenvectors

It is possible to simultaneously compute several eigenvectors in a robust manner. In Algorithm 11 it is simply a matter of replacing the linear updates and linear solves with robust variants operating on augmented vectors. It is critical to appreciate that we need a scaling factor for each block row of each column of the matrix of eigenvectors of T.

5 MPI-based parallel algorithm

The previous sections describe how to robustly and efficiently compute eigenvectors in parallel. Here we discuss aspects of our current MPI implementation. A task based implementation will be developed in the immediate future. Here we expect to capitalize on the lessons learned writing the MPI implementation.

Our current implementation use point-to-point messages via synchronous MPI send and receive operations. The limited number of broadcasts and reductions are efficiently executed using scatter and gather operations.

5.1 Data distribution and memory layout

The input matrices T and U must be identically distributed across the $P_\text{r} \times P_\text{c}$ process grid using a standard two-dimensional block-cyclic distribution with the same block size n_b in both dimensions; see the left of Figure 1 for an example.

The matrix X, is distributed using a 1-dimensional block-cyclic distribution with a row block size n_b. The column block size is equal to the the number of selected eigenvalues,

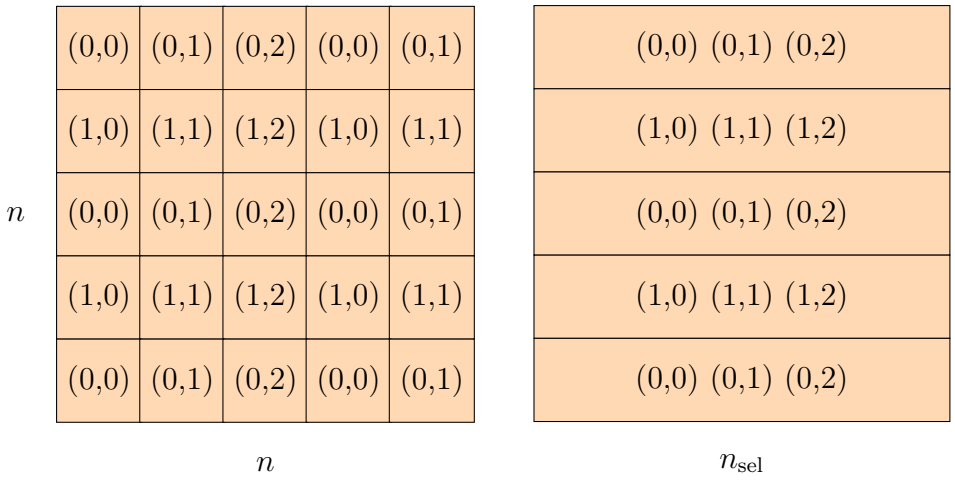

Figure 1: Left: Block cyclic data distribution of the T and U matrices, $n \times n$, exemplified using a 2×3 grid, where the residence for each of the 25 matrix blocks is given by its process coordinate $(P_{\text{r}}, P_{\text{c}})$ with $0 \leq P_{\text{r}} < 2 = P_{\text{r}}$ and $0 \leq P_{\text{c}} < 3 = P_{\text{c}}$. Right: Block cyclic data distribution of the X and W matrices, $n \times n_{\text{sel}}$, exemplified using a 2×3 grid, where the replicated residence of the 5 matrix blocks is given by process coordinates $(P_{\text{r}}, P_{\text{c}})$ with $0 \leq P_{\text{r}} < 2 = P_{\text{r}}$ and $0 \leq P_{\text{c}} < 3 = P_{\text{c}}$.

n_{sel}. Normally, one would store each block row on a single process, for example on the first column process for each process row, but in order to improve parallelism all processors in a process row must allocate space for the full block row. See the right of Figure 1 for an example.

Thus, the total storage requirement for X is $n \cdot n_{\text{sel}} \cdot P_{\text{c}}$. Internally, memory is required for the storage of right-hand-sides W. This matrix has the same distribution and memory requirements as X. To update W in parallel using a delayed update technique, we also need storage for concurrently computed solutions X_{sol}. To this end, each process will allocate $n_{\text{sel}} \cdot (n/n_{\text{b}})/\min(P_{\text{r}}, P_{\text{c}})$ for X_{sol}, or in total $n_{\text{sel}} \cdot \max(P_{\text{r}}, P_{\text{c}}) \cdot (n/n_{\text{b}})$. In summary, the total memory requirements are $\text{O}(2n(n + n_{\text{sel}} \cdot P_{\text{c}}))$. For very large problems with a large fraction of selected eigenvalues this might not be feasible. We could reduce the memory requirement at the expense of efficiency.

5.2 Overview of the algorithm

The parallel algorithm is based and extends upon the serial blocked version; see Algorithm 9 `RobustBlockBackSolve`. It implements the suggested scaling technique for both the solve and update parts.

The algorithm loops backwards over the block columns and backwards over the columns of each block. As it passes the column containing a selected eigenvalue, the eigenvalue becomes active and the corresponding system is added to the set of systems which are being solved.

After we have solved all the linear systems represented by a single $n_{\text{b}} \times n_{\text{b}}$ diagonal block and all active eigenvalues, we use the new information to update nearby right-hand-sides only, and postpone the remaining updates. The goal is to remove all dependencies for the next few diagonal blocks. The algorithm for controlling the update process is complicated and a specific example is given in Figure 2.

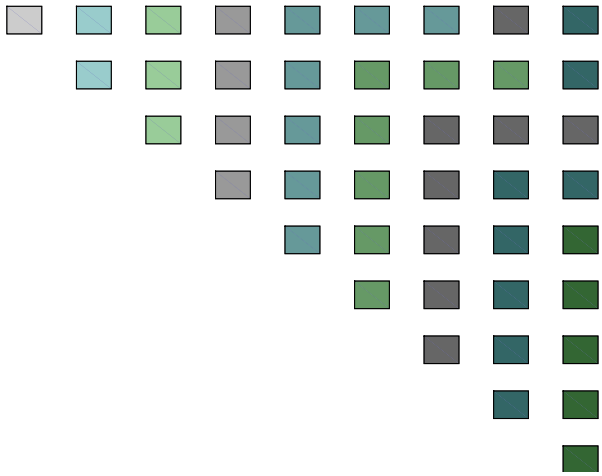

Figure 2: Reference pattern in T illustrated in a 10×10 block matrix, distributed over a 4×4 grid. After a diagonal block has been solved, the blocks that are referenced for the following update of the right-hand-sides are marked in the same color.

Algorithm 13 `ParaEigVecSolve` summarizes how the eigenvectors of given matrix T is computed in parallel. Here we assume that all eigenvectors are wanted.

The algorithm is based on 7 functions:

- The function `Reduce` is an MPI based reduction that performs summation of a vector/matrix over a process column.
- The sequential function `Solve` accepts as input a sequence of linear systems, one system for each active eigenvalue. It uses a robust backward substitution algorithm and returns each solution as an augmented vector with a seperate scaling factor.
- The function `Broadcast` is an MPI based one-to-all communication, over a process row or column that replicates data.
- The sequential function `Transform` performs a partial back-transform of X using current solutions and U.
- The sequential function `Update` performs a robust linear update of right-hand-sides.
- The function `Solving` returns true if the caller is solving a block.
- The function `Reducing` returns true if the caller is taking part in the reduction of right-hand-sides.

The functions `Solve`, `Transform`, and `Update` are briefly described in the next section.

5.3 Main kernels

In this section we describe the main kernels used in Algorithm 13 `ParaEigVecSolve`.

5.3.1 Solve

The sequential function `Solve` accepts as input a sequence of the linear systems of the from $(T_{\text{ii}} - \lambda_j I)x = b_j$, where T_{ii} is a diagonal block and $\{\lambda_j\}$ is the set of active eigenvalues and b_j is the appropropriate right hand side vectors. The process owning the block T_{ii} performs the solves, as described in Algorithm 9 `RobustBlockBackSolve`, and stores the solutions in a vector, with scalings appended at the end of the vector. The vector is broadcasted along current mesh-row, so that all processes along the mesh row receives both solutions and scalings. Each selected eigenvector renders a unique solution, of size n_{b}, and a scaling factor. For each block there are at most n_{b} selected eigenvalues, which limits the amount of data broadcasted to $n_{\text{b}}^2 + n_{\text{b}}$.

5.3.2 Transform

The function `Transform` applies a block column of U to at most n_{b} solutions, using a `GEMM`(level 3) operation, and stores the back-transformed solutions in X. This is done without communication, as each process owning part of the block column of U already has the required n_{b} solutions and stores the computed result in their own copy of X. However, the scalings used might differ, so the already back-transformed data in X and solutions are forced to be consistently scaled before they are merged.

5.3.3 Update

The function `Update` operates on a block of T_{ij} and updates the corresponding right-hand-sides with computed solutions using Algorithm 8 `RobustBlockUpdate`. This function requires no communication, as solutions and scalings were distributed during earlier solves. The result is stored locally, and later combined during the `Reduce` operation before the next `Solve`. Internally `GEMV`(level 2) operations are used to perform the actual linear transformation. A new scale is produced that is tied to the locally computed right-hand-side, one for each selected eigenvalue.

5.4 Tunability

The block size, n_{b}, is the only tunable parameter that can have effect on the parallel performance. We have determined that $n_{\text{b}} = 100$, by measuring execution time where we have varied the block size between 50 and 200 in steps on 10, is close to optimal on Kebnekaise. In general, execution time goes down the smaller n_{sel} gets. However, its more beneficial to have several selected eigenvalues in close range rather than having a few very scattered.

6 Numerical experiments

In this section we report on a sequence of experiments performed to test our new parallel algorithm. We give a brief description of the hardware, the test matrices, the experimental methodology before presenting the results. We report on the single threaded execution time, the weak and the strong scaling performance of our current implementation.

6.1 Computer system

All experiments were executed on the machine `Kebnekaise`[1] system at HPC2N, Umeå University. Each compute node contains 28 Intel Xeon E5-2690v4 cores organized in 2 NUMA islands with 14 cores each. The nodes are connected with a FDR Infiniband Network. Each compute core has 32kb 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 as shown in Figure 3.

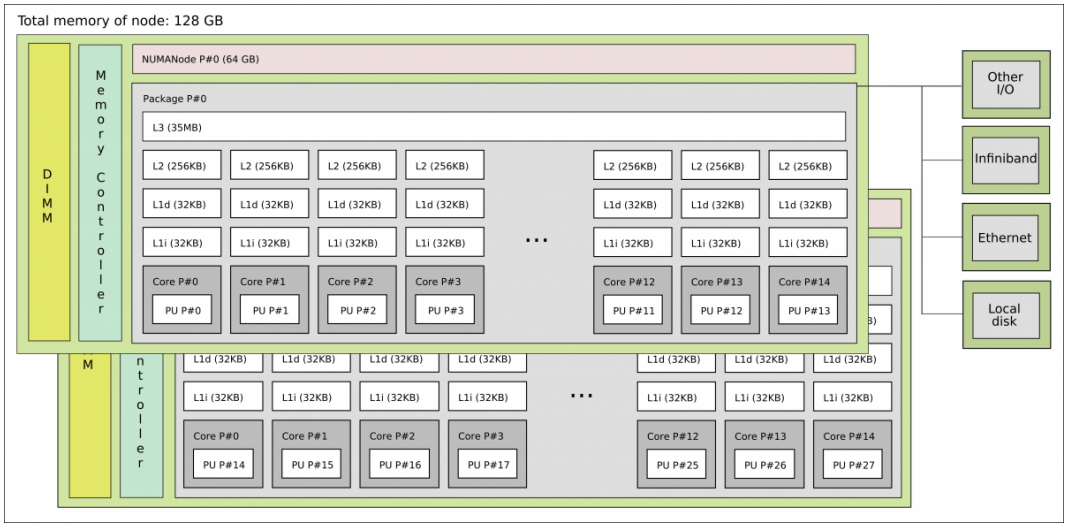

Figure 3: Schematic representation of a compute node on the `Kebnekaise` system.

6.2 Experimental methodology

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

6.3 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.

6.4 Single threaded execution

In Figure 4 our MPI based algorithm is compared against the LAPACK `ztrevc3` solver for

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

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

The single core run-times are quite similar and our MPI version is marginally better. However, the MPI version has significant overhead, even when using a single MPI rank.

[1] `https://www.hpc2n.umu.se/resources/hardware/kebnekaise`

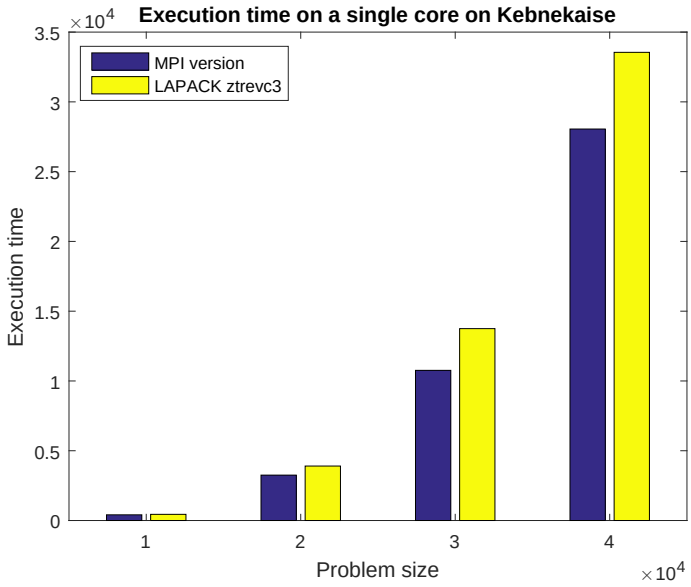

Figure 4: Execution time for our MPI based algorithm and `ztrevc3` from LAPACK for $n \in \{10000, 20000, 30000, 40000\}$.

A serial implementation of the algorithm, without the overhead gives a run-time of 13835 secs 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 T completely before moving on to next the column of T. The the blocking strategy we propose is clearly beneficial as it allows for cache reuse.

6.5 Scalability

The scalability of a program measures its response to an increase in the number of processors. 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

$$\rho = \frac{T_s}{pT_p}, \tag{19}$$

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

6.5.1 Weak scalability

Weak scalability refers to keeping problem size per processor constant as the number of processors is increased.

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 $n_1\sqrt{P_\mathrm{P}/P_1}$.

The results of the weak scalability experiments, with $n_1 = 15000$ and $P_1 = 16$ are shown in Figure 5.

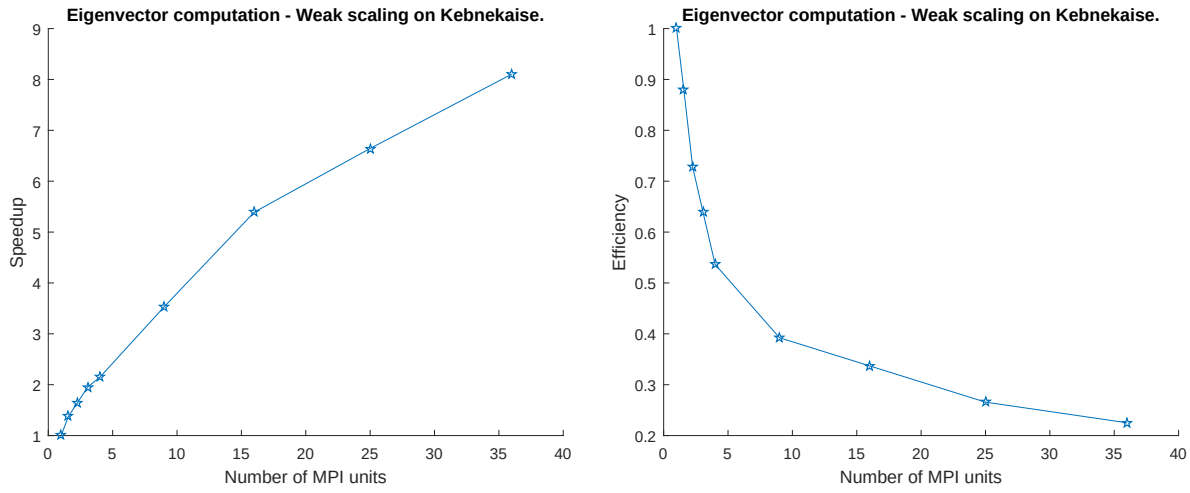

Figure 5: Speedup(left) and efficiency(right) gained when running weak scalability performance test using up to 36 MPI units. One MPI unit consists of 16 MPI ranks.

6.5.2 Strong scalability

Strong scalability refers to keeping the problem size constant as the number of processors is increased. Here we report on strong scalability efficiency. The results of the strong scalability experiments are shown in Figure 6. 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

$$n \in \{10000, 20000, 30000, 40000\} \tag{20}$$

and meshes of size $P_\mathrm{r} \times P_\mathrm{c}$ for

$$P_\mathrm{r} = P_\mathrm{c} \in \{1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 28, 32\} \tag{21}$$

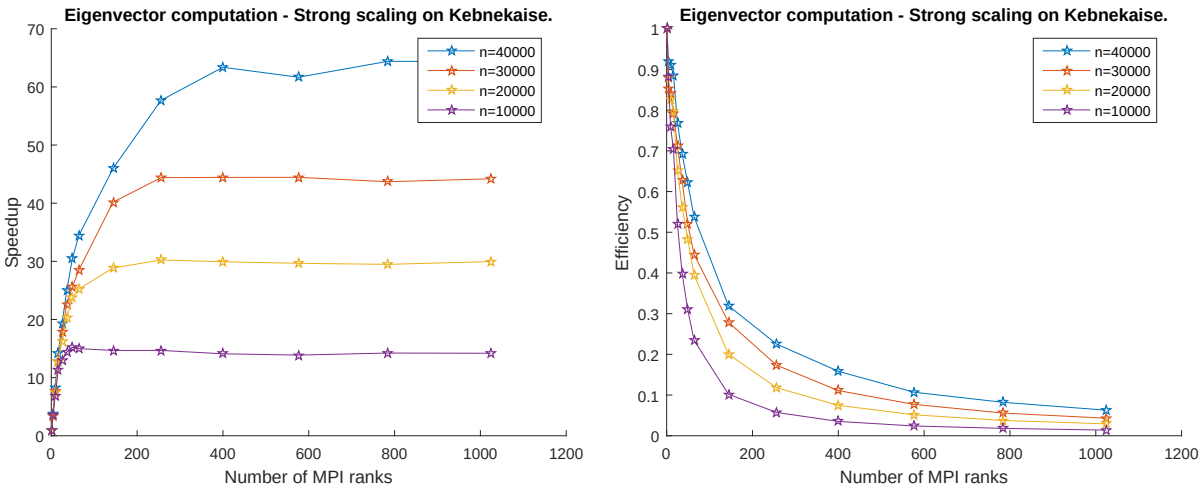

Figure 6: Speedup(left) and efficiency(right) gained when running strong scalability performance test using up to 1024 MPI ranks for four problem sizes.

As expected, 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 tiny with respect to such a large number of cores.

7 Conclusion and future work

In this section we detail the lessons learned and our plans for the next few moths.

We know how to protect against overflow when solving triangular linear systems. Our scalar algorithm is similar to the algorithm implemented in `LAPACK` as `xLATRS`. So far, our main contribution is to simplify and extend the analysis. We have derived a new blocked robust algorithm using the concept of augmented vectors. Our blocked robust algorithm can be parallilized in different settings. Any run-time system which can execute a task-based backward substitution routine can execute a task based robust backward substitution routine. It is simply a matter of replacing the two central kernels with our robust variants which operate on augmented vectors. The parallel overhead imposed by scaling to protect against overflow is minimal. In particular, the number of messages transmitted during the solve is unchanged. Each segment of each right hand side must be augmented with a single word containing the scale factor. After the backward substitution is completed, global communication is required to obtain a global consistent scaling. However, the necessary communication can be amortized over all right hand sides.

We know how to interleave the computation of several eigenvectors. This improves the arithmetic intensity and reduces the solve time. We know how to interleave the computation of the eigenvectors with the gradual back-transform of the eigenvectors of S to eigenvectors of A.

Our MPI implementation incorporates all of these features. However, we are not satisfied with the memory consumption and the parallel performance. We have come to realize that while the standard 2D block cyclic distribution used by `ScaLAPACK` can still be used we need to reorganize the work flow. In particular the following changes will be investigated:

1. Broadcast all eigenvectors to all ranks.

2. Broadcast all diagonal blocks along each process row.

When this information is available, all shifted linear solves can be executed locally. Communication is needed only to update the right hand sides and the partial back-transform. Please refer to Algorithm 12 `BlockSimEigenvectorBacktransform` when considering the accuracy of this claim.

As for the robust solution of triangular linear systems the following questions will be investigated:

1. Complete a standard error analysis for backward substitution with a separate scaling factor for each block row of the solution.

2. What is the exact relationship between overflow in backward substitution routine and the systems condition number?

3. In particular, is it possible to calculate the condition number from the scaling factors?

4. What is the real value of scaling to prevent overflow?

In practice, we can check for overflow between any two points of the code, by investigating a flag set by the runtime system, but tracking down the exact point and column where overflow first occurred appears expensive. Personally, we would prefer to have code which execute to completion without overflowing, returning a list of scaling factors which identifies the location of any problems.

We will also investigate the following related questions

5. How do we implement scalings which are limited to negative powers of 2? Currently it is possible to engineer a situation where the scaling factors underflow. Using 64 bit words will allow us to represent scalings as small as $2^{-(2^{64}-1)}$. The probabilty of underflow in this context is small.

6. How do we apply scalings when dealing with real Schur forms in parallel?

While pursuing an MPI implementation of the algorithm for computing eigenvectors we have also implemented a new task based algorithm for the problem of reordering eigenvalues for matrices in real Schur form. Here we are using the run-time system `StarPU` to execute our code. We expect to capitalize on the lessons learned in this process when transforming our new algorithm for computing eigenvectors to a task based algorithm.

Acknowledgements

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No. 671633. Support has also been received from eSSENCE, a collaborative e-Science programme funded by the Swedish Government via the Swedish Research Council (VR). All experiments have been conducted using the computing resources available at High Performance Computing Center North(HPC2N), Umeå University.

References

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

[2] Carl Christian Kjelgaard Mikkelsen and Lars Karlsson. Robust Solution of Triangular Linear Systems. Technical Report UMINF-17-09, Umeå University, 2017. Also as NLAFET Working Note 9.

Algorithm 13: $X = \texttt{ParaEigVecSolve}(T, U, n, n_\mathrm{b})$

Data: A non singular upper triangular matrix T, a unitary transformation matrix U, problem size n, and a blocking factor n_b. T and U are assumed to be uniformly distributed using a two-dimensional block-cyclic distribution, using n_b as blocking factor in both row and column dimension.

Result: X holds the computed eigenvectors. X is distributed using a one-dimensional block-cyclic distribution with a row block factor n_b, such that row block X_i is stored at process holding block $T_{i,i}$.

`// Calculate the number of blocks`

1 $n_\mathrm{B} \leftarrow \lceil (n/n_\mathrm{b}) \rceil$;

`// Initialize the right-hand-side` W `as the strictly upper part of` T. W `is distributed using a one-dimensional block-cyclic distribution with a row block factor` n_b, `such that row block` W_i `is stored at process holding block` $T_{i,i}$.

2 $W \leftarrow \mathrm{upper}(T)$;

3 **for** $i = n_\mathrm{B}, n_\mathrm{B} - 1, \ldots, 1$ **do**

4 **if** $i \neq n_\mathrm{B}$ **then**

5 `Reduce` W_i to process holding $T_{i,i}$ along current mesh row;

6 Execute $Sol_i \leftarrow \texttt{Solve}(i)$ using $T_{i,i}$ and W_i;

7 **if** $i \neq 1$ **then**

8 `Broadcast` Sol_i along current mesh column;

9 Execute $\texttt{Transform}(i)$ using $U_{*,i}$ and Sol_i;

`// Now perform updates of W, starting at next row block`

10 $CurrUpdRow_k \leftarrow i - 1$;

11 **for** $k \leftarrow i, i+1, \ldots, n_\mathrm{B}$ **do**

12 **if** $CurrUpdRow_k > 0$ **then**

13 Execute $\texttt{Update}(\mathrm{CurrUpdRow}_k, k)$ using $T_{CurrUpdRow_k,k}$ and Sol_k;

14 $CurrUpdRow_k \leftarrow CurrUpdRow_k - 1$;

`// Do more updates, if not involved in next row block solve`

15 **if** *not* $\texttt{Solving}(i-1)$ *and not* $\texttt{Reducing}(i-1)$ **then**

16 **for** $k \leftarrow i, i+1, \ldots, n_\mathrm{B}$ **do**

17 $Cnt \leftarrow 0$;

18 **while** $CurrUpdRow_k > 0$ *and* $Cnt < P_\mathrm{r} - 1$ **do**

19 Execute $\texttt{Update}(CurrUpdRow_k, k)$ using $T_{CurrUpdRow_k,k}$ and Sol_k;

20 $CurrUpdRow_k \leftarrow CurrUpdRow_k - 1$; $Cnt \leftarrow Cnt + 1$;

21 **for** $i = n_\mathrm{B}, n_\mathrm{B} - 1, \ldots, 1$ **do**

22 `Reduce` X_i to process holding $T_{i,i}$ along current mesh row;