Implementing Singular Value and Symmetric/Hermitian Eigenvalue Solvers

Mark Gates
Kadir Akbudak
Mohammed Al Farhan
Ali Charara
Jakub Kurzak
Dalal Sukkari
Asim YarKhan
Jack Dongargara

Innovative Computing Laboratory

December 21, 2021
This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy’s Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation’s exascale computing imperative.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

<table>
<thead>
<tr>
<th>Revision</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>09-2019</td>
<td>first publication</td>
</tr>
<tr>
<td>04-2020</td>
<td>added generalized Hermitian definite eigenvalues (Section 2.3) and eigenvectors (Section 2.5)</td>
</tr>
<tr>
<td>12-2021</td>
<td>added divide and conquer (Chapter 3) and optimization (Chapter 4)</td>
</tr>
</tbody>
</table>

@techreport{gates2019implementing,  
author={Gates, Mark and Akbudak, Kadir and Al Farhan, Mohammed and Charara, Ali and Kurzak, Jakub and Sukkari, Dalal and YarKhan, Asim and Dongarra, Jack},  
title={{SLATE} Working Note 13: Implementing Singular Value and Symmetric/Hermitian Eigenvalue Solvers},  
institution={Innovative Computing Laboratory, University of Tennessee},  
year={2021},  
month={December},  
number={ICL-UT-19-07},  
ote=revision 12-2021}
# Contents

List of Figures

## 1 Introduction
1.1 Significance of SLATE ........................................... 1
1.2 Design of SLATE ............................................. 3

## 2 Implementation
2.1 Singular Value Decomposition ............................... 6
2.2 Hermitian Eigenvalue Problem ............................ 7
2.3 Generalized Hermitian Definite Eigenvalue Problem .... 7
2.4 Three Stage Algorithms ................................. 8
2.5 Eigenvector Computation .................................. 13
  2.5.1 Eigenvectors of tridiagonal matrix .................. 13
  2.5.2 Second stage back-transformation .................... 14
  2.5.3 First stage back-transformation ....................... 15

## 3 Divide and conquer
3.1 Cuppen's method ........................................... 17
  3.1.1 Choice of $\theta$ ........................................ 18
  3.1.2 Secular equation .................................... 18
  3.1.3 Deflation ............................................. 19
  3.1.4 Back-transformation .................................. 20
  3.1.5 Deflation Example .................................... 22
  3.1.6 Eigenvectors via Löwner Theorem ................. 27
  3.1.7 Cost .................................................. 29
3.2 Routines .................................................. 30
  3.2.1 stedc ................................................ 30
  3.2.2 stedc_solve ........................................ 31
3.2.3 stedc_merge ........................................... 32
3.2.4 stedc_z_vector ................................... 33
3.2.5 stedc_deflate ...................................... 34
3.2.6 stedc secular ...................................... 36
3.2.7 stedc_sort .......................................... 37

4 Optimization ......................................... 39
  4.1 Hermitian to Hermitian band reduction (he2hb) ....................... 39
  4.2 Back-transformation (unmtr_hb2st) .................................. 42

5 Performance ........................................... 43
  5.1 Environment ......................................... 43
    5.1.1 Hardware ......................................... 43
    5.1.2 Software ......................................... 43
  5.2 Results ............................................ 44

Bibliography ........................................... 47
### List of Figures

1.1 Dependencies of ECP applications on dense linear algebra software. ........................................ 2  
1.2 SLATE in the ECP software stack. ................................................................. 3  
1.3 Code size comparison - ScaLAPACK vs SLATE ...................................................... 5  
2.1 Three stage Hermitian eigenvalue and SVD algorithms. .................................................. 11  
2.2 One panel of the first stage reduction to band form. ...................................................... 11  
2.3 Bulge-chasing algorithm. .............................................................................................. 12  
2.4 Hermitian bulge-chasing algorithm. .................................................................................. 12  
2.5 Redistribute 1D block row cyclic distributed matrix using $4 \times 1$ grid into a 2D block cyclic distribution using $2 \times 2$ grid. ................................................................. 14  
2.6 Second stage back transformation, with $V$ block size $j_b = 3$ vectors. Block reflector 3 is highlighted to show overlap. ...................................................................................... 15  
2.7 Dependencies allow up to $\lceil m_t^2 \rceil$ parallel tasks. ...................................................... 16  
4.1 Performance of $he2hb$ Using 1 and 2 nodes on Summit, $1 \times 1$ and $2 \times 2$ process grids. 40  
4.2 Overlap the panel factorization with subsequent data movements. ................................. 40  
4.3 The performance impact by overlapping the panel factorization with subsequent data movements. ......................................................................................................................... 41  
4.4 Performance results of Hermitian to Hermitian band reduction, using 1 node, $1 \times 1$ process grid. nb = 128,320, ib= 16, 48 for CPU, GPU tests, panel-threads=10 ...................................... 41  
4.5 Performance of unmtr$_{hb2st}$ on a host with two 20-core Intel Broadwell Xeon E5-2698 v4 CPUs and one NVIDIA V100 activated. $N=16384$. As seen in the figure, the device implementation provides up to 6x speedup. ............................................................ 42  
5.1 Summit node architecture. .............................................................................................. 44  
5.2 SVD performance comparison. ........................................................................................ 45  
5.3 Generalized to standard eigenvalue performance comparison. ........................................ 46  
5.4 Profile of eigenvalue solver implementations showing each phase for $N=12288$. One node of Summit is used. ............................................................................................................. 46
CHAPTER 1

Introduction

1.1 Significance of SLATE

Software for Linear Algebra Targeting Exascale (SLATE) 1 is being developed as part of the Exascale Computing Project (ECP) 2, which is a joint project of the U.S. Department of Energy’s Office of Science and National Nuclear Security Administration (NNSA). SLATE will deliver fundamental dense linear algebra capabilities for current and upcoming distributed-memory systems, including GPU-accelerated systems as well as more traditional multi core–only systems.

SLATE will provide coverage of existing LAPACK and ScaLAPACK functionality, including parallel implementations of Basic Linear Algebra Subroutines (BLAS), linear systems solvers, least squares solvers, and singular value and eigenvalue solvers. In this respect, SLATE will serve as a replacement for LAPACK and ScaLAPACK, which, after two decades of operation, cannot be adequately retrofitted for modern, GPU-accelerated architectures.

Figure 1.1 shows how heavily ECP applications depend on dense linear algebra software. A direct dependency means that the application’s source code contains calls to the library’s routines. An indirect dependency means that the applications needs to be linked with the library due to another component depending on it. Out of 60 ECP applications, 38 depend on BLAS – either directly or indirectly – 40 depend on LAPACK, and 14 depend on ScaLAPACK. In other words, the use of dense linear algebra software is ubiquitous among the ECP applications.

---

1 http://icl.utk.edu/slate/
2 https://www.exascaleproject.org
Figure 1.1: Dependencies of ECP applications on dense linear algebra software.
1.2 Design of SLATE

SLATE is built on top of standards, such as MPI and OpenMP and de facto standard industry solutions such as NVIDIA CUDA and AMD HIP. SLATE also relies on high performance implementations of numerical kernels from vendor libraries, such as Intel MKL, IBM ESSL, NVIDIA cuBLAS, and AMD rocBLAS. SLATE interacts with these libraries through a layer of C++ APIs. Figure 1.2 shows SLATE’s position in the ECP software stack.

![Figure 1.2: SLATE in the ECP software stack.](image)

The following paragraphs outline the foundations of SLATE’s design.

**Object-Oriented Design:** The design of SLATE revolves around the Tile class and the Matrix class hierarchy. The Tile class is intended as a simple class for maintaining the properties of individual tiles and implementing core serial tile operations, such as tile BLAS, while the Matrix class hierarchy maintains the state of distributed matrices throughout the execution of parallel matrix algorithms in a distributed-memory environment. Currently, the classes are structured as follows:

- **BaseMatrix** is an abstract base class for all matrices.
- **Matrix** represents a general \( m \times n \) matrix.
- **BaseTrapezoidMatrix** is an abstract base class for all upper-trapezoid or lower-trapezoid, \( m \times n \) matrices. For upper matrices, tiles \( A(i, j) \) are stored for \( i \leq j \). For lower matrices, tiles \( A(i, j) \) are stored for \( i \geq j \).
- **TrapezoidMatrix** represents an upper-trapezoid or a lower-trapezoid, \( m \times n \) matrix. The opposite triangle is implicitly zero.
- **TriangularMatrix** represents an upper-triangular or a lower-triangular, \( n \times n \) matrix.
- **SymmetricMatrix** represents a symmetric, \( n \times n \) matrix, with only the upper or lower triangle stored. The opposite triangle is implicitly known by symmetry \( (A_{j,i} = A_{i,j}) \).
- **HermitianMatrix** represents a Hermitian, \( n \times n \) matrix, with only the upper or lower triangle stored. The opposite triangle is implicitly known by symmetry \( (A_{j,i} = \bar{A}_{i,j}) \).
1.2. DESIGN OF SLATE

**Tiled Matrix Layout:** The new matrix storage introduced in SLATE is one of its most impactful features. In this respect, SLATE represents a radical departure from other distributed linear algebra software such as ScaLAPACK or Elemental, where the local matrix occupies a contiguous memory region on each process. In contrast, tiles are first class objects in SLATE that can be individually allocated and passed to low-level tile routines. In SLATE, the matrix consists of a collection of individual tiles with no correlation between their positions in the matrix and their memory locations. At the same time, SLATE also supports tiles pointing to data in a traditional ScaLAPACK matrix layout, thereby easing an application’s transition from ScaLAPACK to SLATE.

**Handling of side, uplo, trans:** The classical BLAS takes parameters such as `side`, `uplo`, `trans` (named “op” in SLATE), and `diag` to specify operation variants. Traditionally, this has meant that implementations have numerous cases. The reference BLAS has nine cases in `zgemm` and eight cases in `ztrmm` (times several sub-cases). ScaLAPACK and PLASMA likewise have eight cases in `ztrmm`. In contrast, by storing both `uplo` and `op` within the matrix object itself, and supporting inexpensive shallow copy transposition, SLATE can implement just one or two cases and map all the other cases to that implementation by appropriate transpositions. For instance, SLATE only implements one case for `gemm` (NoTrans, NoTrans) and handles all other cases by swapping indices of tiles and setting `trans` appropriately for the underlying tile operations.

**Templating of Precisions:** SLATE handles multiple precisions by C++ templating, so there is only one precision-independent version of the code, which is then instantiated for the desired precisions. Operations are defined so that they can be applied consistently across all precisions. SLATE’s BLAS++ component provides overloaded, precision-independent wrappers for all underlying, node-level BLAS, and SLATE’s PBLAS are built on top of these. Currently, the SLATE library has explicit instantiations of the four main data types: `float`, `double`, `std::complex<float>`, and `std::complex<double>`. The SLATE code should be able to accommodate other data types, such as half, double-double, or quad precision, given appropriate underlying node-level BLAS.

**Templating of Execution Targets:** Parallelism is expressed in SLATE’s computational routines. Each computational routine solves a sub-problem, such as computing an LU factorization (`getrf`) or solving a linear system given an LU factorization (`getrs`). In SLATE, these routines are templated for different targets (CPU or GPU), with the code typically independent of the target. The user can choose among various target implementations:

- **Target::HostTask** means multithreaded execution by a set of OpenMP tasks.
- **Target::HostNest** means multithreaded execution by a nested “parallel for” loop.
- **Target::HostBatch** means multithreaded execution by calling a batched BLAS routine.
- **Target::Devices** means (multi-)GPU execution using calls to batched BLAS.

**MPI Communication:** Communication in SLATE relies on explicit dataflow information. When a tile is needed for computation, it is broadcast to all the processes where it is required. Rather than explicitly listing MPI ranks, the broadcast is expressed in terms of the destination (sub)matrix to be updated. This way, SLATE’s messaging layer is oblivious to the mapping of tiles to processes. Also, multiple broadcasts are aggregated to allow for pipelining of MPI messages with transfers between the host and the devices. Since the set
of processes involved in a broadcast is determined dynamically, the use of MPI collectives is not ideal, as it would require setting up a new subcommunicator for each broadcast. Instead, SLATE uses point-to-point MPI communication following a hypercube pattern to broadcast the data.

**Node-Level Coherency:** For offload to GPU accelerators, SLATE implements a memory consistency model, inspired by the MOSI cache coherency protocol [1, 2], on a tile-by-tile basis. For read-only access, tiles are mirrored in the memories of, possibly multiple, GPU devices and deleted when no longer needed. For write access, tiles are migrated to the GPU memory and returned to the CPU memory afterwards if needed. A tile’s instance can be in one of three states: Modified, Shared, or Invalid. Additional flag OnHold can be set along any state, as follows:

- **Modified (M)** indicates that the tile’s data is modified. Other instances should be Invalid. The instance cannot be purged.
- **Shared (S)** indicates that the tile’s data is up-to-date. Other instances may be Shared or Invalid. The instance may be purged unless it is on hold.
- **Invalid (I)** indicates that the tile’s data is obsolete. Other instances may be Modified, Shared, or Invalid. The instance may be purged unless it is on hold.
- **OnHold (O)** is a flag orthogonal to the other three states that indicates a hold is set on the tile instance, and the instance cannot be purged until the hold is released.

**Dynamic Scheduling:** Dataflow scheduling (omp task depend) is used to execute a task graph with nodes corresponding to large blocks of the matrix. Dependencies are tracked using dummy vectors, where each element represents a block of the matrix, rather than the matrix data itself. For multi-core execution, each large block is dispatched to multiple cores—using either nested tasking (omp task) or batched BLAS. For GPU execution, calls to batched BLAS are used specifically to deliver fast processing of matrix blocks that are represented as large collections of tiles.

One of the main benefits of SLATE’s architecture is dramatic reduction in the size of the source code, compared to ScaLAPACK (Figure 1.3). As of August 2019, with more than two thirds of ScaLAPACK’s functionality covered, SLATE’s source code is $8 \times$ to $9 \times$ smaller than ScaLAPACK’s.

![Figure 1.3: Code size comparison - ScaLAPACK vs SLATE (numbers from August 2019).](image)
CHAPTER 2

Implementation

2.1 Singular Value Decomposition

In linear algebra, the singular value decomposition (SVD) is a factorization of a real or complex matrix $A$ of the form $U \Sigma V^H$, where $U$ is an $m \times m$ real or complex unitary matrix, $\Sigma$ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, and $V$ is $n \times n$ real or complex unitary matrix. The diagonal entries $\sigma_i$ of $\Sigma$ are known as the singular values of $A$. The columns of $U$ and the columns of $V$ are known as the left-singular vectors and the right-singular vectors of $A$, respectively. Typically the values $\sigma_i$ are ordered such that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(m,n)} \geq 0$. Typically, only the first $\min(m,n)$ columns of $U$ and rows of $V$ are computed, yielding the “reduced” or “economy-size” SVD, since the remaining columns of $U$ and rows of $V$ are multiplied by the zero part of $\Sigma$ and do not contribute to $A$.

The SVD is the generalization of the eigendecomposition of a positive semidefinite normal matrix to any $m \times n$ matrix via an extension of the polar decomposition. The SVD is related to the eigendecomposition in the following way. The singular values are the square roots of the eigenvalues of $A^T A$, the columns of $V$ are the corresponding eigenvectors, and the columns of $U$ are the eigenvectors of $A A^T$.

The discovery of the SVD is attributed to four famous mathematicians, who seem to have come across it independently: Eugenio Beltrami (in 1873), Camille Jordan (in 1874), James Joseph Sylvester (in 1889), Léon César Autonne (in 1915). The first proof of the singular value decomposition for rectangular and complex matrices seems to be by Carl Eckart and Gale J. Young in 1936 [3].

First practical methods for computing the SVD are attributed to Kogbetliantz and Hestenes [4] and resemble closely the Jacobi eigenvalue algorithm, which uses Jacobi (Givens) plane rotations.
These were replaced by the method of Golub and Kahan [5], which uses Householder reflections to reduce to bidiagonal, then plane rotations to continue the reduction to diagonal. The most popular algorithm used today is the variant of the Golub/Kahan algorithm published by Golub and Reinsch [6].

2.2 Hermitian Eigenvalue Problem

In linear algebra, an eigendecomposition or spectral decomposition is the factorization of a matrix into a canonical form, where the matrix is represented in terms of its eigenvalues and eigenvectors. An eigenvector or characteristic vector of a linear transformation is a nonzero vector that changes by a scalar factor when that linear transformation is applied to it. That is, a (non-zero) vector $x$ of dimension $n$ is an eigenvector of a square $n \times n$ matrix $A$ if it satisfies the linear equation $Ax = \lambda x$. In other words, the eigenvectors are the vectors that the linear transformation $A$ merely elongates or shrinks, and the amount that they elongate/shrink by is the eigenvalue.

A square $n \times n$ matrix $A$ with $n$ linearly independent eigenvectors $q_i$ (where $i = 1, ..., n$) can be factored as $A = X\Lambda X^{-1}$ where $X$ is the square $n \times n$ matrix whose $i$th column is the eigenvector $x_i$ of $A$, and $\Lambda$ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, $\Lambda_{ii} = \lambda_i$. Only diagonalizable matrices can be factorized in this way.

Any Hermitian matrix can be diagonalized by a unitary matrix, and the resulting diagonal matrix has only real entries. This implies that all eigenvalues of a Hermitian matrix $A$ with dimension $n$ are real, and that $A$ has $n$ linearly independent eigenvectors. Moreover, a Hermitian matrix has orthogonal eigenvectors for distinct eigenvalues. Given that conjugate transpose of a unitary matrix is also its inverse, the Hermitian eigenvalue problem boils down to $A = X\Lambda X^H$. This means $A = X\Lambda X^T$ in the case of real symmetric matrices.

Historically, eigenvalues arose in the study of quadratic forms and differential equations. The initial discoveries are attributed to Euler, Lagrange, and Cauchy. The list of mathematicians who contributed to the field includes such famous names as Fourier, Sturm, Hermite, Brioschi, Clebsch, Weierstrass, Liouville, Schwarz, and Poincaré. Generally, Hilbert is credited with using the German word eigen, which means “own”, to denote eigenvalues and eigenvectors, though he may have been following a related usage by Helmholtz.

The first numerical algorithm for computing eigenvalues and eigenvectors appeared in 1929, when Von Mises published the power method. One of the most popular methods today, the QR algorithm, was proposed independently by Francis [7] and Kublanovskaya [8] in 1961.

2.3 Generalized Hermitian Definite Eigenvalue Problem

The generalized Hermitian definite eigenvalue problem has various types:

- Type 1: $Az = \lambda Bz$,
- Type 2: $ABz = \lambda z$,
- Type 3: $BAz = \lambda z$,
where $A$ is Hermitian and $B$ is Hermitian positive-definite.

To solve it, we first reduce it to the standard eigenvalue form, $\hat{A}x = \lambda x$. The reductions for types (2) and (3) are identical; they differ in the back-transformation. First, form the Cholesky factorization of $B$ as either $B = LL^H$ with lower triangular $L$, or $B = U^HU$ with upper triangular $U$. Then form $\hat{A}$, which overwrites $A$, as:

- Type 1: compute $\hat{A} = L^{-1}AL^{-H}$ or $\hat{A} = U^{-H}AU^{-1}$, as shown in Algorithm 1;
- Type 2 or 3: compute $\hat{A} = L^HAL$ or $\hat{A} = UAU^H$, as shown in Algorithm 2.

Only the lower or upper triangles of $A$, $\hat{A}$, and $B$ are stored and computed on, the opposite triangle being known from symmetry. The \texttt{hegst} routine (Hermitian generalized to standard) takes $A$ and the Cholesky factor $L$ or $U$ of $B$ as input; the lower or upper triangle of $\hat{A}$ overwrites the lower or upper triangle of $A$ on output.

After solving the standard eigenvalue problem, $\hat{A}x = \lambda x$, an eigenvector $x$ is back-transformed to be an eigenvector $z$ of the generalized eigenvalue problem as follows:

- Type 1 or 2: $z = L^{-H}x$ or $z = U^{-1}x$ using \texttt{trsm};
- Type 3: $z = Lx$ or $z = U^Hx$ using \texttt{trmm}.

## 2.4 Three Stage Algorithms

We solve both the SVD and the Hermitian eigenvalue problem by a three stage algorithm, shown in Figure 2.1:

1. First stage reduction from full to triangular band (SVD) or Hermitian band (eigenvalue) form, which uses Level 3 BLAS.
2. Second stage reduction band to real bidiagonal (SVD) or real symmetric tridiagonal (eigenvalue) form. This uses a bulge chasing algorithm.
3. Third stage reduction to diagonal form, revealing the singular values or eigenvalues. Currently we use QR iteration, but could also use divide and conquer, MRRR, bisection, or other solver.

This is in contrast to the traditional algorithm used in LAPACK and ScaLAPACK that goes directly from full to bidiagonal or symmetric tridiagonal, which uses Level 2 BLAS and is memory-bandwidth limited. If $m \gg n$ (or $m \ll n$), the SVD has an optional initial reduction from tall (or wide) to square, using a QR (or LQ) factorization.

For the SVD, the first stage proceeds by computing a QR factorization of a block column to annihilate entries below the diagonal, and updating the trailing matrix, as shown in Figure 2.2. It then computes an LQ factorization of a block row to annihilate entries right of the upper bandwidth, and updates the trailing matrix. It repeats factoring block columns and block rows, until the entire matrix is brought to band form. The width of the block columns and block rows is the resulting matrix bandwidth, $n_b$. 
Algorithm 1 Reduction to standard form (type 1) pseudocode.

```plaintext
1: function hegst(type, A, B)
2:     for k = 1, ..., nt // nt = number of block rows in A.
3:         // A(k,k) = B(k,k)^(-1) * A(k,k) * B(k,k)^(-H).
4:         hegst(type, A(k,k), B(k,k))
5:     for m = k + 1, ..., nt
6:         trsm(B(k,k), A(m,k))
7:     end
8:     // A(k + 1 : nt,k) = [B(k + 1 : nt,k) * A(k,k)] + A(k + 1 : nt,k).
9:     for m = k + 1, ..., nt
10:        hemm(A(k,k), B(m,k), A(m,k))
11:     end
12:     // A(k + 1 : nt,k+1 : nt) = [A(k + 1 : nt,k) * B(k+1 : nt,k)] + A(k+1 : nt,k+1 : nt).
13:     for m = k + 1, ..., nt
14:         for n = k + 1, ..., nt
15:            her2k(A(m,k), B(m,k), A(m,n))
16:         end
17:     end
18:     // A(k + 1 : nt,k) = B(k+1 : nt,k+1 : nt) * A(k + 1 : nt,k).
19:     for m = k + 1, ..., nt
20:        trsm(B(m,n), A(m,k))
21:     end
22:     end
23:     return A
24: end function
```
Algorithm 2 Reduction to standard form (type 2 or 3) pseudocode.

1: function hegst(type, A, B)
2:     for $k = 1, \ldots, nt$ // $nt =$ number of block rows in $A$
3:         // $A(k, 1 : k) = [A(k, 1 : k) * B(1 : k, 1 : k)]$.
4:         for $m = 1, \ldots, k$
5:             for $n = 1, \ldots, k$
6:                 trmm($B(m, n)$, $A(k, m)$)
7:         end
8:     end
9:     // $A(k, 1 : k) = [A(k, k) * B(k, 1 : k)] + A(k, 1 : k)$.
10:    for $m = 1, \ldots, k$
11:        hemm($A(k, k)$, $B(k, m)$, $A(k, m)$)
12:    end
13:    // $A(1 : k, 1 : k) = [A(1 : k : 1)H * B(1 : kH)] + A(1 : k, 1 : k)$.
14:    for $m = 1, \ldots, k$
15:        for $n = 1, \ldots, k$
16:            her2k($A(k, m)$, $B(k, m)$, $A(m, n)$)
17:        end
18:    end
19:    // $A(k, 1 : k) = [A(k, k) * B(k, 1 : k)] + A(k, 1 : k)$.
20:    for $m = 1, \ldots, k$
21:        hemm($A(k, k)$, $B(k, m)$, $A(k, m)$)
22:    end
23:    // $A(k, 1 : k) = [B(k, k)H * A(k, 1 : k)]$.
24:    for $m = 1, \ldots, k$
25:        trmm($B(k, k)$, $A(k, m)$)
26:    end
27:    // $A(k, k) = B(k, k)H * A(k, k) * B(k, k)$.
28:    hegst(type, $A(k, k)$, $B(k, k)$)
29: end
30: return $A$
31: end function
2.4. THREE STAGE ALGORITHMS

CHAPTER 2. IMPLEMENTATION

Figure 2.1: Three stage Hermitian eigenvalue and SVD algorithms.
Three stage Hermitian eigenvalue (top) and SVD (bottom) algorithms.

Figure 2.2: One panel of the first stage reduction to band form.
The second stage reduces the band form to the final bidiagonal form using a bulge chasing technique. It involves $6n_{\text{b}}n^2$ operations, so it takes a small percentage of the total operations, which decreases with $n$. The operations are memory bound, but are fused together as Level 2.5 BLAS [9] for cache efficiency. We designed the algorithm to use fine-grained, memory-aware tasks in an out-of-order, data-flow task-scheduling technique that enhances data locality [10, 11].

The second stage proceeds in a series of sweeps, each sweep bringing one row to bidiagonal and chasing the created fill-in elements down to the bottom right of the matrix using successive orthogonal transformations. It uses three kernels. Kernel 1 (yellow task $T_{1,1}$ in Figure 2.3b) applies a Householder reflector from the right (indicated by the down arrow) to eliminate a row right of the superdiagonal, which also creates a bulge of fill-in beneath the diagonal. It then applies a Householder reflector from the left (indicated by the right arrow) to eliminate the first column of the bulge below the diagonal, and applies the update to the first block column only. The remainder of the bulge is not eliminated, but is instead left for subsequent sweeps to eliminate, as they would reintroduce the same nonzeros.

Kernel 2 (blue task $T_{1,2}$) continues to apply the left Householder reflector from kernel 1 (or kernel 3) to the next block column, creating a bulge above the upper bandwidth. It then applies a right Householder reflector to eliminate the first row of the bulge right of the upper bandwidth,
Kernel 3 (red task $T_{1,3}$) continues to apply the right Householder reflector from kernel 2, creating a bulge below the main diagonal. As in kernel 1, it then applies a left Householder reflector to eliminate the first column of the bulge below the diagonal and updates just the current block column. After kernel 3, kernel 2 is called again (blue task $T_{1,4}$) to continue application of the left Householder reflector in the next block column. A sweep consists of calling kernel 1 to bring a row to bidiagonal, followed by repeated calls to kernels 2 and 3 to eliminate the first column or row of the resulting bulges, until the bulges are chased off the bottom-right of the matrix.

For parallelism, once a sweep has finished the first kernel 3, a new sweep can start in parallel. This new sweep is shifted over one column and down one row, as shown in Figure 2.3c. Before task $i$ in sweep $s$, denoted as $T_{s,i}$, can start, it depends on task $T_{s-1,i+3}$ in the previous sweep being finished, to ensure that kernels do not update the same entries simultaneously. To maximize cache reuse, tasks are assigned to cores based on their data location. Ideally, the band matrix fits into the cores’ combined caches, and each sweep cycles through the cores as it progresses down the band.

For the Hermitian eigenvalue problem, the second stage shown in Figure 2.4 is very similar to the SVD second stage. Where the SVD has different reflectors from the right and left, here the same reflector is applied from the left and the right. Symmetry is taken into account, so only entries in the lower triangle are computed, while entries in the upper triangle are known by symmetry.

### 2.5 Eigenvector Computation

The three stage Hermitian approach to solve the eigenvalue problem of a dense matrix is to first reduce it to Hermitian band matrix form, $A = Q_1 B Q_1^H$ using Householder reflectors, then reduce the banded matrix further into a real symmetric tridiagonal matrix $B = Q_2 T Q_2^H$, finally, compute the eigenpairs of the tridiagonal matrix using an iterative method such as QR iteration, or the recursive approach of divide-and-conquer, such that $T = Q_3 \Lambda Q_3^H$. The subsequent eigenvectors are then accumulated during the back transformation phase, i.e., $X = Q_1 Q_2 Q_3$ to calculate the eigenvectors $X$ of the original matrix $A$.

#### 2.5.1 Eigenvectors of tridiagonal matrix

Once the tridiagonal reduction is achieved, the implicit QR eigensolver $\text{steqr2}$ calculates the eigenvalues and optionally its associated eigenvectors of the condensed matrix structure. In SLATE (and ScaLAPACK), the $\text{steqr2}$ is a modified version of the LAPACK routine $\text{steqr}$ which allows each process to perform updates on the distributed matrix $Q_2$, and achieve parallelization during this step.

Algorithm 3 shows the call to the tridiagonal eigensolver $\text{steqr2}$. First, a matrix to store the eigenvectors $Q_{3,1D}$ of the tridiagonal matrix $T$ is created using a 1D block row cyclic with a $n_p \times 1$ process grid, where $n_p$ is the number of MPI processes. Then each process updates up to $(n/n_b)/n_p$ rows of the matrix $Q_{3,1D}$, where $n$ is the matrix size and $n_b$ is the block size used to distribute the rows of $Q_{3,1D}$. Finally, the matrix of the eigenvectors is redistributed to a 2D...
2.5. EIGENVECTOR COMPUTATION

Figure 2.5: Redistribute 1D block row cyclic distributed matrix using $4 \times 1$ grid into a 2D block cyclic distribution using $2 \times 2$ grid.

block cyclic distribution as illustrated in Figure 2.5.

**Algorithm 3** Tridiagonal Eigensolver using steqr2 pseudocode.

```plaintext
function steqr2($T$, $Q_3$

// The “1D block row cyclic” grid configuration
$1D = n_p \times 1$
// Compute the number of rows owned by each processor
$nrc = (n/n_b)/n_p$
// Build SLATE matrix $Q_{3,1D}$ using the 1-dim grid
$Q_{3,1D} = \text{Matrix}(nrc, n_b, n_p, 1)$
// Call steqr2 to compute the eigenpairs of the tridiagonal matrix
$(Q_{3,1D}A Q_{3,1D}^H) = \text{steqr2}($
// The “2D block cyclic” grid configuration
$2D = p \times q$
// Redistribute the 1-dim eigenvector matrix into 2-dim matrix
$Q_3 = \text{redistribute}(Q_{3,1D})$
end function
```

2.5.2 Second stage back-transformation

The second stage back-transformation multiplies the vectors $Q_3$ by $Q_2$ from the second stage reduction from band to tridiagonal form (“bulge chasing”), to form $Q_2Q_3$. SLATE uses a distributed version of the scheme developed by [12]. The Householder vectors generated during the bulge chasing (Figure 2.4) are stored in a matrix $V$, shown in Figure 2.6. Conceptually, the vectors from each sweep $i$ are stored in column $i$ of the lower triangular matrix $V$. The vectors are blocked together into parallelograms, as shown in Figure 2.6b, to form block Householder reflectors, $H_r = I - V_r T_r V^H_r$ where $V_r$ is the $r$th block of $V$, using the compact WY format [13]. Thus $Q_2 = H_k \cdots H_2 H_1$. Application of these $H_r$ overlap, illustrated in Figure 2.6c, creating the dependencies between them shown in Figure 2.6b. These dependencies allow up to $\lceil \frac{mt}{2} \rceil$ updates to occur in parallel. Figure 2.7 shows these blocks and the corresponding tasks for a $10 \times 10$ block matrix. For instance, all four dark blue tasks update different rows of $Q_3$ and so can run in parallel. Using the OpenMP task scheduler makes taking advantage of this parallelism very easy.
2.5. EIGENVECTOR COMPUTATION

2.5.3 First stage back-transformation

The first stage back-transformation multiplies the vectors \((Q_2Q_3)\) by \(Q_1\) from the first stage reduction to band, to form \(X = Q_1(Q_2Q_3)\). The routine \texttt{unmtr\_he2hb} applies \(Q_1\) or \(Q_1^H\) on the left or right of a matrix \(C\), which is then overwritten by \(Q_1C\), \(Q_1^HC\), \(CQ_1\), or \(CQ_1^H\). For eigenvectors, we need only the left, no-transpose case with \(C = Q_2Q_3\), to form the eigenvectors \(X = Q_1(Q_2Q_3)\). It is essentially identical to applying \(Q\) from a QR factorization, but shifted by one block-row since we reduced to band form instead of triangular form, as in QR. Thus, as in LAPACK, we can leverage the existing \texttt{unmqr} routine that applies \(Q\) from a QR factorization.
2.5. EIGENVECTOR COMPUTATION

![Diagram](image)

(a) Blocks of vectors, colored by independent blocks.
(b) Simulated run showing task parallelism.

**Figure 2.7:** Dependencies allow up to $\left\lceil \frac{mt}{2} \right\rceil$ parallel tasks.

**Algorithm 4** `unmtr hb2st` back-transformation pseudocode. Indices are block rows/cols.

```plaintext
function unmtr_hb2st(V, C)
    // C is mt x nt block rows/cols, blocksize nb x nb
    // V is mt(mt + 1)/2 blocks, blocksize 2nb x nb
    for j = mt - 1 to 0
        for i = j to mt - 1
            task depend in, out on row[i] and row[i + 1]
            r = i - j + j : mt - j(j - 1)/2
            Broadcast $V_r$
            Compute $T$ from $V_r$ (larft)
            $D = V_r^T$ (gemm or trmm)
            parallel for k = 0 to nt - 1
                if $C_{i:i+1,k}$ are local then
                    // Compute QC = $(I - VTV^H)C$
                    $W = V_r^H C_{i:i+1,k}$
                    $C_{i:i+1,k} = C_{i:i+1,k} - DW$
                end
            end
            end task
        end
    end
end function
```
CHAPTER 3

Divide and conquer

The solution of the tridiagonal eigenvalue problem can use several different methods: QR iteration, divide and conquer, bisection, or MRRR. Here we derive the divide and conquer algorithm proposed by Cuppen [14], which is significantly faster than QR iteration. Our derivation largely follows that of Tisseur and Dongarra [15], with clarifications and notes related to the implementation in SLATE.

3.1 Cuppen’s method

Define tridiagonal matrix $T \in \mathbb{R}^{n \times n}$ with eigenvalue decomposition $T = W \Lambda W^T$ where $\Lambda$ is diagonal and $W$ is orthogonal. Split $T$ into the rank-1 update

$$T = \begin{bmatrix} \tilde{T}_1 & \beta e_k e_k^T \\ \beta e_k e_k^T & T_2 \end{bmatrix} = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + \rho vv^T \quad \text{with} \quad v = \begin{bmatrix} e_k \\ \theta^{-1} e_1 \end{bmatrix}. \quad (3.1)$$

where $T_1$ is $n_1 \times n_1$, $T_2$ is $n_2 \times n_2$, and $\rho = \theta \beta$ (see Section 3.1.1 about choice of $\theta$). $\tilde{T}_1$ and $T_1$ differ in only the bottom-right element, and $\tilde{T}_2$ and $T_2$ differ in only the top-left element:

$$(T_1)_{n_1,n_1} = (\tilde{T}_1)_{n_1,n_1} - \rho,$$

$$(T_2)_{1,1} = (\tilde{T}_2)_{1,1} - \rho.$$

Recursively solve eigen decompositions for $T_1$ and $T_2$, yielding

$$T_1 = Q_1 D_1 Q_1^T,$$

$$T_2 = Q_2 D_2 Q_2^T. \quad (3.3)$$
3.1. CUPPEN’S METHOD  

CHAPTER 3. DIVIDE AND CONQUER

Then the eigen decomposition of $T$ is given by

$$T = Q(D + \rho z z^T)Q^T \quad \text{with} \quad Q = \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix}, \quad D = \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix}, \quad z = Q^Tv.$$ 

From the definition of $v$, $z$ is the last row of $Q_1$, and $\theta^{-1}$ times the first row of $Q_2$. Given Equation (3.2), compute the eigen decomposition

$$D + \rho z z^T = U \Lambda U^T$$

as shown in Section 3.1.2. Finally, the eigen decomposition of $T$ is

$$T = Q(D + \rho z z^T)Q^T = Q(U \Lambda U^T)Q^T = W \Lambda W^T, \quad W = QU.$$  

3.1.1 Choice of $\theta$

In [15], for $\rho = \theta \beta$, scalar $\theta$ is chosen to avoid cancellation, per Dongarra and Sorensen [16]. This is shown in Algorithm 5, though some details are lacking. See http://www.netlib.org/misc/SEV/ for code.

**Algorithm 5** Choice of $\theta$ in Dongarra and Sorensen [16]

```plaintext
if sign($\hat{T}_1(n_1, n_1)$) == sign($\hat{T}_2(1, 1)$) then
  // $-\theta \beta$ has same sign as diagonal elements $\hat{T}_1(n_1, n_1)$ and $\hat{T}_2(1, 1)$.
  $\theta = -\text{sign}(\hat{T}_1(n_1, n_1)) \cdot \text{sign}(\beta)$
else
  // Unclear: $-\theta \beta$ has same sign as one of diagonal elements, and magnitude of $\theta$ chosen
  // to avoid cancellation when $\theta^{-1} \beta$ is subtracted from the other diagonal element.
end
```

Inconsistent with this, (Sca)LAPACK chooses $\theta = \text{sign}(\beta)$, since it does

```plaintext
D(i) = D(i) - abs(E(i-1)) ! ! D(i) - theta beta
```

SLATE follows (Sca)LAPACK’s convention, since LAPACK’s secular equation solver (laed4) requires $\rho > 0$.

3.1.2 Secular equation

The characteristic equation is $\det(A - \lambda I) = 0$. In this case,

$$\det(D + \rho z z^T - \lambda I) = 0.$$ 

(Cf. Golub for explanation?) This leads to the secular equation, so called because of centuries-long secular (non-periodic) perturbations of planetary orbits (see wikipedia).

Eigenvalues of $D + \rho z z^T$ are roots $\{\lambda_j\}$ of the secular equation

$$f(\lambda) = 1 + \rho z^T(D - \lambda I)^{-1}z = 1 + \rho \sum_{i=1}^{n} \frac{z_i^2}{d_i - \lambda}, \quad (3.4)$$
with corresponding (unnormalized) eigenvectors

\[ u_j = (D - \lambda_j I)^{-1} z = \begin{bmatrix} z_1 \\ d_1 - \lambda_j \\ \vdots \\ z_n \\ d_n - \lambda_j \end{bmatrix}, \quad j = 1, \ldots, n. \] (3.5)

Solving Equation (3.4) uses a custom non-linear solver developed by Li [17]. However, direct application of Equation (3.5) to compute eigenvectors can lead to loss of precision and orthogonality; see Section 3.1.6.

The secular equation solver assumes, without loss of generality, that \( \rho > 0 \). To accomplish this, set \( \theta = \text{sign}(\beta) \), which negates \( z_2 \) and \( \rho \); see Section 3.1.1.

The secular equation solver further requires that \( D \) is sorted in ascending order, accomplished by applying permutation \( P_s (\text{isort / INDX}^1) \) to obtain sorted \( D, z, U \) from unsorted \( D_o, z_o, U_o \):

\[ P_s(D_o + \rho z_o z_o^T)P_s^T = D + \rho z z^T = U \Lambda U^T = P_s(U_o \Lambda U_o^T)P_s^T. \]

3.1.3 Deflation

When an eigenvalue has already converged, it can be deflated from the secular equation, reducing the secular equation size by 1. \[15\] set tolerance

\[ \eta = \epsilon \| D + \rho z z^T \|_2 \leq \epsilon (\| D \|_2 + |\rho| \| z z^T \|_2) = \epsilon (\max_j |d_j| + |\rho|) \]

since \( \| z \|_2 = 1 \). However, (Sca)LAPACK uses

\[ \eta = 8u \max_j (\max |d_j|, \max |z_j|) \]

where \( u \) is unit roundoff \((\epsilon/2)\) (i.e., \( u = \text{lamch("e"}) \)). It’s unclear where this tolerance is derived from; presumably from LAPACK code in \[18\].

Two types of deflation can occur. The first is if \( z_j = 0 \) for some \( j \), then \( d_j \) is an eigenvalue of \( T \) with eigenvector \( e_j \). More generally, when \( z_j \) is nearly zero, deflate if

\[ |\rho z_j| \leq \eta. \]

The second type of deflation occurs if \( D \) has eigenvalue \( d_j \) of multiplicity \( m > 1 \), we can rotate to zero out all but one of the corresponding \( z_j \). More generally, when \( d_i \) and \( d_j \) (\( js1 \) and \( js2 / PJ \) and \( NJ \)) are nearly equal, deflate if

\[ \frac{|z_i z_j | \cdot |d_i - d_j|}{\sqrt{z_i^2 + z_j^2}} \leq \eta. \]

\(^1\)variable names in SLATE code and ScaLAPACK code, respectively.
This is the off-diagonal term when applying $G_{ij}$ to $2 \times 2$ of $d_i$ and $d_j$ values:

$$G_{ij} \begin{bmatrix} D_{i,i} & D_{i,j} \\ D_{j,i} & D_{j,j} \end{bmatrix} G_{ij}^T = \begin{bmatrix} c^2 d_i + s^2 d_j & -csd_i + csd_j \\ -csd_i + csd_j & s^2 d_i + c^2 d_j \end{bmatrix}$$

If these off-diagonal terms are negligible, then $D$ is still diagonal. The Givens rotation $G_{ij}$ that applies to rows $i$ and $j$ is defined as

$$G_{ij} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \quad s = -\frac{z_i}{r}, \quad c = \frac{z_j}{r}, \quad r = \sqrt{z_i^2 + z_j^2}, \quad c^2 + s^2 = 1 \quad \text{(by construction).}$$

(Formally, $G_{ij}$ is embedded in an $n \times n$ identity matrix.) Let $G_d$ be the product of all Givens rotations and $P_d$ (ideflate / INDX $^1$) be the product of all permutations for deflating eigenvalues. Then

$$P_d G_d (D + \rho zz^T) (P_d G_d)^T = \begin{bmatrix} D_s + \rho zsz_s^T & 0 \\ 0 & \Lambda_d \end{bmatrix} + E, \quad \|E\|_2 \leq c\eta.$$

where $D_s + \rho zsz_s^T$ defines the secular equation and $\Lambda_d$ are deflated eigenvalues. Let $D_s + \rho zsz_s^T = U_s \Lambda_s U_s^T$, be the solution of the secular equation. Then

$$U = \begin{bmatrix} U_s & 0 \\ 0 & I \end{bmatrix}.$$

### 3.1.4 Back-transformation

The main cost of divide-and-conquer is computing $W = QU$. With deflation,

$$QU = Q(P_d G_d)^T U$$

$$= \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} G_d^T P_d^T \begin{bmatrix} U_s & 0 \\ 0 & I \end{bmatrix}.$$

The $G_d^T P_d^T$ term can be multiplied on the left into $Q$, or on the right into $U$.

First, consider multiplying on the right. $G_d^T P_d^T U$ destroys the block sparsity structure in $U$, yielding something similar to the non-deflated case, with $n^3$ flops:

$$QU = Q \left( G_d^T P_d^T U \right)$$

$$= \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} \left( G_d^T P_d^T \begin{bmatrix} U_s & 0 \\ 0 & I \end{bmatrix} \right)$$

$$= \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \end{bmatrix}$$

$$= \begin{bmatrix} Q_1 U_1 \\ Q_2 U_2 \end{bmatrix}.$$

Now consider multiplying on the left, $QG_d^T P_d^T$. Add another permutation $\bar{P}$ (ibar / INDX $^1$) to retain some block sparsity structure in $Q$. $\bar{P}$ operates on only the first $n - n'_d$ cols of $QG_d^T P_d^T$. 

---

20
and rows of $U$, leaving the last $n_d'$ cols/rows due to deflation unaffected. (In ScaLAPACK, the $\bar{P}$ permutation overwrites the $P$ permutation in $\text{INDX}$.)

$$QU = (QG_d^T P_d^T) U$$

$$= (QG_d^T P_d^T \bar{P}) (\bar{P}^T U)$$

$$= \begin{bmatrix}
Q_{11} & Q_{12} & 0 & Q_{14} \\
0 & Q_{22} & Q_{23} & 0 \\
0 & 0 & 0 & I \\
0 & 0 & 0 & I
\end{bmatrix}
\begin{bmatrix}
P_d^T U_s & 0 \\
0 & I \\
0 & 0 & 0 & I
\end{bmatrix}
n_{d1} + n_{d2} + n_{d3}
$$

$$= \begin{bmatrix}
\bar{U}_{1:2} & \bar{Q}_{11} & \bar{Q}_{12} & \bar{Q}_{14} \\
\bar{Q}_{2:3} & \bar{Q}_{22} & \bar{Q}_{23} & \bar{Q}_{24}
\end{bmatrix}
$$

$$\bar{U} =
\begin{bmatrix}
\bar{U}_1 & 0 \\
\bar{U}_2 & 0 \\
\bar{U}_3 & 0 \\
0 & I
\end{bmatrix}
$$

where$

- $\bar{Q}_{11}$ has $n_{s1}$ cols of $Q$ unaffected by deflation (column type 1),
- $\bar{Q}_{23}$ has $n_{s2}$ cols of $Q$ unaffected by deflation (column type 3).
- $\begin{bmatrix}
\bar{Q}_{14} \\
\bar{Q}_{24}
\end{bmatrix}$ are eigenvectors from deflated eigenvalues (column type 4).

Givens rotation have filled in values, destroying block sparsity.

- $\begin{bmatrix}
\bar{Q}_{12} \\
\bar{Q}_{22}
\end{bmatrix}$ are the rest.

Again, Givens rotations have destroyed sparsity. In serial, these are all column type 2, which arise as non-deflated eigenvalue in each type 2 deflation. In parallel, these include some columns of type 1 and 3.

In the serial algorithm, $n_d' = n_d$ eigenvalues are deflated. In the parallel algorithm, $n_d$ eigenvalues are deflated, but the block has size $n_d' \leq n_d$ due to restrictions on $\bar{P}$ permuting only locally within a node. This yields the two gemm operations in (3.9) to compute $W$.

In parallel, the $\bar{Q}$ matrix has a local block structure like this, but the global structure is different.

---

$^2$Compared to Tisseur, renumbered using column type: $Q_{21} \rightarrow \bar{Q}_{22}, Q_{22} \rightarrow \bar{Q}_{23}, Q_{13} \rightarrow \bar{Q}_{14}, Q_{23} \rightarrow \bar{Q}_{24}$

---

21
3.1.5 Deflation Example

Notes:

- \( n_1 = 4, n_2 = 4, n = n_1 + n_2 = 8, n_b = 2, \text{ npcol } = 2 \) (number of process columns)
- entry \( z_4 = 0 \) indicating type 1 deflation
- entry \( d_2 = d_5 \) indicating type 2 deflation
- entry \( d_3 = d_6 \) indicating type 2 deflation
- \textit{deflate} shows the type of deflation for that column. For type 2 pair, 2a is deflated, 2b is kept.
- \textit{pcolumn} shows the process column
- \textit{coltype} shows the column type:
  1. Unaffected by deflation, part of \( Q_1 \)
  2. Dense, non-deflated eigenvector in type 2 deflation
  3. Unaffected by deflation, part of \( Q_2 \)
  4. Deflated, either type 1 or deflated eigenvector in type 2 deflation
- \( Q(1,:) \) is first row of \( Q \),
  \( Q(n,:) \) is last row of \( Q \). Together they serve to show the block sparsity of \( Q \).
- Numeric values for \( z \) and \( Q \) are not real (obviously), they just correspond with the \( D \) values, except for 0's. The decimal (.1 or .3) denotes the original coltype.

The ScaLAPACK code does not appear to accomodate the case of \( Q_{12} \) or \( Q_{23} \) being empty, since \( ib1 \leq ie1 \) and \( ib2 \leq ie2 \).
3.1. CUPPEN’S METHOD

CHAPTER 3. DIVIDE AND CONQUER

Original

<table>
<thead>
<tr>
<th>D₁</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>D₀ =</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.00 4.00 7.00 3.00</td>
</tr>
<tr>
<td>z₀ =</td>
<td></td>
</tr>
<tr>
<td>6.01 4.13 7.13 0</td>
<td>4.03 7.03 1.03 2.03</td>
</tr>
<tr>
<td>deflate =</td>
<td>2a 2a 1 2b 2b</td>
</tr>
<tr>
<td>pcolumn =</td>
<td>0 0 1 1 0 0 1 1</td>
</tr>
<tr>
<td>coltype =</td>
<td>1 1 1 1 3 3 3 3</td>
</tr>
<tr>
<td>Q(1,:) =</td>
<td>6.1 4.1 7.1 3.1</td>
</tr>
<tr>
<td>Q(n,:) =</td>
<td>0 0 0 0 4.3 7.3 1.3 2.3</td>
</tr>
</tbody>
</table>

Sorted

By applying permutation \( P_s \) given by isort / INDX, \( D = P_sD_o \) values are sorted. We’re abusing notation here, identifying the permutation \( P_s \) with the index vector isort. To get the actual \( P_s \) matrix, using Matlab notation:

\[
I = \text{eye}(n, n);
P_s = I(\text{isort}, :);
\]

These notations are equivalent:

\[
D_o(P_s) = D_o(\text{isort}) = P_sD_o,
Q(:,P_s) = Q(:,\text{isort}) = QP_s^T.
\]

| D = P_sD₀ = | 1.00 2.00 3.00 4.00 4.00 6.00 7.00 7.00 |
| z = P_sz₀ = | 1.03 2.03 0 4.13 4.03 6.01 7.13 7.03 |
| P_s = isort = | 7 8 4 2 5 1 3 6 |
| deflate(P_s) = | 2a 2b |
| pcolumn(P_s) = | 1 1 1 0 0 0 1 0 |
| coltype(P_s) = | 1 3 1 1 3 1 1 3 |
| Q(1,P_s) = | 0 0 3.1 4.1 0 6.1 7.1 0 |
| Q(n,P_s) = | 1.3 2.3 0 0 4.3 0 0 7.3 |
3.1. CUPPEN’S METHOD

CHAPTER 3. DIVIDE AND CONQUER

Deflated

With permutation \( P_D \) given by \texttt{ideflate} / \texttt{INDXP} \(^1\), deflated eigenvectors (col type 4, light orange) are moved to the end and sorted descending. (todo: any reason for sorting deflated eigvals?) Non-deflated eigenvalues are sorted ascending, as required by secular equation solver. \( P_D = P_s P_d \) includes the effect of \( P_s \) sorting. Columns of \( Q \) corresponding to type 2 deflation have lost their block sparsity, becoming dense (.1 → .12 or .14 and .3 → .32 or .34 to indicate modified values, where the second digit denotes its new coltype).

These notations are equivalent:

\[ D_o(P_D) = D_o(\texttt{ideflate}) = P_D D_o, \]
\[ Q(:,P_D) = Q(:,\texttt{ideflate}) = Q P_T D. \]

---

Locally permuted

With permutation \( \bar{P} \) given by \texttt{ibar} / \texttt{INDX} \(^1\), within pcolumn 0 (dark blue), coltypes are sorted, e.g.: 1, 2, 2, 4; within pcolumn 1 (light blue), coltypes are sorted, e.g.: 3, 3, 4, 4.

\([\bar{Q}_{11} \quad \bar{Q}_{12}]\) spans columns 1–5 (dark purple), to cover all coltype 1 and 2. It also includes a couple coltype 3, but this would not necessarily occur. It could also include coltype 4.

\([\bar{Q}_{22} \quad \bar{Q}_{23}]\) spans columns 2–5 (light purple), to cover all coltype 2 and 3. In this case, it does not include any coltype 1 or 4, but it could.

The code copies \( \bar{Q}( :, \texttt{ibar}) = Q(:,\texttt{ideflate}) \), that is, \( \bar{Q} = Q P_T D \bar{P} \) with no transpose on \( \bar{P} \). Thus \( Q(:,\texttt{ibar}) \) is in the same order as \( D(\texttt{ideflate}) \).

---

---

24
Globally permuted

If we globally permuted the matrix (e.g., in the serial algorithm), coltypes would be sorted globally: 1, 2, 3, 4.

In this case, \[
\begin{bmatrix}
\tilde{Q}_{11} & \tilde{Q}_{12}
\end{bmatrix}
\] would span columns 1–3 (dark purple), to cover all coltype 1 and 2. No coltype 3 would be included.

\[
\begin{bmatrix}
\tilde{Q}_{22} & \tilde{Q}_{23}
\end{bmatrix}
\] would span columns 2–5 (light purple), to cover all coltype 2 and 3. No coltype 1 would be included.

Again, the code doesn’t compute \(\text{igbar}\). It computes \(iQ\) such that \(\tilde{Q} = \tilde{Q}(;,iQ)\). Hence \(\text{igbar} = \text{ibar}(iQ)\).

<table>
<thead>
<tr>
<th>(D(p) = 6.00)</th>
<th>(4.00)</th>
<th>(7.00)</th>
<th>(1.00)</th>
<th>(2.00)</th>
<th>(7.00)</th>
<th>(4.00)</th>
<th>(3.00)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(z(p) = 6.00)</td>
<td>(4.00)</td>
<td>(7.00)</td>
<td>(1.00)</td>
<td>(2.00)</td>
<td>(7.00)</td>
<td>(4.00)</td>
<td>(3.00)</td>
</tr>
</tbody>
</table>

| \(p = \text{igbar} = \) | 1 | 5 | 6 | 7 | 8 | 3 | 2 | 4 |
| \(\text{deflate}(p) = \) | 2 | 2 | 2 | 2 | 1 |
| \(\text{pcolumn}(p) = \) | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| \(\text{coltype}(p) = \) | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 4 |

| \(Q(1,p) = \) | 6.1 | 4.34 | 7.34 | 0 | 0 | 7.14 | 4.14 | 3.1 |
| \(Q(n,p) = \) | 0 | 4.34 | 7.34 | 1.3 | 2.3 | 7.14 | 4.14 | 0 |
| \(iQ = \) | 1 | 2 | 5 | 3 | 4 | 7 | 6 | 8 |
Deflation Example 2

Same setup, except type 1 deflation of entries \( d_5 \) and \( d_6 \). In this case, entries 5 and 6 are moved to the end of process 0, and \( \begin{bmatrix} Q_{11} \\ Q_{24} \end{bmatrix} \) is physically in the middle. Having no type 2 deflation, there are no coltype 2.

**Original**

\[
D = \begin{bmatrix}
6.00 & 5.00 & 8.00 & 3.00 \\
4.00 & 7.00 & 1.00 & 2.00 \\
\end{bmatrix}
\]

\[
z = \begin{bmatrix}
6.11 & 5.11 & 8.11 & 3.11 \\
0 & 0 & 1.33 & 2.33 \\
\end{bmatrix}
\]

\[\text{deflate} = \begin{bmatrix} 1 & 1 \end{bmatrix}\]

**Locally permuted**

\[
D(p) = \begin{bmatrix}
5.00 & 6.00 & 3.00 & 8.00 \\
7.00 & 4.00 & 1.00 & 2.00 \\
\end{bmatrix}
\]

\[
z(p) = \begin{bmatrix}
5.11 & 6.11 & 3.11 & 8.11 \\
0 & 0 & 1.33 & 2.33 \\
\end{bmatrix}
\]

\[\bar{P} = \text{ibar} = \begin{bmatrix} 7 & 8 & 3 & 1 \\
2 & 4 & 5 & 6 \end{bmatrix}\]

\[\text{deflate}(p) = \begin{bmatrix} 1 & 1 \end{bmatrix}\]

\[
\text{pcolumn}(p) = \begin{bmatrix} 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \end{bmatrix}
\]

\[
\text{coltype}(p) = \begin{bmatrix} 1 & 1 & 1 & 1 \\
4 & 4 & 3 & 3 \end{bmatrix}
\]

\[
Q(1,p) = \begin{bmatrix}
5.1 & 6.1 & 3.1 & 8.1 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
Q(n,p) = \begin{bmatrix}
7.3 & 4.3 & 1.3 & 2.3 \\
\end{bmatrix}
\]

Deflation Example 3

Same setup, except type deflation of entries \( d_1 \), \( d_2 \), and \( d_3 \). Two coltype 3 entries are permuted up to columns 1 and 2. \( \begin{bmatrix} \bar{Q}_{11} & \bar{Q}_{12} \end{bmatrix} \) is only column 3 (dark purple), while \( \begin{bmatrix} \bar{Q}_{22} & \bar{Q}_{23} \end{bmatrix} \) spans 1–7, covering coltypes 3, 3, 1, 3, 4, 4, 3.

**Original**

\[
D(p) = \begin{bmatrix}
6.00 & 5.00 & 8.00 & 3.00 \\
4.00 & 7.00 & 1.00 & 2.00 \\
\end{bmatrix}
\]

\[
z(p) = \begin{bmatrix}
0 & 0 & 0 & 3.11 \\
4.33 & 7.33 & 1.33 & 2.33 \\
\end{bmatrix}
\]

\[\text{deflate} = \begin{bmatrix} 1 & 1 \end{bmatrix}\]

**Locally permuted**

\[
D(\bar{P}) = \begin{bmatrix}
4.00 & 7.00 & 3.00 & 1.00 \\
6.00 & 5.00 & 2.00 & 8.00 \\
\end{bmatrix}
\]

\[
z(\bar{P}) = \begin{bmatrix}
4.33 & 7.33 & 3.11 & 1.33 \\
0 & 0 & 2.33 & 0 \\
\end{bmatrix}
\]

\[\bar{P} = \text{ibar} = \begin{bmatrix} 4 & 7 & 3 & 1 \\
2 & 8 & 5 & 6 \end{bmatrix}\]

\[\text{deflate}(\bar{P}) = \begin{bmatrix} 1 & 1 \end{bmatrix}\]

\[
\text{pcolumn}(\bar{P}) = \begin{bmatrix} 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \end{bmatrix}
\]

\[
\text{coltype}(\bar{P}) = \begin{bmatrix} 3 & 3 & 1 & 3 \\
4 & 4 & 3 & 4 \end{bmatrix}
\]

\[
Q(1,\bar{P}^T) = \begin{bmatrix}
0 & 0 & 3.1 & 0 \\
6.1 & 5.1 & 0 & 8.1 \\
\end{bmatrix}
\]

\[
Q(n,\bar{P}^T) = \begin{bmatrix}
4.3 & 7.3 & 0 & 1.3 \\
0 & 0 & 2.3 & 0 \\
\end{bmatrix}
\]
3.1.6 Eigenvectors via Löwner Theorem

When computing eigenvectors from the computed eigenvalues \( \tilde{\lambda}_j \) of \( D + \rho \tilde{z} \tilde{z}^T \) using (3.5), repeated here,

\[
\mathbf{u}_j = (D - \lambda_j I)^{-1} \mathbf{z} = \begin{bmatrix}
\frac{z_1}{d_1 - \lambda_j} \\
\vdots \\
\frac{z_n}{d_n - \lambda_j}
\end{bmatrix}
\]

for \( j = 1, \ldots, n \),

if \( d_i \approx \lambda_j \), then \( z_i/(d_i - \lambda_j) \) can be inaccurate, causing a loss of orthogonality in \( U \). Instead, consider \( \{\tilde{\lambda}_j\} \) as exact eigenvalues of the modified system \( D + \rho \tilde{z} \tilde{z}^T \). See derivation in Tisseur and Dongarra [15]; here each term is multiplied by \(-1\) to match \texttt{lased4} code. Result:\(^3\)

\[
\tilde{z}_i = \pm \sqrt{-\prod_{j=1}^{n} \frac{d_i - \tilde{\lambda}_j}{d_i - d_j}} = \pm \sqrt{-\prod_{j=1}^{n} \frac{\delta_{i,j}}{d_i - d_j}} \quad \text{with } \delta_{i,j} = d_i - \tilde{\lambda}_j,
\]

yielding (unnormalized) eigenvectors

\[
\tilde{\mathbf{u}}_j = (D - \tilde{\lambda}_j I)^{-1} \tilde{\mathbf{z}} = \begin{bmatrix}
\frac{\tilde{z}_1}{d_1 - \tilde{\lambda}_j} \\
\vdots \\
\frac{\tilde{z}_n}{d_n - \tilde{\lambda}_j}
\end{bmatrix}
\]

for \( j = 1, \ldots, n \),

where \( \odot \) denotes element-wise division (Matlab \( ./ \)).

Both Tisseur and Dongarra [15] and Gu and Eisenstat [19] seem to gloss over the sign of \( \tilde{z}_i \). (Sca)LAPACK copies the sign from \( z_i \) in \texttt{dlaed3.f}:

\[
\texttt{w}(i) = \text{sign}(\sqrt{-\texttt{w}(i)}, s(i))
\]

where \( \texttt{w} \) (\texttt{ztilde}) contains the “first \( k \) (\texttt{nsecular}) values of the final deflation-altered \( z \)-vector”, per \texttt{dlaed2}.

\(^3\)Tisseur has upper limit \( \prod_{j=1, j \neq i}^{n} \); Gu and Eisenstat [19] correctly has \( \prod_{j=1, j \neq i}^{n} \).
3.1. CUPPEN’S METHOD

CHAPTER 3. DIVIDE AND CONQUER

Parallelization of eigenvectors

ScaLAPACK computes \( n \) roots on \( p \) processes as follows:

\[ n_c = n/npcol \quad (klc) \]

is the number of eigenvalues each process column computes.

\[ n_r = n_c/nprow \quad (klr) \]

is the number of eigenvalues each process row within a process column computes.

Each process computes \( n_r \) eigenvalues, calling LAPACK’s \texttt{laed4} for each. (Within each pcol, prov 0 (or drow?) computes any remainder. Globally, pcol 0 (or dcol?) computes any remainder.)

After each call to \texttt{laed4}, each process updates its \( \bar{z} \) vector of partial products, multiplying one more term:

\[
\bar{z}_i = \prod_{k=1}^{n_r} (d_i - \tilde{\lambda}_k) = \prod_{k=1}^{n_r} \frac{\delta_{i,i}}{\prod_{k=1}^{n_r} (d_k - d_i)} \quad \text{for } i = 1 \ldots n,
\]

where \( k_g \) maps from local index \( k \) to global index \( k_g \).

Then it does a global multiply reduction (in a 2D fashion) of \( \bar{z} \), similar to \texttt{MPI_Reduce}. The root node finishes the computation of \( \tilde{\Lambda} \),

\[
\tilde{z}_i = \text{sign}(z_i) \sqrt{- \prod_{\text{process } p} \bar{z}^{(p)}_i} \quad \text{for } i = 1 \ldots n,
\]

then broadcasts \( \tilde{z} \).

It also gathers (in a 2D fashion) the computed eigenvalues, \( \tilde{\Lambda} \), to the root node, then broadcasts it back out, similar to \texttt{MPI_Allgather}.

Now, each process computes its local portion of the \( U \) matrix, per the 2D block cyclic distribution. For each column \( j \) of \( U \) that pcol \( p \) owns, each process in pcol \( p \) redundantly re-computes the corresponding eigenvalue \( \lambda_j \) and \( \delta_j \) vector. Each process in pcol then redundantly computes the entire vector \( u_j = \tilde{z}_j \odot \delta_j \), takes its norm, and normalizes and saves just the portion that the process owns. Note this calls \texttt{laed4} redundantly \( nprow \) times, thus limiting its parallel speedup to \( npcol \) times. However, it is \( O(n^2) \) work, so may not take significant time.

Despite the definition

\[
\delta_{i,j} = d_i - \tilde{\lambda}_j,
\]

using that definition — even with the compute \( \tilde{\lambda}_j \) — produces inaccurate results (as confirmed in MAGMA). This is unfortunate as recomputing \( \delta \) in that fashion would avoid all the redundant \texttt{laed4} calls. It’s also unclear since Gu and Eisenstat [19] seems to use that definition. Neither Rutter [18] nor Li [17] seem to discuss computing \( \delta_{i,j} \) in \texttt{laed4} and why that might be necessary for stability.

Alternatively, it could skip all this redundant computation by saving the \( \delta_j \) vectors and communicating them, and doing a distributed computation of column norms to normalize the vectors.
3.1.7 Cost

Without deflation, flops for multiplying $QU$ is $n^3 + O(n^2)$, since it is two gemms of size $\frac{n}{2} \times \frac{n}{2} \times n$:

$$QU = \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} = \begin{bmatrix} Q_1 U_1 \\ Q_2 U_2 \end{bmatrix}.$$ 

The total cost for divide-and-conquer thus satisfies the recursion

$$t_n = n^3 + 2t_{n/2}$$

with solution

$$t_n \approx \frac{4}{3} n^3 + O(n^2).$$

With deflation, flops can be $O(n^{2.3})$ on average, or $O(n^2)$ in special cases.
3.2 Routines

3.2.1 stedc

Symmetric Tridiagonal Eigenvalue Divide & Conquer solver, top-level routine called from heev.

**Algorithm 6** Main divide & conquer driver

```verbatim
function stedc(D, E, Q)
    input: real tridiagonal matrix A represented by diagonal D and sub-diagonal E vectors
    output: eigvals D (sorted) and eigvecs Q of A
            scale A (i.e., D and E) by 1/∥A∥, so it has unit norm
            allocate workspaces W, U
            // Computing in workspace W avoids copy in sort, compared to ScaLAPACK.
            stedc_solve( D, E, W; workspace Q, U ) computes eigvals D and eigvecs W
            stedc_sort( D, W, Q ) sorts eigvals D and permutes eigvecs W into Q
            scale eigvals D back by original ∥A∥
end function
```

---

30
3.2. ROUTINES  

3.2.2 stedc_solve

Main divide & conquer driver.
Corresponds to ScaLAPACK laed0.

**Algorithm 7** Main divide & conquer driver

```plaintext
function stedc_solve(D, E, Q; workspace W, U)
input: A represented by diagonal D and sub-diagonal E vectors.
output: D is (unsorted) eigvals and Q is eigvecs of A.

// Tear into subproblems
for i = b, 2b, . . . , n − 1 with blocksize b
    subtract ρ = |Ei−1| from Di−1 and Di
end

// Solve subproblems
parallel for each block-col i = 0, b, . . . , n; split over MPI ranks and OpenMP threads
    // todo: In 2DBC, seems highly load imbalanced
    // —only nodes assigned diagonal tiles do work.
    if Qi,i is local then
        i2 = i + nb − 1
        lapack::steqr( Di:i2, Ei:i2−1, Qi,i ) or stedc
        to solve subproblem using serial algorithm
    end
end

gather and bcast D to all nodes

// Merge subproblems
for each level in divide & conquer tree, from leaf to root
    for each pair of subproblems, indexed i, . . . , i1 and i1 + 1, . . . , i2
        ρ = Ei1 // abs dealt with in merge routine
        stedc_merge( ρ, Di:i2, Qi:i2, W:i2, Ui:i2 )
    end
end
end function
```
3.2.3 \texttt{stedc\_merge}

Merges two subproblems.
Corresponds to ScaLAPACK \texttt{laed1}.

\begin{verbatim}
function stedc_merge(D, Q; workspace \bar{Q}, \bar{U})
  input: eigvals \(D_1\) and \(D_2\) in \(D\), eigvecs \(Q_1\) and \(Q_2\) in \(Q\) of subproblems
  output: eigvals \(D\) and eigvecs \(Q\) of merged problem
    stedc_z_vector( Q, z ) gets \(z\)
    stedc_deflate( D, z, Ds, zs, Q, \bar{Q}, \bar{P} ) deflates \(n_d\) eigvals, leaving \(n_s\) secular eqn eigvals
    stedc_secular( Ds, zs, U, \bar{P} ) solves secular equation for eigvals \(Ds\) and eigvecs \(U\)
    // todo: merge \(D\) and \(Ds\)
    Q_{1,1:2} = \bar{Q}_{1,1:2}U_{1:2}
    Q_{2,2:3} = \bar{Q}_{2,2:3}U_{2:3}
    Copy with permutation deflated eigvecs \(\bar{Q}_{1:2,4}\) to \(Q\)
end function
\end{verbatim}
3.2.4 stedc_z_vector

Gathers onto all nodes vector $z$ that is last row of $Q_1$ and first row of $Q_2$,

$$z = Q^T \begin{bmatrix} e_{n_1} \\ e_1 \end{bmatrix} = \begin{bmatrix} Q_1^T e_{n_1} \\ Q_2^T e_1 \end{bmatrix}.$$ 

Corresponds to ScaLAPACK laedz.

This is conceptually like MPI Allgatherv, but due to 2DBC distribution, it doesn’t seem a single Allgatherv could do this. Alternatively, each rank could pack its local pieces, then do MPI Gatherv, root unpacks to correct locations, and MPI Bcast; or MPI Allgatherv and everyone unpacks (without bcast).

```plaintext
function stedc_z_vector( Q, z )
input: eigvecs $Q_1$ and $Q_2$ of subproblems in $Q$
output: vector $z$ is last row of $Q_1$ and first row of $Q_2$
for each block-col $j = 0, \ldots, n_t - 1$
  if $j < n_{t1}$ then
    $i = n_{t1} - 1$  // last block-row of $Q_1$
  else
    $i = n_{t1}$  // first block-row of $Q_2$
  end
  if $Q_{i,j}$ is local then
    copy last or first row of $Q_{i,j}$ to $z_{i:i+b}$
    MPI_send $z_{i:i+b}$ to root, if rank $\neq$ root
  else if rank == root then
    MPI_recv $z_{i:i+b}$ from source
  end
end
MPI_Bcast z to all ranks
end function
```
3.2. ROUTINES

3.2.5 stedc_deflate

Deflates eigenvalues where $z_i$ is (close to) zero (type 1), or where two eigenvalues are (nearly) the same (type 2), identified by applying a rotation to zero out $z_i$. Forms permutation to group columns of $Q$ according to the column type:

- column type 1: non-deflated eigvecs from $Q_1$
- column type 2: non-deflated eigvecs updated by deflation
- column type 3: non-deflated eigvecs from $Q_2$
- column type 4: deflated eigvecs

Locally within each rank,

$$ Q_{\text{local}} = \begin{bmatrix} Q_{1,1} & Q_{1,2} & 0 & Q_{1,4} \\ 0 & Q_{2,2} & Q_{2,3} & Q_{2,4} \end{bmatrix}. $$

Corresponds to ScaLAPACK laed2.

**Algorithm 8** Deflation, part 1

```plaintext
function stedc_deflate( \( \rho \), \( D \), \( D_s \), \( z \), \( z_s \), \( Q \), \( \bar{Q} \) )

input: \( \rho \) that tore subproblems,
- eigenvalues \( D_1 \) and \( D_2 \) in vector \( D \),
- \( z_1 \) and \( z_2 \) in \( z \),
- eigvecs \( Q_1 \) and \( Q_2 \) of subproblems in \( Q \).

output: \( D \) has \( n_d \) deflated eigvals,
- \( D_s \) has \( n_s \) non-deflated eigvals for secular equation,
- \( z_s \) of length \( n_s \) is updated \( z \) vector for secular equation,
- \( \bar{Q} \) is \( n_s \) updated eigvecs permuted into \( Q_{1,1:2}, Q_{2,2:3}, Q_{1,2:4} \), another permutation?

// LAPACK secular equation solver (laed4) requires \( \rho > 0 \)
if \( \rho < 0 \) then
  \( \rho = -\rho \)
  \( z_2 = -z_2 \)
end
// \( z_1 \) and \( z_2 \) are normalized; re-normalize so \( \|z\|_2 = 1 \)
\( \rho = 2\rho \)
\( z = z/\sqrt{2} \)
compute permutation \( P_s \) to sort \( D \)
// note lamch("e") is unit roundoff \( u = \epsilon/2 \)
\( \text{tol} = 4\epsilon \max(\|D\|_\infty, \|z\|_\infty) \)
if \( \rho \|z\|_\infty < \text{tol} \) then
  \( n_s = 0 \) return
end
```

34
Algorithm 9 Deflation, part 2

// Deflate eigvals
// A candidate eigval is non-negligible (not type 1), but may have type 2 deflation.
// $j_{s_1}$ is candidate eigval; initially none ($-1$). $s$ indicates sorted permutation.
// $j_{s_2}$ is current eigval under consideration.
$j_{s_1} = -1$
for $j = 0, \ldots, n - 1$
    $j_{s_2} = P_s[j]$
    if $|\rho_{j_{s_2}}| < \text{tol}$ then
        store $j_{s_2}$ as deflated eigval (type 1)
    else if not first candidate eigval (i.e., $j_{s_1} \geq 0$) then
        generate Givens rotation $G$ to zero first entry of
        \[
        \begin{bmatrix}
        z_{j_{s_1}} \\
        z_{j_{s_2}}
        \end{bmatrix}
        \]
        compute $D_{j_{s_1},j_{s_2}}$ off-diagonal from applying $G \begin{bmatrix}
        D_{j_{s_1}} & 0 \\
        0 & D_{j_{s_2}}
        \end{bmatrix} G^T$
        if $D_{j_{s_1},j_{s_2}} < \text{tol}$ then
            update $\begin{bmatrix}
            z_{j_{s_1}} \\
            z_{j_{s_2}}
            \end{bmatrix} = G \begin{bmatrix}
            z_{j_{s_1}} \\
            z_{j_{s_2}}
            \end{bmatrix} = \begin{bmatrix}
            0 \\
            \tau
            \end{bmatrix}$
            update columns $Q_{j_{s_1},j_{s_2}} = Q_{j_{s_1},j_{s_2}} G$ // involves MPI for remote columns
            update $\begin{bmatrix}
            D_{j_{s_1}} \\
            D_{j_{s_2}}
            \end{bmatrix} = G \begin{bmatrix}
            D_{j_{s_1}} & 0 \\
            0 & D_{j_{s_2}}
            \end{bmatrix} G^T$ // off-diag $D_{j_{s_1},j_{s_2}}$ is negligible
        store $j_{s_1}$ as deflated eigval (type 2)
        else
            store $j_{s_1}$ as non-deflated eigval
        end
    end
    $j_{s_1} = j_{s_2}$ // $j_{s_2}$ becomes next candidate eigval
else
    $j_{s_1} = j_{s_2}$ // $j_{s_2}$ becomes first candidate eigval
end
store $j_{s_1}$ as non-deflated eigval

Algorithm 10 Deflation, part 3 (todo)

// find permutation to group types 1, 2, 3 together locally.
// find global permutation.
// find indices of $Q_{1:2}, Q_{2:3},$ and $U_{1:3}$.
end function stedc.deflate

35
3.2. ROUTINES

3.2.6 stedc_secular

Solves secular equation and computes eigenvectors via Löwner theorem. Corresponds to ScaLAPACK laed3.

```plaintext
function stedc_secular( D, z, P_u, Λ, U )
input: deflation-adjusted D, z, global permutation P_u
output: eigvals Λ of merged problem, eigvecs U of merged problem (before multiplying by Q)

// Compute Λ and modified ˜z.
˜z = 1
parallel for j = 0, ..., n_s - 1; split over MPI ranks
lapack::laed4( D, z, λ_j, δ_j ) solves secular equation for λ_j and δ_j vector
for i = 0, ..., n_s - 1 (todo: OpenMP parallel?)
if i == j then
  ˜z_i *= δ_ij
else
  ˜z_i *= δ_ij / (D_i - D_j)
end
end
MPI_Allreduce ˜z
fix sign ˜z_j to match sign z_j, for j = 0, ..., n - 1
MPI_Allgather Λ
permute Λ = P_u Λ

// Compute U via Löwner theorem.
// All processes within a process column do this computation redundantly.
// We could avoid that by communicating u_j.
parallel for j = 0, ..., n_s - 1; split over MPI 2DBC process columns
Re-compute secular equation (laed4) to get δ_j vector
u_j = ˜z ⊗ δ_j element-wise
u_j = u_j / ||u_j||
Store local part of u_j
end
end function
```
3.2.7 stedc_sort

Sorts eigenvalues $D$ and applies same permutation to eigenvectors $Q$.
Corresponds to ScaLAPACK lasrt.
function stedc_sort( D, Q, Qout )

input: eigvals D, eigvecs Q
output: sorted eigvals D, permuted eigvecs Qout

compute permutation $P_s$ to sort eigvals.
compute inverse permutation $P_s^{-1}$

for each block-col $j = 0, \ldots, n_t - 1$

// todo: these are bad descriptions
fill pcols[jj] with destination process of column $P_s^{-1}(j + jj)$ for $jj = 0, \ldots, jb - 1$
fill mine[jj] with $P_s^{-1}(jj)$ where pcols[jj] == mycol
fill pcnts[p] = length( where( pcnts[jj] == p ) ) for $p = 0, \ldots, ncol$
fill poffset = prefix_sum( pcnt )
if block-col $j$ is local then
    for $jj = 0, \ldots, jb - 1$
        $j_g = j + jj$
        $k_g = P_s^{-1}(j + jj)$
        if local then
            local copy $Q(:, j_g) \rightarrow Qout(:, k_g)$
        else
            // Pack into workspace
            copy $Q(:, j_g) \rightarrow$ work( :, poffset($p_k$) )
            poffset($p_k$) += 1
        end
    end
for $p = 0, \ldots, ncol - 1$
    if $p \neq me$ and pcnt[$p$] > 0 then
        MPI_Send pcnt[$p$] columns at work( :, poffset($p$) ) to rank( myrow, $p$ )
    end
end
else
    MPI_Recv pcnt[ mycol ] columns at work from rank( myrow, pj )
for $jj = 0, \ldots, \text{length}( mine )$
    $k_g = \text{mine}(jj)$
    copy work( :, jj ) to $Qout(:, k_g)$
end
end
end function
4.1 Hermitian to Hermitian band reduction (he2hb)

In this section, we present the performance optimization of the Hermitian to Hermitian band reduction \texttt{he2hb}. In [20] we show the optimization technique to extend the parallelization of the different steps in \texttt{he2hb} on CPU only. Where we introduce a new internal functions of the various operations of \texttt{he2hb}, such as \texttt{internal::he2hb_hemm<HostTask>}, \texttt{internal::he2hb_trmm<Target::HostTask>}, etc. As most of the operations in \texttt{he2hb} are expressed through Level 3 BLAS, there is obviously still room for improvement by doing GPU computations and further hiding the communication overhead by computations. Therefore, we provide a GPU implementation of the new internal functions introduced in \texttt{he2hb}. Figure 4.1 shows the performance of \texttt{he2hb} using CPU and 1 GPU, 2 GPUs on 1 node and 2 nodes. Using single GPU achieves up to 3.2× and 1.6× speedup compared to the host test on a 1 node and 2 nodes, respectively.

We generate traces using the Nvidia Nsight Systems viewer \texttt{nsys} to highlight the performance bottlenecks. The traces shows that the panel factorization is the most time consuming and it does not overlap with any of the subsequent computations and data transfer. Therefore, we add new omp tasks to overlap the panel factorization with data movements, as long as the data dependencies satisfied. Figure 4.2 shows the panel factorization overlap with allocating batch arrays and create CUDA streams, and sending the data to the GPU. Figure 4.3 studies the impact of this change on the \texttt{he2hb} performance using different number of GPUs, the new implementaion with enabling the data transfer during the panel factorization achieves up to 20% improvement compared to the initial implementation.

Figure 4.4 shows the performance of \texttt{he2hb} in time and Gflops using 1 node on Summit.
Figure 4.1: Performance of he2hb Using 1 and 2 nodes on Summit, $1 \times 1$ and $2 \times 2$ process grids.

Figure 4.2: Overlap the panel factorization with subsequent data movements.
4.1. HERMITIAN TO HERMITIAN BAND REDUCTION

**Figure 4.3:** The performance impact by overlapping the panel factorization with subsequent data movements.

**Figure 4.4:** Performance results of Hermitian to Hermitian band reduction, using 1 node, $1 \times 1$ process grid. nb = 128,320, ib= 16, 48 for CPU, GPU tests, panel-threads=10
4.2 Back-transformation (unmtr_hb2st)

The initial implementation of the second stage back-transformation (unmtr_hb2st) was sequential. For optimization of the unmtr_hb2st routine presented in Algorithm 4, CPU-only OpenMP parallelism is introduced. Then for further optimization, the gemm operations are moved to GPU. The performance comparison of these two implementations is presented in Figure 4.5. The device implementation of unmtr_hb2st achieves up to 6x speedup over the CPU-only implementation.

![Figure 4.5: Performance of unmtr_hb2st on a host with two 20-core Intel Broadwell Xeon E5-2698 v4 CPUs and one NVIDIA V100 activated. N=16384. As seen in the figure, the device implementation provides up to 6x speedup.](image)
CHAPTER 5

Performance

5.1 Environment

5.1.1 Hardware

Performance numbers were collected using the Summit system \(^{12}\) at the Oak Ridge Leadership Computing Facility (OLCF). Summit is equipped with IBM POWER9 processors and NVIDIA V100 (Volta) GPUs. Each of Summit’s nodes contains two POWER9 CPUs (with 22 cores each) and six V100 GPUs. Each node has 512 GB of DDR4 memory, and each GPU has 16 GB of HBM2 memory. NVLink 2.0 provides all-to-all 50 GB/s connections for one CPU and three GPUs (i.e., one CPU is connected to three GPUs with 50 GB/s bandwidth each, and each GPU is connected to the other two with 50 GB/s bandwidth each). The two CPUs are connected with a 64 GB/s X Bus. Each node has a Mellanox enhanced-data rate (EDR) InfiniBand network interface controller (NIC) that supports 25 GB/s of bi-directional traffic. Figure 5.1 shows the hardware architecture of a Summit node.

5.1.2 Software

The software environment used for the SVD experiments included:

- GNU Compiler Collection (GCC) 6.4.0,
- NVIDIA CUDA 10.1.105,
- IBM Engineering Scientific Subroutine Library (ESSL) 6.1.0,

\(^{1}\)https://www.olcf.ornl.gov/summit/
\(^{2}\)https://en.wikichip.org/wiki/supercomputers/olcf-4
5.2. Results

Here, we present the results of our preliminary performance experiment with the singular value solve. Figure 5.2 shows the execution time of ScaLAPACK compared to SLATE with and without GPU acceleration. Two MPI ranks are mapped to one node of Summit, i.e., one rank is mapped to one CPU socket (22 cores) and three GPU devices. Only singular values are computed in all cases (no vectors).

For a matrix of size $32,768 \times 32,768$, ScaLAPACK took 925 seconds, while SLATE took 324 seconds using CPUs only and 233 seconds with GPU acceleration. That is, SLATE was almost
5.2. RESULTS

Figure 5.2: SVD performance comparison.

3 times faster without acceleration and almost 4 times faster with acceleration. Since the performance gap increases with the problem size, we expect SLATE to be an order of magnitude faster for matrices in the $O(100K)$ range without acceleration, and further benefit $3 \times$ to $4 \times$ from acceleration.

For the generalized Hermitian definite eigenvalue problem, Figure 5.3 shows the performance for conversion from the generalized form to standard form (hegst). On the CPU host, SLATE closely matches ScaLAPACK’s performance, while when using GPUs, SLATE gets a modest acceleration. We will continue to investigate ways to optimize the performance.

Figure 5.4 presents the performance breakdown of eigensolver routines in ScaLAPACK and SLATE. Double precision is used for all routines. ScaLAPACK’s pdsyev routine with QR iteration and pdsyevd routine with the D&C algorithm are used. Both pdsyev and pdsyevd implement 1-stage reduction. SLATE’s eigensolver is based on 2-stage reduction and it has the recently-implemented tridiagonal eigensolver with the D&C algorithm. The results belong to only one node of Summit. For ScaLAPACK, 6-by-6 process grid consisting of 36 MPI ranks is used, whereas for SLATE, 2-by-2 process grid having 9 cores and 1 GPU per rank is used. Consequently, both libraries are run on 36 cores for the sake of fair comparison. Both libraries are tuned for various block sizes. The best block sizes are found to be 96 for ScaLAPACK and 224 for SLATE. The default inner blocking size, which is 16, and 6 panel threads for the QR algorithm are used for SLATE.

As seen in Figure 5.4, the solve part with D&C algorithm in ScaLAPACK is significantly faster than the solve part with the QR iteration. The red bars in the figure represent times spent for the eigensolver. The most time consuming eigensolver is the one based on QR iteration in ScaLAPACK. When ScaLAPACK and SLATE with the D&C algorithm are compared, the first stage hermitian to band and the back transformation times are shorter in SLATE since SLATE utilizes GPUs for these computations. In the overall comparison, SLATE is slightly faster than ScaLAPACK. The second stage band to tridiagonal reduction and the recently-implemented D&C tridiagonal eigenvalue solver in SLATE need further optimization to better utilize the available system resources including GPUs.
5.2. RESULTS

CHAPTER 5. PERFORMANCE

DHEGST: Time taken on 18 nodes
18 nodes x (42 POWER9 + 6 V100 per node) (summit@ORNL)

Figure 5.3: Generalized to standard eigenvalue performance comparison.

Figure 5.4: Profile of eigenvalue solver implementations showing each phase for N=12288. One node of Summit is used.


