SLATE WORKING NOTES 13

Implementing Singular Value and Symmetric/Hermitian Eigenvalue Solvers

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

Innovative Computing Laboratory

June 26, 2023

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

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.

Revision	**Notes**
2019-09	first publication
2020-04	added generalized Hermitian definite eigenvalues (Section 2.3) and eigenvectors (Section 2.6)
2021-12	added divide and conquer (Chapter 3) and optimization (Chapter 4)
2022-11	added details of Hermitian to Hermitian band (Section 2.5)
2023-06	added details of divide and conquer

```
@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={2023},
  month={June},
  number={ICL-UT-19-07},
  note={first published 2019-09, revision 2023-06}
}
```

Contents

List of Figures

CHAPTER 1

Introduction

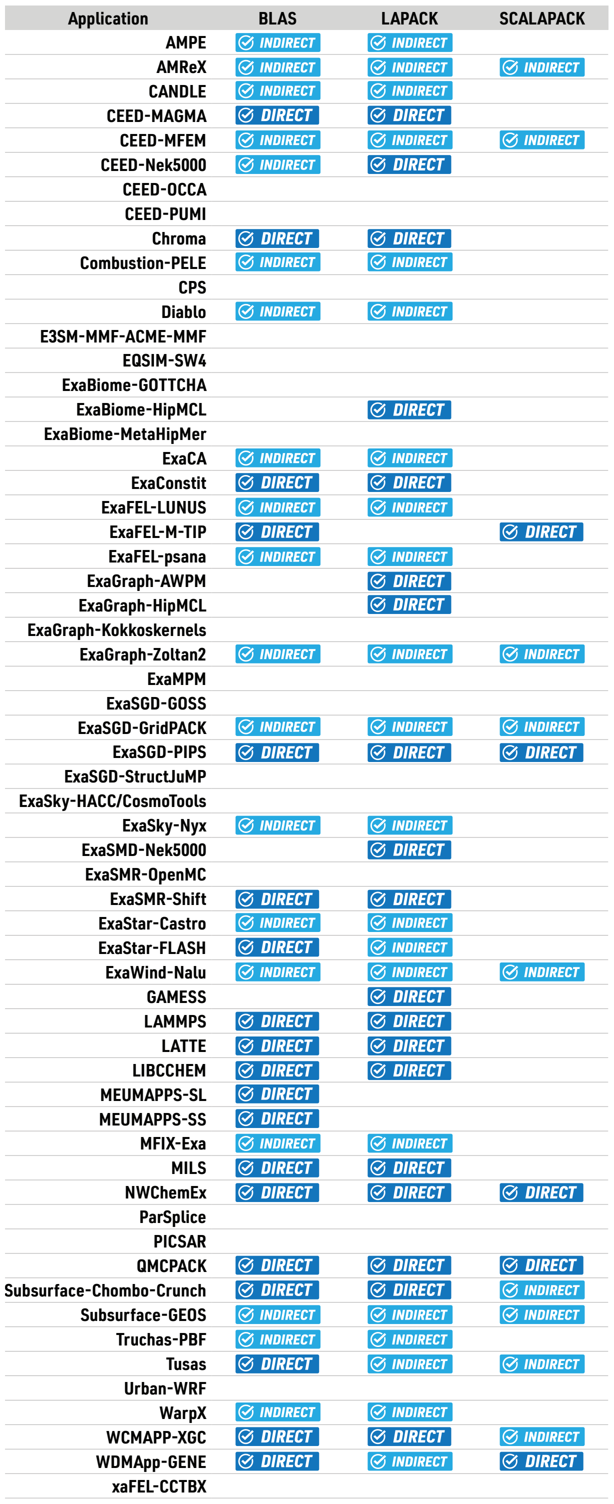
1.1 Significance of SLATE

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 on 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



Application	BLAS	LAPACK	SCALAPACK
AMPE	✓ INDIRECT	✓ INDIRECT	
AMReX	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
CANDLE	✓ INDIRECT	✓ INDIRECT	
CEED-MAGMA	✓ DIRECT	✓ DIRECT	
CEED-MFEM	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
CEED-Nek5000	✓ INDIRECT	✓ DIRECT	
CEED-OCCA			
CEED-PUMI			
Chroma	✓ DIRECT	✓ DIRECT	
Combustion-PELE	✓ INDIRECT	✓ INDIRECT	
CPS			
Diablo	✓ INDIRECT	✓ INDIRECT	
E3SM-MMF-ACME-MMF			
EQSIM-SW4			
ExaBiome-GOTTCHA			
ExaBiome-HipMCL		✓ DIRECT	
ExaBiome-MetaHipMer			
ExaCA	✓ INDIRECT	✓ INDIRECT	
ExaConstit	✓ DIRECT	✓ DIRECT	
ExaFEL-LUNUS	✓ INDIRECT	✓ INDIRECT	
ExaFEL-M-TIP	✓ DIRECT		✓ DIRECT
ExaFEL-psana	✓ INDIRECT	✓ INDIRECT	
ExaGraph-AWPM		✓ DIRECT	
ExaGraph-HipMCL		✓ DIRECT	
ExaGraph-Kokkoskernels			
ExaGraph-Zoltan2	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
ExaMPM			
ExaSGD-GOSS			
ExaSGD-GridPACK	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
ExaSGD-PIPS	✓ DIRECT	✓ DIRECT	✓ DIRECT
ExaSGD-StructJuMP			
ExaSky-HACC/CosmoTools			
ExaSky-Nyx	✓ INDIRECT	✓ INDIRECT	
ExaSMD-Nek5000		✓ DIRECT	
ExaSMR-OpenMC			
ExaSMR-Shift	✓ DIRECT	✓ DIRECT	
ExaStar-Castro	✓ INDIRECT	✓ INDIRECT	
ExaStar-FLASH	✓ DIRECT	✓ INDIRECT	
ExaWind-Nalu	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
GAMESS		✓ DIRECT	
LAMMPS	✓ DIRECT	✓ DIRECT	
LATTE	✓ DIRECT	✓ DIRECT	
LIBCCHEM	✓ DIRECT	✓ DIRECT	
MEUMAPPS-SL	✓ DIRECT		
MEUMAPPS-SS	✓ DIRECT		
MFIX-Exa	✓ INDIRECT	✓ INDIRECT	
MILS	✓ DIRECT	✓ DIRECT	
NWChemEx	✓ DIRECT	✓ DIRECT	✓ DIRECT
ParSplice			
PICSAR			
QMCPACK	✓ DIRECT	✓ DIRECT	✓ DIRECT
Subsurface-Chombo-Crunch	✓ DIRECT	✓ DIRECT	✓ INDIRECT
Subsurface-GEOS	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
Truchas-PBF	✓ INDIRECT	✓ INDIRECT	
Tusas	✓ DIRECT	✓ INDIRECT	✓ INDIRECT
Urban-WRF			
WarpX	✓ INDIRECT	✓ INDIRECT	
WCMAPP-XGC	✓ DIRECT	✓ DIRECT	✓ INDIRECT
WDMApp-GENE	✓ DIRECT	✓ INDIRECT	✓ DIRECT
xaFEL-CCTBX			

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 oneMKL, 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.

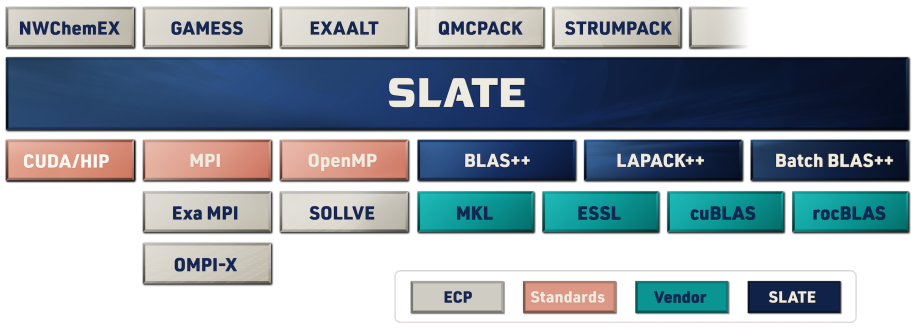


Figure 1.2: SLATE in the ECP software stack.

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 matrices stored as upper-trapezoid or lower-trapezoid. For upper matrices, tiles $A(i, j)$ are stored for $i \le j$. For lower matrices, tiles $A(i, j)$ are stored for $i \ge j$.

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 ($A_{j,i} = \bar{A}_{i,j}$).

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× to 9× smaller than ScaLAPACK's.

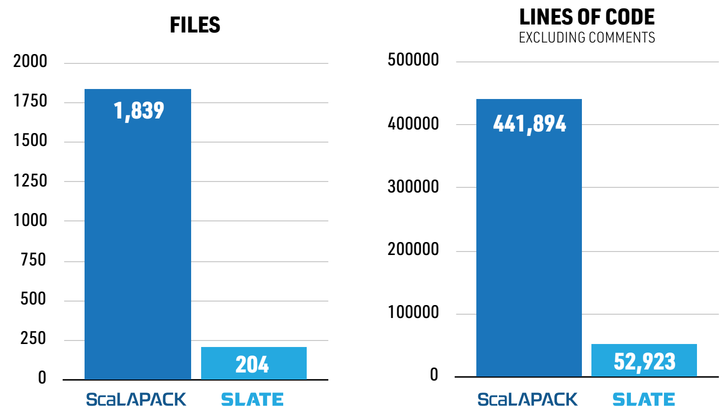


Figure 1.3: Code size comparison - ScaLAPACK vs SLATE (numbers from August 2019).

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, Σ 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 σ_i of Σ 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 σ_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 Σ 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 AA^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 Autonneand (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 ith column is the eigenvector x_i of A, and Λ 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 $B = U^H U$ 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 $\hat{A} = L^H AL$ 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 `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 `trsm`;
- Type 3: $z = Lx$ or $z = U^H x$ using `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.

(In the literature, this *three stage* algorithm is often called a *two stage* algorithm, meaning the reduction from dense to tri/bi-diagonal is two stages. The reduction to diagonal is then a separate phase of the algorithm.)

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.

Algorithm 1 Reduction to standard form (type 1) pseudocode.

```
function hegst(type, $A$, $B$)
    for $k = 1, \dots, nt$  // $nt$ = number of block rows in $A$.
        // $A(k,k) = B(k,k)^{-1} * A(k,k) * B(k,k)^{-H}$.
        hegst(type, $A(k,k)$, $B(k,k)$)
        // $A(k+1:nt,k) = A(k+1:nt,k) * B(k,k)^{H}$.
        for $m = k+1, \dots, nt$
            trsm($B(k,k)$, $A(m,k)$)
        end
        // $A(k+1:nt,k) = [B(k+1:nt,k) * A(k,k)] + A(k+1:nt,k)$.
        for $m = k+1, \dots, nt$
            hemm($A(k,k)$, $B(m,k)$, $A(m,k)$)
        end
        // $A(k+1:nt,k+1:nt) = [A(k+1:nt,k) * B(k+1:nt,k)] + A(k+1:nt,k+1:nt)$.
        for $m = k+1, \dots, nt$
            for $n = k+1, \dots, nt$
                her2k($A(m,k)$, $B(m,k)$, $A(m,n)$)
            end
        end
        // $A(k+1:nt,k) = [B(k+1:nt,k) * A(k,k)] + A(k+1:nt,k)$.
        for $m = k+1, \dots, nt$
            hemm($A(k,k)$, $B(m,k)$, $A(m,k)$)
        end
        // $A(k+1:nt,k) = B(k+1:nt,k+1:nt) * A(k+1:nt,k)$.
        for $m = k+1, \dots, nt$
            for $n = k+1, \dots, nt$
                trsm($B(m,n)$, $A(m,k)$)
            end
        end
    end
    return $A$
end function
```

Algorithm 2 Reduction to standard form (type 2 or 3) pseudocode.

```
function hegst(type, $A$, $B$)
    for $k = 1, \ldots, nt$  // $nt$ = number of block rows in $A$.
        // $A(k, 1:k) = [A(k, 1:k) * B(1:k, 1:k)]$.
        for $m = 1, \ldots, k$
            for $n = 1, \ldots, k$
                trmm($B(m, n)$, $A(k, m)$)
            end
        end
        // $A(k, 1:k) = [A(k, k) * B(k, 1:k)] + A(k, 1:k)$.
        for $m = 1, \ldots, k$
            hemm($A(k, k)$, $B(k, m)$, $A(k, m)$)
        end
        // $A(1:k, 1:k) = [A(k, 1:k)^H * B(k, (1:k)^H] + A(1:k, 1:k)$.
        for $m = 1, \ldots, k$
            for $n = 1, \ldots, k$
                her2k($A(k, m)$, $B(k, m)$, $A(m, n)$)
            end
        end
        // $A(k, 1:k) = [A(k, k) * B(k, 1:k)] + A(k, 1:k)$.
        for $m = 1, \ldots, k$
            hemm($A(k, k)$, $B(k, m)$, $A(k, m)$)
        end
        // $A(k, 1:k) = [B(k, k)^H * A(k, 1:k)]$.
        for $m = 1, \ldots, k$
            trmm($B(k, k)$, $A(k, m)$)
        end
        // $A(k, k) = B(k, k)^H * A(k, k) * B(k, k)$.
        hegst(type, $A(k, k)$, $B(k, k)$)
    end
    return $A$
end function
```

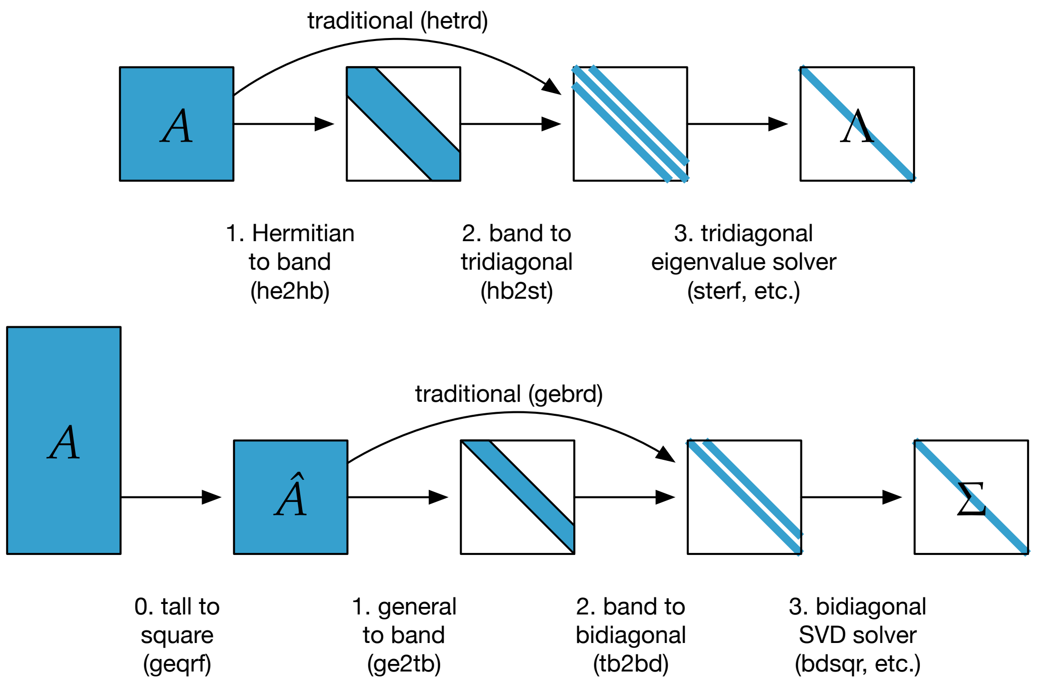

traditional (hetrd)
A
Λ
1. Hermitian to band (he2hb)
2. band to tridiagonal (hb2st)
3. tridiagonal eigenvalue solver (sterf, etc.)
A
traditional (gebrd)
$\hat{A}$
Σ
0. tall to square (geqrf)
1. general to band (ge2tb)
2. band to bidiagonal (tb2bd)
3. bidiagonal SVD solver (bdsqr, etc.)

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

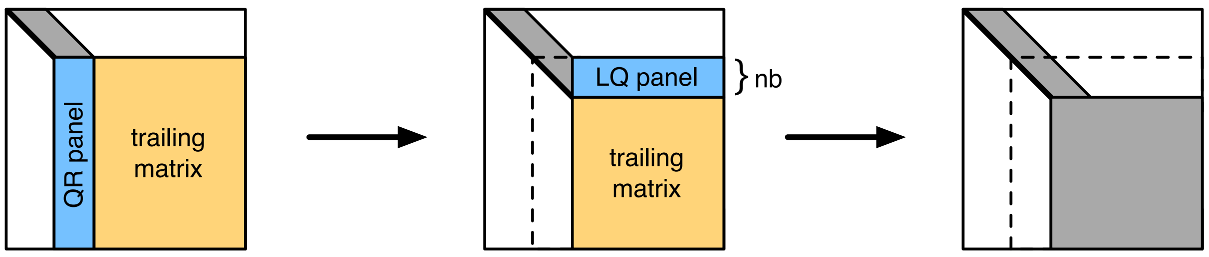


Figure 2.2: One panel of the first stage reduction to band form.

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 rows is the resulting matrix bandwidth, n_b.

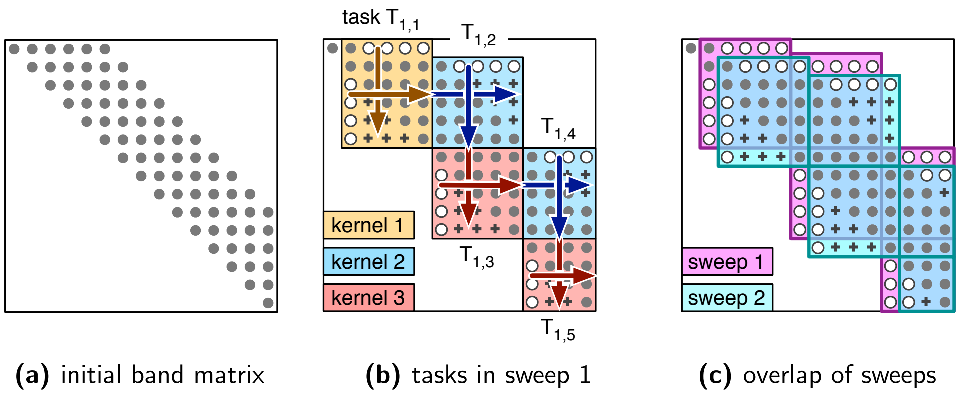


Figure 2.3: Bulge-chasing algorithm. "o" indicates eliminated elements; "+" indicates fill. Arrows show application of Householder reflector on left ($\rightarrow$), which update a block row, and on right ($\downarrow$), which update a block column.

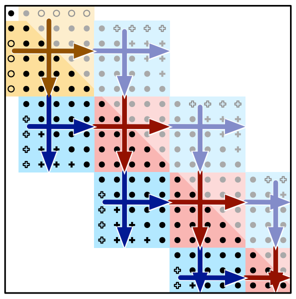


Figure 2.4: Hermitian bulge-chasing algorithm. Only the lower triangle is accessed; the upper triangle is known implicitly by symmetry.

The second stage reduces the band form to the final bidiagonal form using a bulge chasing technique. It involves $6n_bn^2$ 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 Section 2.4) 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, updating only the first block row.

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 Section 2.4. 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 Hermitian to Hermitian band reduction (he2hb)

2.5.1 Single node

Colon notation $i : k = i, \ldots, k$ includes both i and k as in Matlab (unlike Python).

Algorithm 3 Hermitian to Hermitian band (he2hb).

```
for $k = 0 : nt - 1$
    // Denote $k$-th trailing sub-matrix $A_k = A_{k+1:nt-1,\, k+1:nt-1}$
    // Panel, Householder vectors $V$, and workspace $W$ are block columns,
    // $kt \times 1$ tiles for $kt = nt - k - 1$.
    Panel factorization: $QR = A_{k+1:nt-1,\, k}$ where $Q = I - VTV^H$
    $W = A_k V$                                          hemm, $kt \times kt \cdot kt \times 1$
    $W = WT = (A_k V)T$                                  trmm, right, $kt \times 1 \cdot 1 \times 1$
    // Symmetric update
    // $X$ is 1-by-1 Hermitian tile ("TVAVT" in code)
    $X = V^H W = V^H (A_k VT)$                           inner-product gemm, $1 \times kt \cdot kt \times 1$
    $X = T^H X = T^H (V^H A_k VT)$                       trmm, left, $1 \times 1 \cdot 1 \times 1$
    $Y = W - \frac{1}{2} VX = A_k VT - V(T^H V^H A_k VT)$  gemm, $kt \times 1 \cdot 1 \times 1$
    $A_k = A_k - VY^H - YV^H$                            her2k, $kt \times 1 \cdot 1 \times kt$
end
```

Derivation of Hermitian 2-sided update.

$$\begin{aligned}
A &= Q^H A Q \\
&= (I - VT^H V^H) A (I - VTV^H) \\
&= (I - VT^H V^H)(A - AVTV^H) \\
&= (I - VT^H V^H)(A - WV^H) \\
&= A - WV^H - VT^H V^H A^* + VT^H V^H W V^H \\
&= A - WV^H - VW^H + VT^H V^H W V^H \\
&= A - (VW^H - \tfrac{1}{2} VT^H V^H W V^H) - (WV^H - \tfrac{1}{2} VT^H V^H W V^H) \\
&= A - V(W^H - \tfrac{1}{2} T^H V^H W V^H) - (W - \tfrac{1}{2} VT^H V^H W) V^H \\
&= A - V(W - \tfrac{1}{2} V (T^H V^H W)^*)^H - (W - \tfrac{1}{2} VT^H V^H W) V^H \\
&= A - VY^H - YV^H
\end{aligned}$$

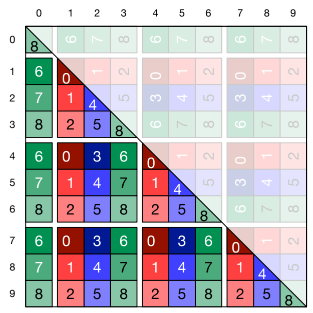
where

$$\begin{aligned}
W &= AVT, \\
Y &= W - \tfrac{1}{2} V (T^H V^H W) = W - \tfrac{1}{2} V (T^H V^H AVT).
\end{aligned}$$

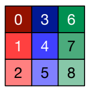
A^* is where we use that $A = A^H$. Note that $X = T^H V^H W = T^H V^H AVT$ is also Hermitian.

2.5.2 Multi-node

For a distributed, multi-node alogrithm, we use a 2D block-cyclic distribution, shown in Figure 2.5. Each tile is labeled with its node (MPI rank). For later illustration purposes, we start the sub-matrix at the 3rd block row and column, so the first tile is on node 8 instead of node 0. Tiles in the lower triangle are stored. Tiles in the upper triangle are not stored and are known by symmetry; the rotated labels indicate the node where the symmetric tile is stored. Nodes on the diagonal of the MPI process grid (here, $\{0, 4, 8\}$) we call *diagonal nodes*.



(a) Sub-matrix starting at 3rd block row and column.



(b) MPI nodes (ranks) in 3×3 process grid.

Figure 2.5: 2D block-cyclic distribution.

Summarizing Algorithm 3, there are 4 basic steps in each iteration:

$$QR = A_{\text{panel}} \qquad \text{panel factorization with } Q = I - VTV^H, \tag{2.1}$$

$$W = A_k VT \qquad \text{hemm, trmm} \tag{2.2}$$

$$Y = W - \tfrac{1}{2} V (T^H V^H W) \qquad \text{gemm, trmm, gemm} \tag{2.3}$$

$$A_k -= VY^H + Y^H V \qquad \text{her2k} \tag{2.4}$$

We use CAQR for a multi-node algorithm. In each panel, there is a local QR factorization within each node, giving a local block Householder reflector, then a tree reduction of resulting R_i triangles, giving small coupling Householder reflectors. For a $p \times p$ grid, each panel has p local QR factorizations, except the last p panels have fewer local panels. For $p = 3$, this results in block Householder vectors V_a, V_b, and V_c. For column 0 in Figure 2.5, V_a is computed on node 6 (dark green), V_b on node 7 (medium green), and V_c on node 8 (light green).

Computing $W = AV$ (hemm)

After the panel factorization, we start the trailing matrix update by applying Q_a on the left and right to update A_k. We compute $W_a = AV_aT_a$, and Y_a by (2.3), then update A_k $-= V_aY_a^H + Y_aV_a^H$. Figure 2.6 depicts $W_a = AV_a$, which involves nodes $\{0, 1, 2, 3, 6\}$, with diagonal node 0. Tiles that have 2 colors (blue/red or green/red) are computed as partial sums on 2 nodes, then sum-reduced on both nodes, although for some edge tiles only 1 node contributes. Diagonal node 0 (dark red) has only local contributions. Blank tiles in V_a are zero, which eliminates contributions from the whited-out regions of A_k. This yields

$$W_i = \sum_{j \,\in\, \text{panel\_rank\_rows}} A_{ij}V_j, \quad \text{for } i = k+1, \ldots, nt-1, \tag{2.5}$$

where `panel_rank_rows` $= \{1, 4, 7\}$ is a list of non-zero block-rows in V_a, which are local block-rows on node 6. For a tile A_{ij} in the upper triangle of A_k, we conjugate-transpose the corresponding tile in the lower triangle, A_{ji}^H, so (2.5) becomes

$$W_i = \sum_{\substack{j \,\in\, \text{panel\_rank\_rows}, \\ i \geq j \ (A_{ij} \text{ in lower})}} A_{ij}V_j + \sum_{\substack{j \,\in\, \text{panel\_rank\_rows}, \\ i < j \ (A_{ji} \text{ in lower})}} A_{ji}^H V_j. \tag{2.6}$$

The first of these sums is done on the node where the local rows matches `panel_rank_rows`; the second sum is done on the node where the local columns matches `panel_rank_rows`. For diagonal nodes, the local rows and local columns are the same, so the entire sum is local. For diagonal tiles, $A_{jj}V_j$ is a hemm instead of a gemm. Each node computes and uses W_i where i is in its local rows or its local columns.

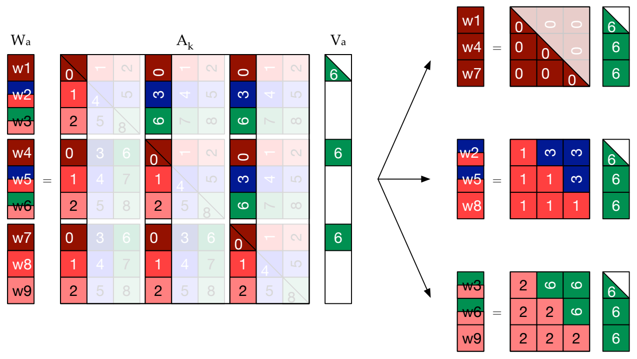
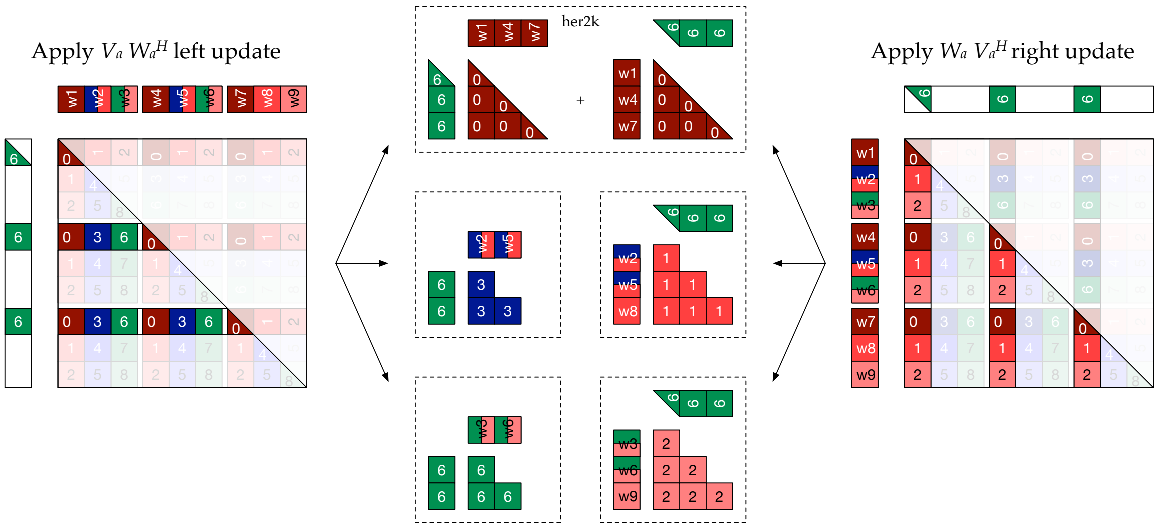


Figure 2.6: Hermitian matrix multiply $W_a = AV_a$ (hemm). Tiles that have 2 colors (blue/red or green/red) are computed as partial sums on 2 nodes, then reduced.

Computing $A -= VW^H + WV^H$ (her2k)

Figure 2.7 shows the two products $V_a W_a^H$ and $W_a V_a^H$. Due to zero blocks in V_a, the whited-out rows and columns of A_k are not updated, and we need to compute blocks only in the lower triangle, as blocks in the upper triangle are the same by symmetry. Diagonal node 0 (dark red) has non-zero blocks in both products and can be updated using a standard Hermitian rank $2k$ (`her2k`) operation. Nodes 3 (blue) and 6 (green) have non-zero blocks only in $V_a W_a^H$, which comes from applying Q_a on the left, while nodes 1 (red) and 2 (light red) have non-zero blocks only in $W_a V_a^H$, which comes from applying Q_a on the right. Thus we have a custom routine, `her2k_offdiag_ranks`, that applies either a left or a right update as needed. (We keep the `her2k` name, as the overall operation is a rank $2k$ update, even though for individual tiles it is only rank k.)



Apply V_a W_a^H left update
her2k
Apply W_a V_a^H right update
w1
w2
w3
w4
w5
w6
w7
w8
w9
+

Figure 2.7: Hermitian rank $2k$ (her2k) update, $A_k -= V_a W_a^H + W_a V_a^H$.

Applying Q_b

After applying Q_a, we apply $Q_b = I - V_b T_b V_b^H$, as shown in Figures 2.8 and 2.9. This largely follows the same structure as applying Q_a, but involves nodes $\{1, 3, 4, 5, 7\}$, with diagonal node 4.

When applying Q_a on the right previously, node 1 used W_2, W_5, W_8, corresponding to the local rows on node 1. Here, applying Q_b on the left, node 1 uses W_1, W_4, W_7, corresponding to the local columns on node 1. This pattern holds for all the nodes: a left update uses W_i for i in the node's local rows; a right update uses W_i for i in the node's local columns. In a symmetric update the local rows and local columns are the same.

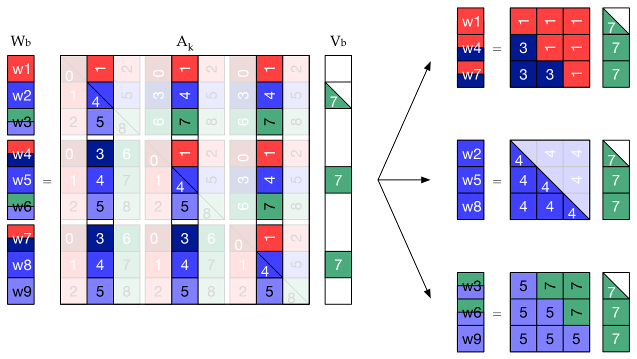


Figure 2.8: Hermitian matrix multiply $W_b = AV_b$ (hemm).

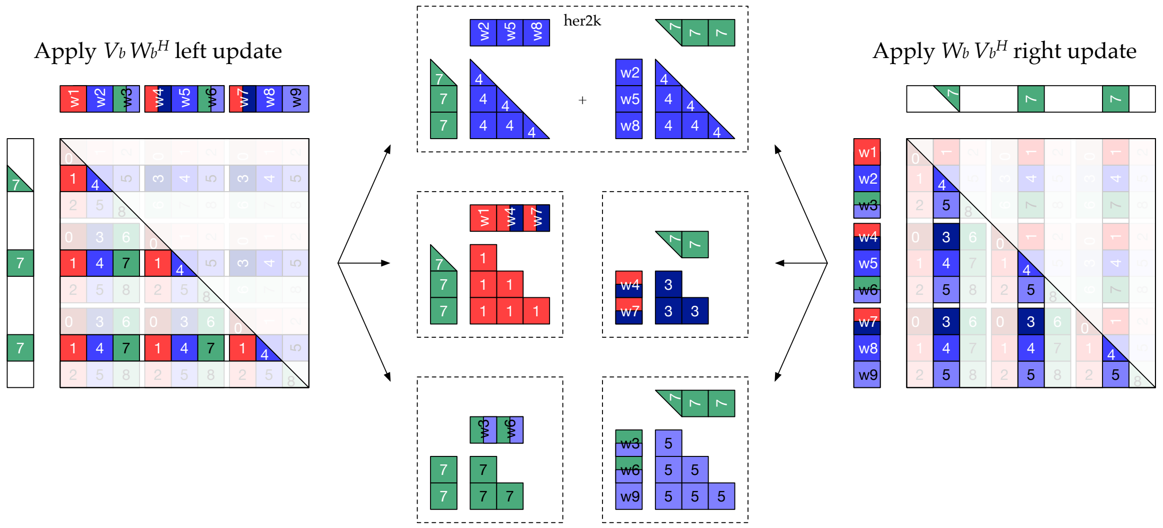


Figure 2.9: Hermitian rank $2k$ (her2k) update, $A_k \mathrel{-}= V_b W_b^H + W_b V_b^H$.

Applying Q_c

After applying Q_a and Q_b, we apply $Q_c = I - V_c T_c V_c^H$, as shown in Figures 2.10 and 2.11. This largely follows the same structure as applying Q_a and Q_b, but involves nodes $\{2, 5, 6, 7, 8\}$, with diagonal node 4.

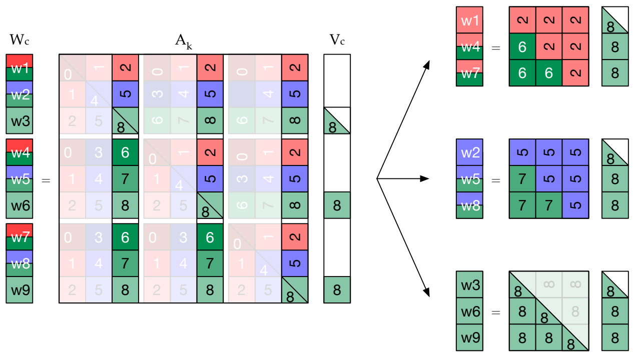


Figure 2.10: Hermitian matrix multiply $W_c = AV_c$ (hemm).

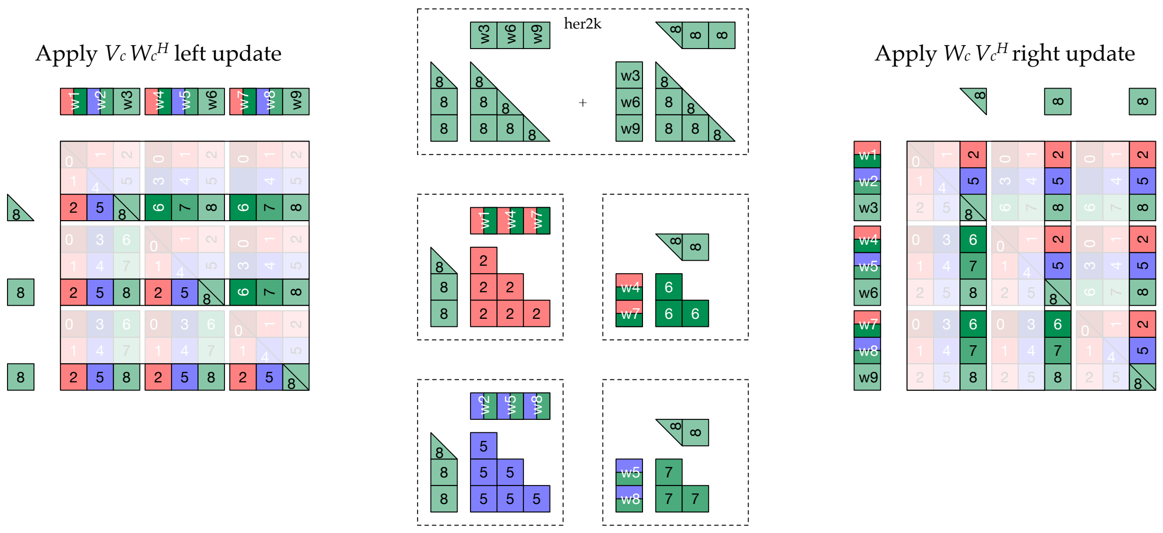
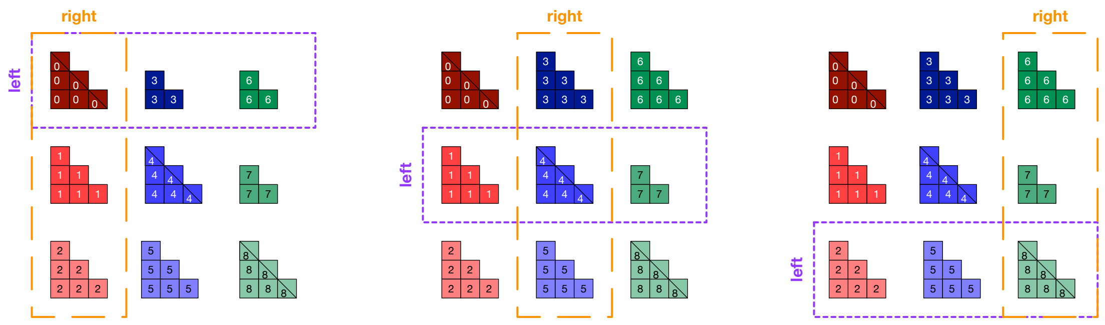


Figure 2.11: Hermitian rank $2k$ (her2k) update, $A_k \mathrel{-}= V_c W_c^H + W_c V_c^H$.

Participating nodes

Examining the MPI process grid, we see that each V updates one process row and one process column. The diagonal node at the intersection of that row and column does a 2-sided update (`her2k`), while nodes in the process row do a left update, and nodes in the process column do a right update, as shown in Figure 2.12. Since nodes 5 and 7 don't participate in Q_a, they can immediately proceed with applying Q_b. Similarly, all diagonal nodes, $\{0, 4, 8\}$, can be updated at the start.



(a) Applying Q_a updates MPI row and column 1.

(b) Applying Q_b updates MPI row and column 2.

(c) Applying Q_c updates MPI row and column 3.

Figure 2.12: Nodes and tiles that are updated by each Q_r from panel.

Multi-node Algorithm

Algorithm 4 Hermitian to Hermitian band (he2hb).

```
for $k = 0 : nt - 1$
    // Denote $k$-th trailing sub-matrix as $A_k = A_{k+1:nt-1,\, k+1:nt-1}$.
    // Panel, Householder vectors $V$, and workspace $W$ are block columns,
    // $kt \times 1$ tiles for $kt = nt - k - 1$.
    local panel factorization: $QR = A_{k+1:nt-1,\, k}$ where $Q = I - VTV^H$
    triangle panel reduction
    for $r = 1 : p$ with $p$ nodes in panel
        panel_rank = panel_ranks[ r ]
        if I am in process row $r$ or column $r$ then
            $W = A_k V_r$                                  hemm, $kt \times kt \cdot kt \times 1$
            $W = W T_r = (A_k V_r) T_r$                    trmm, right, $kt \times 1 \cdot 1 \times 1$
            if I am diagonal node then
                // Symmetric update
                // $X$ is 1-by-1 Hermitian tile ("TVAVT" in code)
                $X = V_r^H W$                              inner-product gemm, $1 \times kt \cdot kt \times 1$
                $X = T_r^H X = T_r^H V_r^H A_k V_r T_r$    trmm, left, $1 \times 1 \cdot 1 \times 1$
                $Y = W - \frac{1}{2} V_r X$                gemm, $kt \times 1 \cdot 1 \times 1$
                $A_k = A_k - V_r Y^H - Y V_r^H$            her2k, $kt \times 1 \cdot 1 \times kt$
            else
                // Offdiagonal update
                $A_k$ −= $V_r W^H$ or $A_k$ −= $W V_r^H$   her2k_offdiag_ranks, $kt \times 1 \cdot 1 \times kt$
            end
        end  // if in process row/col
    end  // for r
    apply update from triangle panel reduction (hettmqr)
end  // for k
```

2.6 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.

2.6.1 Eigenvectors of tridiagonal matrix

Once the tridiagonal reduction is achieved, the implicit QR eigensolver `steqr2` calculates the eigenvalues and optionally its associated eigenvectors of the condensed matrix structure. In

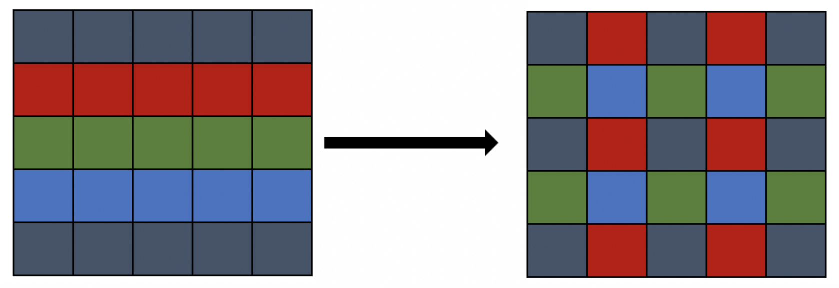


Figure 2.13: Redistribute 1D block row cyclic distributed matrix using 4×1 grid into a 2D block cyclic distribution using 2×2 grid.

SLATE (and ScaLAPACK), the `steqr2` is a modified version of the LAPACK routine `steqr` which allows each process to perform updates on the distributed matrix Q_2, and achieve parallelization during this step.

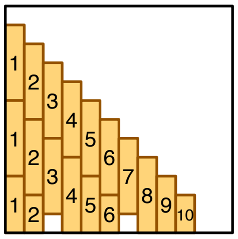
Algorithm 5 shows the call to the tridiagonal eigensolver `steqr2`. First, a matrix to store the eigenvectors $Q_{3,1D}$ of the tridaigonal matrix T is created using a 1D block row cyclic with a $n_p \times 1$ process grid, where n_p is the number of MPI processes. Then each process updates up to $(n/n_b)/n_p$ 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 block cyclic distribution as illustrated in Figure 2.13.

Algorithm 5 Tridiagonal Eigensolver using steqr2 pseudocode.

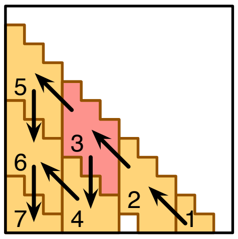
```
function steqr2($T$, $Q_3$)
    // The "1D block row cyclic" grid configuration
    $1D = n_p \times 1$
    // Compute the number of rows owned by each processor
    $nrc = (n/n_b)/n_p$
    // Build SLATE matrix $Q_{3,1D}$ using the 1-dim grid
    $Q_{3,1D}$ = Matrix($nrc, n_b, n_p, 1$)
    // Call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

2.6.2 Second stage back-transformation

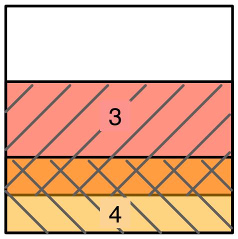
The second stage back-transformation multiplies the vectors Q_3 by Q_2 from the second stage reduction from band to tridiagonal form ("bulge chasing"), to form $Q_2 Q_3$. SLATE uses a distributed version of the scheme developed by [12]. The Householder vectors generated during the bulge chasing (Figure 2.4) are stored in a matrix V, shown in Figure 2.14. Conceptually, the



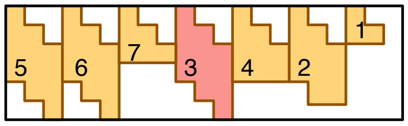
(a) Householder vectors, numbered by sweep.



(b) Dependencies between blocks of vectors.



(c) Application of block reflectors 3 and 4 overlaps.



(d) Blocks stored in packed order.

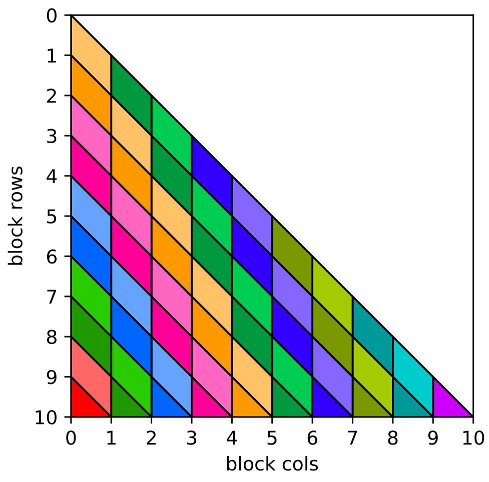
Figure 2.14: Second stage back transformation, with V block size $j_b = 3$ vectors. Block reflector 3 is highlighted to show overlap.

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 Section 2.6.2, to form block Householder reflectors, $H_r = I - V_r T_r V_r^H$ where V_r is the rth 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 Section 2.6.2, creating the dependencies between them shown in Section 2.6.2. These dependencies allow up to $\left\lceil \frac{mt}{2} \right\rceil$ updates to occur in parallel. Figure 2.15 shows these blocks and the corresponding tasks for a 10×10 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. The routine `unmtr_hb2st`, outlined in Algorithm 6, applies Q to a matrix C; for eigenvectors, $C = Q_3$. Application of each H_r becomes a single task, with dependencies on the two rows it updates, row$[i]$ and row$[i+1]$. The parallelism in Section 2.6.2 occurs automatically based on these dependencies. Within a row of C, updating each tile is independent, so we can use nested parallelism in the `parallel for` loop.

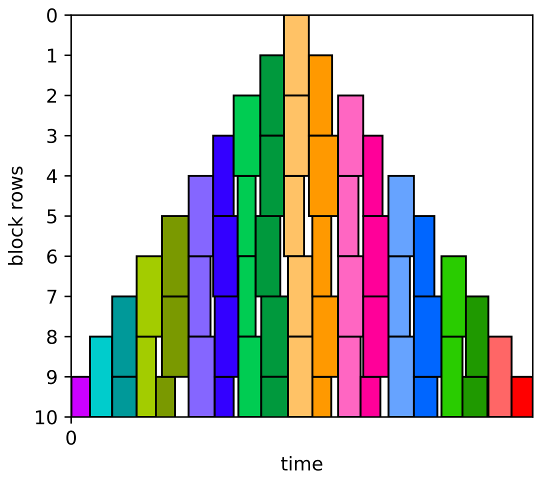
In SLATE, each parallelogram block V_r is $2n_b \times n_b$. To ease computation, instead of storing blocks in a lower triangular matrix (Section 2.6.2), each block is stored as one $2n_b \times n_b$ tile, with explicit zeros in the upper and lower triangular areas, as shown in Section 2.6.2. This allows us, for instance, to use LAPACK's `larft` function to compute T_r from V_r, and to use `gemm` instead of `trmm`. Normally, V_r has unit diagonal. SLATE stores the Householder τ values on the diagonal of V_r. During computation, the diagonal is set to 1's, and the τ values are restored afterwards.

2.6.3 First stage back-transformation

The first stage back-transformation multiplies the vectors $(Q_2 Q_3)$ by Q_1 from the first stage reduction to band, to form $X = Q_1(Q_2 Q_3)$. The routine `unmtr_he2hb` applies Q_1 or Q_1^H on the left or right of a matrix C, which is then overwritten by $Q_1 C$, $Q_1^H C$, $C Q_1$, or $C Q_1^H$. For eigenvectors, we need only the left, no-transpose case with $C = Q_2 Q_3$, to form the eigenvectors $X = Q_1(Q_2 Q_3)$. It is essentially identical to applying Q from a QR factorization, but shifted by



(a) Blocks of vectors, colored by independent blocks.



(b) Simulated run showing task parallelism.

Figure 2.15: Dependencies allow up to $\lceil \frac{mt}{2} \rceil$ parallel tasks.

Algorithm 6 `unmtr_hb2st` back-transformation pseudocode. Indices are block rows/cols.

```
function unmtr_hb2st($V, C$)
    // $C$ is $mt \times nt$ block rows/cols, blocksize $n_b \times n_b$
    // $V$ is $mt(mt+1)/2$ blocks, blocksize $2n_b \times n_b$
    for $j = mt - 1$ to 0
        for $i = j$ to $mt - 1$
            task depend in, out on row[$i$] and row[$i+1$]
                $r = i - j + j \cdot mt - j(j-1)/2$
                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
```

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 `unmqr` routine that applies Q from a QR factorization.

CHAPTER 3

Divide and conquer

The solution of the tridiagonal eigenvalue decomposition (EVD) 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 $\boldsymbol{T} \in \mathbb{R}^{n \times n}$ with eigenvalue decomposition $\boldsymbol{T} = \boldsymbol{W}\boldsymbol{\Lambda}\boldsymbol{W}^T$ where $\boldsymbol{\Lambda}$ is diagonal and $\boldsymbol{W}$ is orthogonal. Split $\boldsymbol{T}$ into the rank-1 update

$$\boldsymbol{T} = \begin{bmatrix} \mathring{\boldsymbol{T}}_1 & \Theta\rho\boldsymbol{e}_k\boldsymbol{e}_1^T \\ \Theta\rho\boldsymbol{e}_1\boldsymbol{e}_k^T & \mathring{\boldsymbol{T}}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{T}_1 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}_2 \end{bmatrix} + \rho\boldsymbol{v}\boldsymbol{v}^T \quad \text{with} \quad \boldsymbol{v} = \begin{bmatrix} \boldsymbol{e}_k \\ \Theta\boldsymbol{e}_1 \end{bmatrix}. \tag{3.1}$$

where $\boldsymbol{T}_1$ is $n_1 \times n_1$, $\boldsymbol{T}_2$ is $n_2 \times n_2$, and $\Theta\rho = T_{n_1+1,n_1}$ is the off-diagonal element, with $\Theta = \pm 1$ such that $\rho > 0$ (see Section 3.1.1). $\mathring{\boldsymbol{T}}_1$ and $\boldsymbol{T}_1$ differ in only the bottom-right element, and $\mathring{\boldsymbol{T}}_2$ and $\boldsymbol{T}_2$ differ in only the top-left element:

$$\begin{aligned} \left(\boldsymbol{T}_1\right)_{n_1,n_1} &= \left(\mathring{\boldsymbol{T}}_1\right)_{n_1,n_1} - \rho, \\ \left(\boldsymbol{T}_2\right)_{1,1} &= \left(\mathring{\boldsymbol{T}}_2\right)_{1,1} - \rho. \end{aligned}$$

Recursively solve eigen decompositions for $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$, yielding

$$\boldsymbol{T}_1 = \boldsymbol{Q}_1\boldsymbol{D}_1\boldsymbol{Q}_1^T, \tag{3.2}$$

$$\boldsymbol{T}_2 = \boldsymbol{Q}_2\boldsymbol{D}_2\boldsymbol{Q}_2^T. \tag{3.3}$$

Then the eigen decomposition of $\boldsymbol{T}$ is given by

$$\begin{aligned}\boldsymbol{T} &= \boldsymbol{Q}\boldsymbol{D}\boldsymbol{Q}^T + \rho\boldsymbol{v}\boldsymbol{v}^T \\ &= \boldsymbol{Q}(\boldsymbol{D} + \rho\boldsymbol{z}\boldsymbol{z}^T)\boldsymbol{Q}^T\end{aligned}$$

with

$$\boldsymbol{Q} = \begin{bmatrix}\boldsymbol{Q}_1 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_2\end{bmatrix}, \quad \boldsymbol{D} = \begin{bmatrix}\boldsymbol{D}_1 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{D}_2\end{bmatrix}, \quad \boldsymbol{z} = \boldsymbol{Q}^T\boldsymbol{v}.$$

From the definition of $\boldsymbol{v}$, $\boldsymbol{z}$ is the last row of $\boldsymbol{Q}_1$, and Θ times the first row of $\boldsymbol{Q}_2$. Given Equation (3.2), compute the eigen decomposition

$$\boldsymbol{D} + \rho\boldsymbol{z}\boldsymbol{z}^T = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T$$

as shown in Section 3.1.2. Finally, the eigen decomposition of $\boldsymbol{T}$ is

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

3.1.1 Historical note on choice of θ

In Tisseur and Dongarra [15], for $\rho = \theta T_{n_1+1,n_1}$ (what they refer to as $\theta\beta$), scalar θ is chosen to avoid cancellation, per Dongarra and Sorensen [16]. This is shown in Algorithm 7, though some details are lacking. See http://www.netlib.org/misc/SEV/ for code. As used here, $\Theta = \theta^{-1}$, but we restrict $\Theta = \pm 1$, so $\Theta = \theta^{-1} = \theta$.

Algorithm 7 Choice of θ in Dongarra and Sorensen [16]

```
if sign($\hat{\boldsymbol{T}}_1(n_1, n_1)$) == sign($\hat{\boldsymbol{T}}_2(1, 1)$) then
    // $-\theta\beta$ has same sign as diagonal elements $\hat{\boldsymbol{T}}_1(n_1, n_1)$ and $\hat{\boldsymbol{T}}_2(1, 1)$.
    $\theta = -\,\text{sign}(\hat{\boldsymbol{T}}_1(n_1, n_1)) \cdot \text{sign}(\beta)$
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 implicitly chooses $\theta = \text{sign}(T_{n_1+1,n_1})$, since it does:

```
!! pdlaed0.f, line 148-149
!! T1_{n1, n1} -= rho
!! T2_{1,  1 } -= rho
D( I-1 ) = D( I-1 ) - ABS( E( I-1 ) )
D( I   ) = D( I   ) - ABS( E( I-1 ) )

!! pdlade0.f, line 214-215
!! Pass E( ID+N1-1 ) as rho
CALL PDLAED1( MATSIZ, N1, D( ID ), ID, Q, IQ, JQ, DESCQ, &
              E( ID+N1-1 ), WORK, IWORK( SUBPBS+1 ), IINFO )

!! pdlaed2.f, line 206
!! rho = | 2 rho | = | 2 T_{n1+1, n1} |
!!     = 2 theta T_{n1+1, n1}, theta = sign( T_{n1+1, n1} ).
!! Factor of 2 comes from normalizing [ z1, z2 ].
RHO = ABS( TWO*RHO )
```

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 $\det(\boldsymbol{A} - \lambda \boldsymbol{I}) = 0$. In this case,

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

Applying Sylvester's determinant theorem, with non-singular $\boldsymbol{X} = \boldsymbol{D} - \lambda \boldsymbol{I}$, we obtain

$$\begin{aligned}
&\det(\boldsymbol{X} + \rho \boldsymbol{z}\boldsymbol{z}^T) = 0 \\
&\det(\boldsymbol{X}\boldsymbol{X}^{-1}(\boldsymbol{X} + \rho \boldsymbol{z}\boldsymbol{z}^T)) = 0 \\
&\det(\boldsymbol{X}) \det(\boldsymbol{I} + \rho \boldsymbol{X}^{-1}\boldsymbol{z}\boldsymbol{z}^T) = 0 \\
&\det(\boldsymbol{I} + \rho (\boldsymbol{X}^{-1}\boldsymbol{z})\boldsymbol{z}^T) = 0 \quad \text{assuming } \boldsymbol{X} \text{ is non-singular} \\
&1 + \rho \boldsymbol{z}^T(\boldsymbol{X}^{-1}\boldsymbol{z}) = 0 \quad \text{Sylvester's determinant theorem.}
\end{aligned}$$

This is the *secular equation*, so called because of centuries-long secular (non-periodic) perturbations of planetary orbits (see wikipedia).

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

$$f(\lambda) = 1 + \rho \boldsymbol{z}^T(\boldsymbol{D} - \lambda \boldsymbol{I})^{-1}\boldsymbol{z} = 1 + \rho \sum_{i=1}^{n} \frac{z_i^2}{d_i - \lambda}, \tag{3.4}$$

with corresponding (unnormalized) eigenvectors

$$\boldsymbol{u}_j = (\boldsymbol{D} - \lambda_j \boldsymbol{I})^{-1}\boldsymbol{z} = \begin{bmatrix} \dfrac{z_1}{d_1 - \lambda_j} \\ \vdots \\ \dfrac{z_n}{d_n - \lambda_j} \end{bmatrix}, \quad j = 1, \ldots, n. \tag{3.5}$$

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 3.1.11.

The secular equation solver assumes, without loss of generality, that $\rho > 0$. To accomplish this, set $\Theta = \operatorname{sign}(T_{n_1+1,n_1})$, which negates $\boldsymbol{z}_2$ and ρ; see Section 3.1.1.

The secular equation solver further requires that $\boldsymbol{D}$ is sorted in ascending order, accomplished by applying permutation $\boldsymbol{P}_s$ represented by vector `isort` (`INDX`)[1] to obtain sorted $\boldsymbol{D}_s, \boldsymbol{z}_s, \boldsymbol{U}_s$ from unsorted $\boldsymbol{D}, \boldsymbol{z}, \boldsymbol{U}$:

$$\boldsymbol{P}_s(\boldsymbol{D} + \rho \boldsymbol{z}\boldsymbol{z}^T)\boldsymbol{P}_s^T = \boldsymbol{D}_s + \rho \boldsymbol{z}_s\boldsymbol{z}_s^T$$

[1]Variable names in SLATE code and ScaLAPACK code (in parenthesis), respectively.

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

$$\eta = \epsilon \left\| \boldsymbol{D} + \rho \boldsymbol{z}\boldsymbol{z}^T \right\|_2 \le \epsilon \left(\|\boldsymbol{D}\|_2 + |\rho| \left\| \boldsymbol{z}\boldsymbol{z}^T \right\|_2 \right) = \epsilon(\max_j |d_j| + |\rho|)$$

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

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

where u is unit roundoff ($\epsilon/2$) (i.e., $u =$ `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 $\boldsymbol{T}$ with eigenvector $\boldsymbol{e}_j$. More generally, when z_j is nearly zero, deflate if

$$|\rho z_j| \le \eta.$$

The second type of deflation occurs if $\boldsymbol{D}$ 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`) [1]) are nearly equal, deflate if

$$\frac{|z_i z_j| \cdot |d_i - d_j|}{\sqrt{z_i^2 + z_j^2}} \le \eta.$$

This is the off-diagonal term when applying $\boldsymbol{G}_{ij}$ to 2×2 of d_i and d_j values:

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

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

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

(Formally, $\boldsymbol{G}_{ij}$ is embedded in an $n \times n$ identity matrix.) Let $\boldsymbol{G}_d$ be the product of all Givens rotations and let $\boldsymbol{P}_d$, represented by `ideflate` (`INDXP`)[1], be the permutation for both sorting and deflating eigenvalues (i.e., $\boldsymbol{P}_d$ includes the effect of $\boldsymbol{P}_s$). Then

$$\boldsymbol{P}_d \boldsymbol{G}_d (\boldsymbol{D} + \rho \boldsymbol{z}\boldsymbol{z}^T)(\boldsymbol{P}_d \boldsymbol{G}_d)^T = \begin{bmatrix} \boldsymbol{D}_{\text{sec}} + \rho \boldsymbol{z}_{\text{sec}} \boldsymbol{z}_{\text{sec}}^T & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{\Lambda}_{\text{dfl}} \end{bmatrix} + \boldsymbol{E}, \quad \|\boldsymbol{E}\|_2 \le c\eta. \tag{3.6}$$

where

$$\boldsymbol{D}_{\text{sec}} + \rho \boldsymbol{z}_{\text{sec}} \boldsymbol{z}_{\text{sec}}^T = \boldsymbol{U}_{\text{sec}} \boldsymbol{\Lambda}_{\text{sec}} \boldsymbol{U}_{\text{sec}}^T$$

is the secular equation with its EVD, $\boldsymbol{\Lambda}_{\text{dfl}}$ are deflated eigenvalues, and $\boldsymbol{E}$ is the error. Then the EVD of (3.6) is

$$\boldsymbol{U}_d \boldsymbol{\Lambda}_d \boldsymbol{U}_d^T, \quad \text{where} \quad \boldsymbol{U}_d = \begin{bmatrix} \boldsymbol{U}_{\text{sec}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix}, \quad \boldsymbol{\Lambda}_d = \begin{bmatrix} \boldsymbol{\Lambda}_{\text{sec}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{\Lambda}_{\text{dfl}} \end{bmatrix},$$

yielding

$$\boldsymbol{D} + \rho \boldsymbol{z}\boldsymbol{z}^T = (\boldsymbol{P}_d \boldsymbol{G}_d)^T \boldsymbol{U}_d \boldsymbol{\Lambda}_d \boldsymbol{U}_d^T (\boldsymbol{P}_d \boldsymbol{G}_d) = \boldsymbol{U} \boldsymbol{\Lambda}_d \boldsymbol{U}.$$

3.1.4 Back-transformation

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

$$\begin{aligned}\boldsymbol{QU} &= \boldsymbol{Q}(\boldsymbol{P}_d\boldsymbol{G}_d)^T\boldsymbol{U}_d \\ &= \begin{bmatrix}\boldsymbol{Q}_1 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_2\end{bmatrix} \boldsymbol{G}_d^T\boldsymbol{P}_d^T \begin{bmatrix}\boldsymbol{U}_{\text{sec}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}\end{bmatrix}.\end{aligned}$$

The $\boldsymbol{G}_d^T\boldsymbol{P}_d^T$ term can be multiplied on the left into $\boldsymbol{Q}$, or on the right into $\boldsymbol{U}_d$.

First, consider multiplying on the right. This leaves the $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$ block sparsity structure in $\boldsymbol{Q}$, but $\boldsymbol{G}_d^T\boldsymbol{P}_d^T\boldsymbol{U}_d$ destroys the block sparsity structure in $\boldsymbol{U}_d$, yielding something similar to the non-deflated case, with n^3 flops:

$$\begin{aligned}\boldsymbol{QU} &= \boldsymbol{Q}\left(\boldsymbol{G}_d^T\boldsymbol{P}_d^T\boldsymbol{U}_d\right) \\ &= \begin{bmatrix}\boldsymbol{Q}_1 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_2\end{bmatrix} \left(\boldsymbol{G}_d^T\boldsymbol{P}_d^T \begin{bmatrix}\boldsymbol{U}_{\text{sec}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}\end{bmatrix}\right) \\ &= \begin{bmatrix}\boldsymbol{Q}_1 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_2\end{bmatrix} \begin{bmatrix}\boldsymbol{U}_1 \\ \boldsymbol{U}_2\end{bmatrix} \\ &= \begin{bmatrix}\boldsymbol{Q}_1\boldsymbol{U}_1 \\ \boldsymbol{Q}_2\boldsymbol{U}_2\end{bmatrix}.\end{aligned}$$

Now consider multiplying on the left, $\boldsymbol{QG}_d^T\boldsymbol{P}_d^T$. Add another permutation $\boldsymbol{P}_t$, represented by `itype` (`INDX`)[1], to retain some block sparsity structure in $\boldsymbol{Q}$. (In ScaLAPACK, the $\boldsymbol{P}_t$ permutation overwrites the $\boldsymbol{P}_s$ permutation in `INDX`.) $\boldsymbol{P}_t$ orders cols of $\boldsymbol{QG}_d^T\boldsymbol{P}_d^T$ into 4 column types:

- column type 1 are non-zero in only the top n_1 rows,
- column type 2 are non-zero in all rows ("dense"),
- column type 3 are non-zero in only the bottom n_2 rows,
- column type 4 are deflated; they may be non-zero in all rows.

$\boldsymbol{P}_t$ operates on only the first $n - n'_d$ cols of $\boldsymbol{QG}_d^T\boldsymbol{P}_d^T$ and rows of $\boldsymbol{U}_d$, leaving unaffected the last

n'_d cols or rows corresponding to deflated eigenvalues.

$$\boldsymbol{QU} = \left(\boldsymbol{QG}_d^T\boldsymbol{P}_d^T\right)\boldsymbol{U}_d \tag{3.7}$$

$$= \left(\boldsymbol{QG}_d^T\boldsymbol{P}_d^T\boldsymbol{P}_t\right)\left(\boldsymbol{P}_t^T\boldsymbol{U}_d\right) \tag{3.8}$$

$$= \begin{array}{c} n_1 \\ n_2 \\ \\ \end{array} \begin{array}{cccc|c} \{\boldsymbol{Q}_t\}_{1,1} & \{\boldsymbol{Q}_t\}_{1,2} & \boldsymbol{0} & & \{\boldsymbol{Q}_t\}_{1,4} \\ \boldsymbol{0} & \{\boldsymbol{Q}_t\}_{2,2} & \{\boldsymbol{Q}_t\}_{2,3} & & \{\boldsymbol{Q}_t\}_{2,4} \\ n_{s_1} & n_{s_2} & n_{s_3} & & n'_d \end{array} \begin{array}{cc|c} \boldsymbol{P}_t^T\boldsymbol{U}_{\text{sec}} & \boldsymbol{0} & n_{s_1}+n_{s_2}+n_{s_3} \\ \boldsymbol{0} & \boldsymbol{I} & n'_d \\ n_{s_1}+n_{s_2}+n_{s_3} & n'_d & \end{array} \tag{3.9}$$

$$= \left[\begin{array}{c|c} \{\boldsymbol{Q}_t\}_{1,1:2}\{\boldsymbol{U}_t\}_{1:2} & \{\boldsymbol{Q}_t\}_{1,4} \\ \{\boldsymbol{Q}_t\}_{2,2:3}\{\boldsymbol{U}_t\}_{2:3} & \{\boldsymbol{Q}_t\}_{2,4} \end{array}\right] \tag{3.10}$$

$$= \boldsymbol{Q}_t\boldsymbol{U}_t, \tag{3.11}$$

$$\boldsymbol{U}_t = \left[\begin{array}{cc} \{\boldsymbol{U}_t\}_1 & \boldsymbol{0} \\ \{\boldsymbol{U}_t\}_2 & \boldsymbol{0} \\ \{\boldsymbol{U}_t\}_3 & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{I} \end{array}\right], \tag{3.12}$$

where

- $\{\boldsymbol{Q}_t\}_{1,1}$ has n_{s_1} non-deflated cols from $\boldsymbol{Q}_1$ of column type 1 that are non-zero in only the top n_1 rows. These can be a non-deflated col from type 2 deflation, if both cols involved were from $\boldsymbol{Q}_1$, so the Givens rotation doesn't destroy the block structure.
- $\{\boldsymbol{Q}_t\}_{2,3}$ has n_{s_3} non-deflated cols from $\boldsymbol{Q}_2$ of column type 3 that are non-zero in only the bottom n_2 rows. Again, these can be a non-deflated col from type 2 deflation, if both cols involved were from $\boldsymbol{Q}_2$.
- $\begin{bmatrix} \{\boldsymbol{Q}_t\}_{1,2} \\ \{\boldsymbol{Q}_t\}_{2,2} \end{bmatrix}$

  has n_{s_3} non-deflated cols from either $\boldsymbol{Q}_1$ or $\boldsymbol{Q}_2$ of column type 2 that are non-zero in all n rows, which arise as the non-deflated eigenvector in type 2 deflation when one col is from $\boldsymbol{Q}_1$ and one from $\boldsymbol{Q}_2$, where the Givens rotation filled in all rows. In parallel, these include some columns from types 1 and 3, as explained below.
- $\begin{bmatrix} \{\boldsymbol{Q}_t\}_{1,4} \\ \{\boldsymbol{Q}_t\}_{2,4} \end{bmatrix}$ are cols from deflated eigenvalues (column type 4).

  No block structure is assumed. For type 2 deflation, if one col was from $\boldsymbol{Q}_1$ and one from $\boldsymbol{Q}_2$, then the Givens rotation filled in all rows, destroying the block structure.
- 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 $n'_d \le n_d$ due to restrictions on $\boldsymbol{P}_t$ permuting only locally within a node.

This yields the two gemm operations in (3.10) to compute $\boldsymbol{W} = \boldsymbol{QU}$.

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

Compared to Tisseur and Dongarra [15], matrices here are renamed from $\bar{\boldsymbol{Q}}$ to $\boldsymbol{Q}_t$, and renumbered using the column type as the second index:

$$\begin{array}{ccc} \left[\begin{array}{ccc|c} \bar{\boldsymbol{Q}}_{1,1} & \bar{\boldsymbol{Q}}_{1,2} & \boldsymbol{0} & \bar{\boldsymbol{Q}}_{1,3} \\ \boldsymbol{0} & \bar{\boldsymbol{Q}}_{2,1} & \bar{\boldsymbol{Q}}_{2,2} & \bar{\boldsymbol{Q}}_{1,3} \end{array}\right] & \longrightarrow & \left[\begin{array}{ccc|c} \{\boldsymbol{Q}_t\}_{1,1} & \{\boldsymbol{Q}_t\}_{1,2} & \boldsymbol{0} & \{\boldsymbol{Q}_t\}_{1,4} \\ \boldsymbol{0} & \{\boldsymbol{Q}_t\}_{2,2} & \{\boldsymbol{Q}_t\}_{2,3} & \{\boldsymbol{Q}_t\}_{2,4} \end{array}\right] \\ \text{Tisseur} & & \text{SLATE} \end{array}$$

3.1.5 Summary

$$
\begin{aligned}
\boldsymbol{T} &= \begin{bmatrix} \boldsymbol{T}_1 & \mathbf{0} \\ \mathbf{0} & \boldsymbol{T}_2 \end{bmatrix} + \rho \boldsymbol{v}\boldsymbol{v}^T && \text{with} \quad \boldsymbol{v} = \begin{bmatrix} \boldsymbol{e}_{n_1} \\ \boldsymbol{e}_1 \end{bmatrix}, \\
&= \boldsymbol{Q}\boldsymbol{D}\boldsymbol{Q}^T + \rho \boldsymbol{v}\boldsymbol{v}^T && \text{with} \quad \boldsymbol{Q} = \begin{bmatrix} \boldsymbol{Q}_1 & \mathbf{0} \\ \mathbf{0} & \boldsymbol{Q}_2 \end{bmatrix}, \quad \boldsymbol{D} = \begin{bmatrix} \boldsymbol{D}_1 & \mathbf{0} \\ \mathbf{0} & \boldsymbol{D}_2 \end{bmatrix}, \\
&= \boldsymbol{Q}(\boldsymbol{D} + \rho \boldsymbol{z}\boldsymbol{z}^T)\boldsymbol{Q}^T && \text{with} \quad \boldsymbol{z} = \boldsymbol{Q}^T\boldsymbol{v} = \begin{bmatrix} \{\boldsymbol{Q}_1\}_{n_1,:}^T \\ \{\boldsymbol{Q}_2\}_{1,:}^T \end{bmatrix}, \\
&= \boldsymbol{Q}(\boldsymbol{U}\Lambda\boldsymbol{U}^T)\boldsymbol{Q}^T \\
&= \boldsymbol{W}\Lambda\boldsymbol{W}^T
\end{aligned}
$$

Applying permutation $\boldsymbol{P}_s$ to sort eigenvalues in the deflation routine,

$$\boldsymbol{D}_s = \boldsymbol{P}_s\boldsymbol{D}, \qquad \boldsymbol{z}_s = \boldsymbol{P}_s\boldsymbol{z}.$$

Applying permutation $\boldsymbol{P}_d$ and Givens rotations $\boldsymbol{G}_d$ to sort and deflate eigenvalues,

$$(\boldsymbol{P}_d\boldsymbol{G}_d)(\boldsymbol{D} + \rho\boldsymbol{z}\boldsymbol{z}^T)(\boldsymbol{P}_d\boldsymbol{G}_d)^T = \begin{bmatrix} \boldsymbol{D}_{\text{sec}} + \rho\boldsymbol{z}_{\text{sec}}\boldsymbol{z}_{\text{sec}}^T & \mathbf{0} \\ \mathbf{0} & \Lambda_{\text{dfl}} \end{bmatrix} = \boldsymbol{U}_d\Lambda_d\boldsymbol{U}_d^T,$$

with

$$\boldsymbol{U}_d = \begin{bmatrix} \boldsymbol{U}_{\text{sec}} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix}, \qquad \Lambda_d = \begin{bmatrix} \Lambda_{\text{sec}} & \mathbf{0} \\ \mathbf{0} & \Lambda_{\text{dfl}} \end{bmatrix}.$$

Hence,

$$\boldsymbol{U} = (\boldsymbol{P}_d\boldsymbol{G}_d)^T\boldsymbol{U}_d.$$

Applying permutation $\boldsymbol{P}_t$ to group column types, we obtain the eigenvectors

$$
\begin{aligned}
\boldsymbol{W} &= \boldsymbol{Q}\boldsymbol{U} = \boldsymbol{Q}(\boldsymbol{P}_d\boldsymbol{G}_d)^T\boldsymbol{U}_d = \boldsymbol{Q}(\boldsymbol{P}_d\boldsymbol{G}_d)^T(\boldsymbol{P}_t\boldsymbol{P}_t^T)\boldsymbol{U}_d \\
&= \begin{bmatrix} \{\boldsymbol{Q}_t\}_{1,1} & \{\boldsymbol{Q}_t\}_{1,2} & \mathbf{0} & \{\boldsymbol{Q}_t\}_{1,4} \\ \mathbf{0} & \{\boldsymbol{Q}_t\}_{2,2} & \{\boldsymbol{Q}_t\}_{2,3} & \{\boldsymbol{Q}_t\}_{2,4} \end{bmatrix} \begin{bmatrix} \{\boldsymbol{U}_t\}_1 & \\ \{\boldsymbol{U}_t\}_2 & \\ \{\boldsymbol{U}_t\}_3 & \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix}
\end{aligned}
$$

with

$$\boldsymbol{Q}_t = \boldsymbol{Q}\boldsymbol{G}_d^T\boldsymbol{P}_d^T\boldsymbol{P}_t, \qquad \boldsymbol{U}_t = \boldsymbol{P}_t^T\boldsymbol{U}_d.$$

Note for deflation, the transpose $\boldsymbol{P}_d^T$ is applied to $\boldsymbol{Q}$, but for grouping column types, the untransposed $\boldsymbol{P}_t$ (the inverse of $\boldsymbol{P}_t^T$) is applied to $\boldsymbol{Q}$. Or in Matlab notation:

$$
\begin{aligned}
&\boldsymbol{Q}_d = \boldsymbol{Q}(:, \text{ideflate}), \\
&\boldsymbol{Q}_t(:, \text{itype}) = \boldsymbol{Q}_d, && \textit{not } \boldsymbol{Q}_t = \boldsymbol{Q}_d(:, \text{itype}).
\end{aligned}
$$

3.1.6 Examples

Notes:

- Let $n_1 = 4$, $n_2 = 4$, $n = n_1 + n_2 = 8$, $n_b = 2$, npcol = 2 (number of process columns).
- *deflate* shows the type of deflation for that column, blank for none.
- *pcolumn* shows the process column.
- *coltype* shows the column type:
  (1) Unaffected by deflation, part of $\boldsymbol{Q}_1$.
  (2) Non-deflated eigenvector from type 2 deflation that is non-zero in all rows.
  (3) Unaffected by deflation, part of $\boldsymbol{Q}_2$.
  (4) Deflated, either type 1 or deflated eigenvector in type 2 deflation.
- $Q(1,:)$ is first row of $\boldsymbol{Q}$,
  $Q(n,:)$ is last row of $\boldsymbol{Q}$. Together they serve to show the block sparsity of $\boldsymbol{Q}$.
- The decimal in D values (0.1 or 0.3) denotes the original column type (1 or 3).
- $\boldsymbol{z}$ and $\boldsymbol{Q}$ values are not realistic (e.g., not normalized), they are just for ease of tracking how values are permuted. $\boldsymbol{z} = \boldsymbol{D}/10$.

Each example will show how various vectors and matrices are affected by permutations.

In the output, ε represents an arbitrary small value that will invoke type 2 deflation.

Original The original $\boldsymbol{D}$ eigenvalues, $\boldsymbol{z}$ vector, and $\boldsymbol{Q}$ matrix of eigenvectors.

Sorted By applying permutation $\boldsymbol{P}_s$ represented by vector `isort` (`INDX`)[1], $\boldsymbol{D} = \boldsymbol{P}_s \boldsymbol{D}_0$ values are sorted. To get the $\boldsymbol{P}_s$ matrix, using Matlab notation:

```
I  = eye( n, n );
Ps = I( isort, : );
```

We will use these notations interchangeably:

$$D_0(\text{isort}) = P_s D_0,$$

$$Q(:, \text{isort}) = Q P_s^T.$$

Deflated With permutation $\boldsymbol{P}_d$ given by `ideflate` (`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 the secular equation solver. $\boldsymbol{P}_d$ includes both deflation and sorting. Columns of $\boldsymbol{Q}$ 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 column type).

Locally permuted With permutation $\boldsymbol{P}_t$ given by `itype` (`INDX`)[1],
within pcolumn 0 (dark blue), col types are sorted, e.g.: 1, 2, 2, 4;
within pcolumn 1 (light blue), col types are sorted, e.g.: 3, 3, 4, 4.

$\left[\{\boldsymbol{Q}_t\}_{1,1} \quad \{\boldsymbol{Q}_t\}_{1,2}\right]$ spans columns 1–5 (dark purple), to cover all col type 1 and 2. It also includes a couple col type 3, but this would not necessarily occur. It could also include col type 4.

$\left[\{\boldsymbol{Q}_t\}_{2,2} \quad \{\boldsymbol{Q}_t\}_{2,3}\right]$ spans columns 2–5 (light purple), to cover all col type 2 and 3. In this case, it does not include any col type 1 or 4, but it could.

The code copies $\boldsymbol{Q}_t(:, \text{itype}) = \boldsymbol{Q}(:, \text{ideflate})$, that is, $\boldsymbol{Q}_t\boldsymbol{P}_t^T = \boldsymbol{Q}\boldsymbol{P}_d^T$. Thus $\boldsymbol{Q}_t(:, \text{itype})$ is in the same order as $\boldsymbol{D}(\text{ideflate})$.

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

In this case, $\left[\bar{\bar{\boldsymbol{Q}}}_{11} \quad \bar{\bar{\boldsymbol{Q}}}_{12}\right]$ would span columns 1–3 (dark purple), to cover all col type 1 and 2. No col type 3 would be included.

$\left[\bar{\bar{\boldsymbol{Q}}}_{22} \quad \bar{\bar{\boldsymbol{Q}}}_{23}\right]$ would span columns 2–5 (light purple), to cover all col type 2 and 3. No col type 1 would be included.

Again, the code doesn't compute `igbar`. It computes `iglobal` such that $Q_G = Q_g(:, \text{iQ})$. Hence `igbar = itype( iQ )`.

3.1.7 Example 0: no deflation

Using the above setup, with no entries deflated. With no deflation, $P_t = P_d$, so $P_t^T P_d = I$ and $Q_t = Q$.

To run the example in parallel:

```
slate/test> mpirun -np 4 ./tester --dim 8 --nb 2 \
                --verbose 1 --ref n --print-precision 2 --print-width 6 \
                stedc_deflate
```

$D =$	2.10	4.10	6.10	8.10	3.30	6.30	9.30	12.30
$z =$	0.21	0.41	0.61	0.81	0.33	0.63	0.93	1.23
$Q(1,:) =$	0.21	0.41	0.61	0.81	0	0	0	0.
$Q(n,:) =$	0	0	0	0	0.33	0.63	0.93	1.23
pcolumn =	0	0	1	1	0	0	1	1
coltype =	1	1	1	1	3	3	3	3

$P_s =$ isort =	0	4	1	2	5	3	6	7
$D_s = P_s D = D($ isort $) =$	2.10	3.30	4.10	6.10	6.30	8.10	9.30	12.30
$z_s = P_s z = z($ isort $) =$	0.21	0.33	0.41	0.61	0.63	0.81	0.93	1.23
$Q(1,$ isort $) =$	0.21	0	0.41	0.61	0	0.81	0	0.
$Q(n,$ isort $) =$	0	0.33	0	0	0.63	0	0.93	1.23
pcolumn(isort) =	0	0	0	1	0	1	1	1
coltype(isort) =	1	3	1	1	3	1	3	3

$P_d =$ ideflate =	0	4	1	2	5	3	6	7
$D_d = P_d D = D($ ideflate $) =$	2.10	3.30	4.10	6.10	6.30	8.10	9.30	12.30
$D_{\text{secular}} =$	2.10	3.30	4.10	6.10	6.30	8.10	9.30	12.30
$z_d = P_d z = z($ ideflate $) =$	2.97	4.67	5.80	8.63	8.91	11.46	13.15	17.39
$z_{\text{secular}} =$	0.21	0.33	0.41	0.61	0.63	0.81	0.93	1.23
$Q(1,$ ideflate $) =$	0.21	0	0.41	0.61	0	0.81	0	0.
$Q(n,$ ideflate $) =$	0	0.33	0	0	0.63	0	0.93	1.23
pcolumn(ideflate) =	0	0	0	1	0	1	1	1
coltype(ideflate) =	1	3	1	1	3	1	3	3
coltype post-deflation =	1	1	1	1	3	3	3	3

$P_t =$ itype =	0	4	1	2	5	3	6	7
$P_t^T P_d =$	0	1	2	3	4	5	6	7
$P_t^T P_d D =$	2.10	4.10	6.10	8.10	3.30	6.30	9.30	12.30
$D_{\text{deflated}} =$	—	—	—	—	—	—	—	—
$Q_t(1,:) =$	0.21	0.41	0.61	0.81	0	0	0	0.
$Q_t(n,:) =$	0	0	0	0	0.33	0.63	0.93	1.23
$P_t^T P_d$ pcolumn =	0	0	1	1	0	0	1	1
$P_t^T P_d$ coltype =	1	1	1	1	3	3	3	3
local $P_t^T P_d$ pcolumn =	0	0	0	0	1	1	1	1
local $P_t^T P_d$ coltype =	1	1	3	3	1	1	3	3

$P_g =$ iglobal =	0	1	2	3	4	5	6	7
$P_g P_t^T P_d D =$	2.10	4.10	6.10	8.10	3.30	6.30	9.30	12.30
$P_g P_t^T P_d$ coltype =	1	1	1	1	3	3	3	3

3.1.8 Example 1: type 1 deflation — serial

Using the above setup, then set $d_0 \approx 0$, $d_4 \approx 0$, $d_7 \approx 0$, which triggers type 1 deflation for those 3 eigvals. In serial, all 3 are moved to the end of the matrix. Having no type 2 deflation, there are no coltype 2.

To run the example in serial:

```
slate/test> mpirun -np 1 ./tester --dim 8 --nb 2 --deflate '0 4 6' \
                --verbose 1 --ref n --print-precision 2 --print-width 6 \
                stedc_deflate
```

$D =$	2.10	4.10	6.10	8.10	3.30	6.30	9.30	12.30
$z =$	ε	0.41	0.61	0.81	ε	0.63	ε	1.23
$Q(1,:) =$	0.21	0.41	0.61	0.81	0	0	0	0.
$Q(n,:) =$	0	0	0	0	0.33	0.63	0.93	1.23
pcolumn $=$	0	0	0	0	0	0	0	0
coltype $=$	1	1	1	1	3	3	3	3

$P_s =$ isort $=$	0	4	1	2	5	3	6	7
$D_s = P_s D = D($ isort $) =$	2.10	3.30	4.10	6.10	6.30	8.10	9.30	12.30
$z_s = P_s z = z($ isort $) =$	ε	ε	0.41	0.61	0.63	0.81	ε	1.23
$Q(1,$ isort $) =$	0.21	0	0.41	0.61	0	0.81	0	0.
$Q(n,$ isort $) =$	0	0.33	0	0	0.63	0	0.93	1.23
pcolumn(isort) $=$	0	0	0	0	0	0	0	0
coltype(isort) $=$	1	3	1	1	3	1	3	3

$P_d =$ ideflate $=$	1	2	5	3	7	6	4	0
$D_d = P_d D = D($ ideflate $) =$	4.10	6.10	6.30	8.10	12.30	9.30	3.30	2.10
$D_{\text{secular}} =$	4.10	6.10	6.30	8.10	12.30	—	—	—
$z_d = P_d z = z($ ideflate $) =$	5.80	8.63	8.91	11.46	17.39	13.15	4.67	2.97
$z_{\text{secular}} =$	0.41	0.61	0.63	0.81	1.23	—	—	—
$Q(1,$ ideflate $) =$	0.41	0.61	0	0.81	0	0	0	0.21
$Q(n,$ ideflate $) =$	0	0	0.63	0	1.23	0.93	0.33	0.
pcolumn(ideflate) $=$	0	0	0	0	0	0	0	0
coltype(ideflate) $=$	1	1	3	1	3	4	4	4
coltype post-deflation $=$	4	1	1	1	4	3	4	3

$P_t =$ itype $=$	0	1	3	2	4	5	6	7
$P_t^T P_d =$	1	2	3	5	7	6	4	0
$P_t^T P_d D =$	4.10	6.10	8.10	6.30	12.30	9.30	3.30	2.10
$D_{\text{deflated}} =$	—	—	—	—	—	9.30	3.30	2.10
$Q_t(1,:) =$	0.41	0.61	0.81	0	0	0	0	0.21
$Q_t(n,:) =$	0	0	0	0.63	1.23	0.93	0.33	0.
$P_t^T P_d$ pcolumn $=$	0	0	0	0	0	0	0	0
$P_t^T P_d$ coltype $=$	1	1	1	3	3	4	4	4
local $P_t^T P_d$ pcolumn $=$	0	0	0	0	0	0	0	0
local $P_t^T P_d$ coltype $=$	1	1	1	3	3	4	4	4

$P_g =$ iglobal $=$	0	1	2	3	4	5	6	7
$P_g P_t^T P_d D =$	4.10	6.10	8.10	6.30	12.30	9.30	3.30	2.10
$P_g P_t^T P_d$ coltype $=$	1	1	1	3	3	4	4	4

Example 1: type 1 deflation — parallel

In parallel with 2 process columns (4 MPI ranks in 2×2 grid), entries 0 and 4 are moved to the end of process col 0, so are unfortunately included in $\{Q_t\}_{1,1:2}$ and $\{Q_t\}_{2,2:3}$, while entry 7 is moved to the end of process col 1.

To run the example in parallel:

```
slate/test> mpirun -np 4 ./tester --dim 8 --nb 2 --deflate '0 4 6' \
                --verbose 1 --ref n --print-precision 2 --print-width 6 \
                stedc_deflate
```

$D =$	2.10	4.10	6.10	8.10	3.30	6.30	9.30	12.30
$z =$	ε	0.41	0.61	0.81	ε	0.63	ε	1.23
$Q(1,:) =$	0.21	0.41	0.61	0.81	0	0	0	0.
$Q(n,:) =$	0	0	0	0	0.33	0.63	0.93	1.23
pcolumn =	0	0	1	1	0	0	1	1
coltype =	1	1	1	1	3	3	3	3

$P_s =$ isort =	0	4	1	2	5	3	6	7
$D_s = P_s D = D($ isort $) =$	2.10	3.30	4.10	6.10	6.30	8.10	9.30	12.30
$z_s = P_s z = z($ isort $) =$	ε	ε	0.41	0.61	0.63	0.81	ε	1.23
$Q(1,$ isort $) =$	0.21	0	0.41	0.61	0	0.81	0	0.
$Q(n,$ isort $) =$	0	0.33	0	0	0.63	0	0.93	1.23
pcolumn(isort) =	0	0	0	1	0	1	1	1
coltype(isort) =	1	3	1	1	3	1	3	3

$P_d =$ ideflate =	1	2	5	3	7	6	4	0
$D_d = P_d D = D($ ideflate $) =$	4.10	6.10	6.30	8.10	12.30	9.30	3.30	2.10
$D_{\text{secular}} =$	4.10	6.10	6.30	8.10	12.30	—	—	—
$z_d = P_d z = z($ ideflate $) =$	5.80	8.63	8.91	11.46	17.39	13.15	4.67	2.97
$z_{\text{secular}} =$	0.41	0.61	0.63	0.81	1.23	—	—	—
$Q(1,$ ideflate $) =$	0.41	0.61	0	0.81	0	0	0	0.21
$Q(n,$ ideflate $) =$	0	0	0.63	0	1.23	0.93	0.33	0.
pcolumn(ideflate) =	0	1	0	1	1	1	0	0
coltype(ideflate) =	1	1	3	1	3	4	4	4
coltype post-deflation =	4	1	1	1	4	3	4	3

$P_t =$ itype =	0	2	1	3	6	7	4	5
$P_t^T P_d =$	1	5	2	3	4	0	7	6
$P_t^T P_d D =$	4.10	6.30	6.10	8.10	3.30	2.10	12.30	9.30
$D_{\text{deflated}} =$	—	—	—	—	3.30	2.10	—	9.30
$Q_t(1,:) =$	0.41	0	0.61	0.81	0	0.21	0	0.
$Q_t(n,:) =$	0	0.63	0	0	0.33	0	1.23	0.93
$P_t^T P_d$ pcolumn =	0	0	1	1	0	0	1	1
$P_t^T P_d$ coltype =	1	3	1	1	4	4	3	4
local $P_t^T P_d$ pcolumn =	0	0	0	0	1	1	1	1
local $P_t^T P_d$ coltype =	1	3	4	4	1	1	3	4

$P_g =$ iglobal =	0	2	3	1	6	7	4	5
$P_g P_t^T P_d D =$	4.10	6.10	8.10	6.30	12.30	9.30	3.30	2.10
$P_g P_t^T P_d$ coltype =	1	1	1	3	3	4	4	4

3.1.9 Example 2: type 2 deflation — serial

Using the above setup, deflate pairs $\{d_0, d_5\}$ and $\{d_4, d_7\}$. The first entry of each pair is deflated (column type 4). Since d_0 is in $\boldsymbol{Q}_1$ and d_5 is in $\boldsymbol{Q}_2$, d_5 becomes column type 2. Since d_4 and d_7 are both in $\boldsymbol{Q}_2$, d_7 remains column type 3.

To run the example in serial:

```
slate/test> mpirun -np 1 ./tester --dim 8 --nb 2 --deflate '0/5 4/7' \
                --verbose 1 --ref n --print-precision 2 --print-width 6 \
                stedc_deflate
```

$D =$	2.10	4.10	6.10	8.10	3.30	2.10	9.30	3.30
$z =$	0.21	0.41	0.61	0.81	0.33	0.63	0.93	1.23
$Q(1,:) =$	0.21	0.41	0.61	0.81	0	0	0	0.
$Q(n,:) =$	0	0	0	0	0.33	0.63	0.93	1.23
pcolumn =	0	0	0	0	0	0	0	0
coltype =	1	1	1	1	3	3	3	3

$P_s =$ isort =	0	5	4	7	1	2	3	6
$D_s = P_s D = D($ isort $) =$	2.10	2.10	3.30	3.30	4.10	6.10	8.10	9.30
$z_s = P_s z = z($ isort $) =$	0.21	0.63	0.33	1.23	0.41	0.61	0.81	0.93
$Q(1,$ isort $) =$	0.21	0	0	0	0.41	0.61	0.81	0.
$Q(n,$ isort $) =$	0	0.63	0.33	1.23	0	0	0	0.93
pcolumn(isort) =	0	0	0	0	0	0	0	0
coltype(isort) =	1	3	3	3	1	1	1	3

$P_d =$ ideflate =	5	7	1	2	3	6	4	0
$D_d = P_d D = D($ ideflate $) =$	2.10	3.30	4.10	6.10	8.10	9.30	3.30	2.10
$D_{\text{secular}} =$	2.10	3.30	4.10	6.10	8.10	9.30	—	—
$z_d = P_d z = z($ ideflate $) =$	2.97	4.67	5.80	8.63	11.46	13.15	4.67	2.97
$z_{\text{secular}} =$	0.66	1.27	0.41	0.61	0.81	0.93	—	—
$Q(1,$ ideflate $) =$	0.07	0	0.41	0.61	0.81	0	0	0.20
$Q(n,$ ideflate $) =$	0.60	1.27	0	0	0	0.93	ε	-0.20
pcolumn(ideflate) =	0	0	0	0	0	0	0	0
coltype(ideflate) =	2	3	1	1	1	3	4	4
coltype post-deflation =	4	1	1	1	4	2	3	3

$P_t =$ itype =	3	4	0	1	2	5	6	7
$P_t^T P_d =$	1	2	3	5	7	6	4	0
$P_t^T P_d D =$	4.10	6.10	8.10	2.10	3.30	9.30	3.30	2.10
$D_{\text{deflated}} =$	—	—	—	—	—	—	3.30	2.10
$Q_t(1,:) =$	0.41	0.61	0.81	0.07	0	0	0	0.20
$Q_t(n,:) =$	0	0	0	0.60	1.27	0.93	ε	-0.20
$P_t^T P_d$ pcolumn =	0	0	0	0	0	0	0	0
$P_t^T P_d$ coltype =	1	1	1	2	3	3	4	4
local $P_t^T P_d$ pcolumn =	0	0	0	0	0	0	0	0
local $P_t^T P_d$ coltype =	1	1	1	2	3	3	4	4

$P_g =$ iglobal =	0	1	2	3	4	5	6	7
$P_g P_t^T P_d D =$	4.10	6.10	8.10	2.10	3.30	9.30	3.30	2.10
$P_g P_t^T P_d$ coltype =	1	1	1	2	3	3	4	4

Example 2: type 2 deflation — parallel

Lorem ipsum.

To run the example in parallel:

```
slate/test> mpirun -np 4 ./tester --dim 8 --nb 2 --deflate '0/5 4/7' \
            --verbose 1 --ref n --print-precision 2 --print-width 6 \
            stedc_deflate
```

$D =$	2.10	4.10	6.10	8.10	3.30	2.10	9.30	3.30
$z =$	0.21	0.41	0.61	0.81	0.33	0.63	0.93	1.23
$Q(1,:) =$	0.21	0.41	0.61	0.81	0	0	0	0.
$Q(n,:) =$	0	0	0	0	0.33	0.63	0.93	1.23
pcolumn $=$	0	0	1	1	0	0	1	1
coltype $=$	1	1	1	1	3	3	3	3

$P_s =$ isort $=$	0	5	4	7	1	2	3	6
$D_s = P_s D = D($ isort $) =$	2.10	2.10	3.30	3.30	4.10	6.10	8.10	9.30
$z_s = P_s z = z($ isort $) =$	0.21	0.63	0.33	1.23	0.41	0.61	0.81	0.93
$Q(1,$ isort $) =$	0.21	0	0	0	0.41	0.61	0.81	0.
$Q(n,$ isort $) =$	0	0.63	0.33	1.23	0	0	0	0.93
pcolumn(isort) $=$	0	0	0	1	0	1	1	1
coltype(isort) $=$	1	3	3	3	1	1	1	3

$P_d =$ ideflate $=$	5	7	1	2	3	6	4	0
$D_d = P_d D = D($ ideflate $) =$	2.10	3.30	4.10	6.10	8.10	9.30	3.30	2.10
$D_{\text{secular}} =$	2.10	3.30	4.10	6.10	8.10	9.30	—	—
$z_d = P_d z = z($ ideflate $) =$	2.97	4.67	5.80	8.63	11.46	13.15	4.67	2.97
$z_{\text{secular}} =$	0.66	1.27	0.41	0.61	0.81	0.93	—	—
$Q(1,$ ideflate $) =$	0.07	0	0.41	0.61	0.81	0	0	0.20
$Q(n,$ ideflate $) =$	0.60	1.27	0	0	0	0.93	ε	-0.20
pcolumn(ideflate) $=$	0	1	0	1	1	1	0	0
coltype(ideflate) $=$	2	3	1	1	1	3	4	4
coltype post-deflation $=$	4	1	1	1	4	2	3	3

$P_t =$ itype $=$	1	6	0	2	3	7	4	5
$P_t^T P_d =$	1	5	2	3	4	0	7	6
$P_t^T P_d D =$	4.10	2.10	6.10	8.10	3.30	2.10	3.30	9.30
$D_{\text{deflated}} =$	—	—	—	—	3.30	2.10	—	—
$Q_t(1,:) =$	0.41	0.07	0.61	0.81	0	0.20	0	0.
$Q_t(n,:) =$	0	0.60	0	0	ε	-0.20	1.27	0.93
$P_t^T P_d$ pcolumn $=$	0	0	1	1	0	0	1	1
$P_t^T P_d$ coltype $=$	1	2	1	1	4	4	3	3
local $P_t^T P_d$ pcolumn $=$	0	0	0	0	1	1	1	1
local $P_t^T P_d$ coltype $=$	1	2	4	4	1	1	3	3

$P_g =$ iglobal $=$	0	2	3	1	6	7	4	5
$P_g P_t^T P_d D =$	4.10	6.10	8.10	2.10	3.30	9.30	3.30	2.10
$P_g P_t^T P_d$ coltype $=$	1	1	1	2	3	3	4	4

3.1.10 Example 3: type 1 and 2 deflation — serial

Using the above setup, deflate d_3 (type 1) and pairs $\{d_1, d_4\}$ and $\{d_2, d_5\}$ (type 2).

To run the example in serial:

```
slate/test> mpirun -np 1 ./tester --dim 8 --nb 2 --deflate '3 1/4 2/5' \
                --verbose 1 --ref n --print-precision 2 --print-width 6 \
                stedc_deflate
```

$D =$	2.10	4.10	6.10	8.10	4.10	6.10	9.30	12.30
$z =$	0.21	0.41	0.61	ε	0.33	0.63	0.93	1.23
$Q(1,:) =$	0.21	0.41	0.61	0.81	0	0	0	0.
$Q(n,:) =$	0	0	0	0	0.33	0.63	0.93	1.23
pcolumn =	0	0	0	0	0	0	0	0
coltype =	1	1	1	1	3	3	3	3

$P_s =$ isort =	0	1	4	2	5	3	6	7
$D_s = P_s D = D($ isort $) =$	2.10	4.10	4.10	6.10	6.10	8.10	9.30	12.30
$z_s = P_s z = z($ isort $) =$	0.21	0.41	0.33	0.61	0.63	ε	0.93	1.23
$Q(1,$ isort $) =$	0.21	0.41	0	0.61	0	0.81	0	0.
$Q(n,$ isort $) =$	0	0	0.33	0	0.63	0	0.93	1.23
pcolumn(isort) =	0	0	0	0	0	0	0	0
coltype(isort) =	1	1	3	1	3	1	3	3

$P_d =$ ideflate =	0	4	5	6	7	3	2	1
$D_d = P_d D = D($ ideflate $) =$	2.10	4.10	6.10	9.30	12.30	8.10	6.10	4.10
$D_{\text{secular}} =$	2.10	4.10	6.10	9.30	12.30	—	—	—
$z_d = P_d z = z($ ideflate $) =$	2.97	5.80	8.63	13.15	17.39	11.46	8.63	5.80
$z_{\text{secular}} =$	0.21	0.53	0.88	0.93	1.23	—	—	—
$Q(1,$ ideflate $) =$	0.21	0.32	0.42	0	0	0.81	0.44	0.26
$Q(n,$ ideflate $) =$	0	0.21	0.45	0.93	1.23	0	−0.44	−0.26
pcolumn(ideflate) =	0	0	0	0	0	0	0	0
coltype(ideflate) =	1	2	2	3	3	4	4	4
coltype post-deflation =	1	4	4	4	2	2	3	3

$P_t =$ itype =	0	1	2	3	4	5	6	7
$P_t^T P_d =$	0	4	5	6	7	3	2	1
$P_t^T P_d D =$	2.10	4.10	6.10	9.30	12.30	8.10	6.10	4.10
$D_{\text{deflated}} =$	—	—	—	—	—	8.10	6.10	4.10
$Q_t(1,:) =$	0.21	0.32	0.42	0	0	0.81	0.44	0.26
$Q_t(n,:) =$	0	0.21	0.45	0.93	1.23	0	−0.44	−0.26
$P_t^T P_d$ pcolumn =	0	0	0	0	0	0	0	0
$P_t^T P_d$ coltype =	1	2	2	3	3	4	4	4
local $P_t^T P_d$ pcolumn =	0	0	0	0	0	0	0	0
local $P_t^T P_d$ coltype =	1	2	2	3	3	4	4	4

$P_g =$ iglobal =	0	1	2	3	4	5	6	7
$P_g P_t^T P_d D =$	2.10	4.10	6.10	9.30	12.30	8.10	6.10	4.10
$P_g P_t^T P_d$ coltype =	1	2	2	3	3	4	4	4

Example 3: type 1 and 2 deflation — parallel

Lorem ipsum.

To run the example in parallel:

```
slate/test> mpirun -np 4 ./tester --dim 8 --nb 2 --deflate '3 1/4 2/5' \
                  --verbose 1 --ref n --print-precision 2 --print-width 6 \
                  stedc_deflate
```

$D =$	2.10	4.10	6.10	8.10	4.10	6.10	9.30	12.30
$z =$	0.21	0.41	0.61	ε	0.33	0.63	0.93	1.23
$Q(1,:) =$	0.21	0.41	0.61	0.81	0	0	0	0.
$Q(n,:) =$	0	0	0	0	0.33	0.63	0.93	1.23
pcolumn =	0	0	1	1	0	0	1	1
coltype =	1	1	1	1	3	3	3	3

$P_s =$ isort =	0	1	4	2	5	3	6	7
$D_s = P_s D = D($ isort $) =$	2.10	4.10	4.10	6.10	6.10	8.10	9.30	12.30
$z_s = P_s z = z($ isort $) =$	0.21	0.41	0.33	0.61	0.63	ε	0.93	1.23
$Q(1,$ isort $) =$	0.21	0.41	0	0.61	0	0.81	0	0.
$Q(n,$ isort $) =$	0	0	0.33	0	0.63	0	0.93	1.23
pcolumn(isort) =	0	0	0	1	0	1	1	1
coltype(isort) =	1	1	3	1	3	1	3	3

$P_d =$ ideflate =	0	4	5	6	7	3	2	1
$D_d = P_d D = D($ ideflate $) =$	2.10	4.10	6.10	9.30	12.30	8.10	6.10	4.10
$D_{\text{secular}} =$	2.10	4.10	6.10	9.30	12.30	—	—	—
$z_d = P_d z = z($ ideflate $) =$	2.97	5.80	8.63	13.15	17.39	11.46	8.63	5.80
$z_{\text{secular}} =$	0.21	0.53	0.88	0.93	1.23	—	—	—
$Q(1,$ ideflate $) =$	0.21	0.32	0.42	0	0	0.81	0.44	0.26
$Q(n,$ ideflate $) =$	0	0.21	0.45	0.93	1.23	0	-0.44	-0.26
pcolumn(ideflate) =	0	0	0	1	1	1	1	0
coltype(ideflate) =	1	2	2	3	3	4	4	4
coltype post-deflation =	1	4	4	4	2	2	3	3

$P_t =$ itype =	0	1	4	2	3	6	7	5
$P_t^T P_d =$	0	4	6	7	5	1	3	2
$P_t^T P_d D =$	2.10	4.10	9.30	12.30	6.10	4.10	8.10	6.10
$D_{\text{deflated}} =$	—	—	—	—	—	4.10	8.10	6.10
$Q_t(1,:) =$	0.21	0.32	0	0	0.42	0.26	0.81	0.44
$Q_t(n,:) =$	0	0.21	0.93	1.23	0.45	-0.26	0	-0.44
$P_t^T P_d$ pcolumn =	0	0	1	1	0	0	1	1
$P_t^T P_d$ coltype =	1	2	3	3	2	4	4	4
local $P_t^T P_d$ pcolumn =	0	0	0	0	1	1	1	1
local $P_t^T P_d$ coltype =	1	2	2	4	3	3	4	4

$P_g =$ iglobal =	0	1	4	2	3	6	7	5
$P_g P_t^T P_d D =$	2.10	4.10	6.10	9.30	12.30	8.10	6.10	4.10
$P_g P_t^T P_d$ coltype =	1	2	2	3	3	4	4	4

3.1.11 Eigenvectors via Löwner Theorem

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

$$\boldsymbol{u}_j = (\boldsymbol{D} - \lambda_j \boldsymbol{I})^{-1}\boldsymbol{z} = \begin{bmatrix} \dfrac{z_1}{d_1 - \lambda_j} \\ \vdots \\ \dfrac{z_n}{d_n - \lambda_j} \end{bmatrix} \quad \text{for } j = 1, \dots, n,$$

if $d_i \approx \lambda_j$, then $z_i/(d_i - \lambda_j)$ can be inaccurate, causing a loss of orthogonality in $\boldsymbol{U}$. Instead, consider $\{\tilde{\lambda}_j\}$ as *exact* eigenvalues of the modified system $\boldsymbol{D} + \rho \tilde{\boldsymbol{z}}\tilde{\boldsymbol{z}}^T$. See derivation in Tisseur and Dongarra [15]; here each term is multiplied by -1 to match `laed4` code. Result:[2]

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

yielding (unnormalized) eigenvectors

$$\tilde{\boldsymbol{u}}_j = (\boldsymbol{D} - \tilde{\lambda}_j \boldsymbol{I})^{-1}\tilde{\boldsymbol{z}} = \begin{bmatrix} \dfrac{\tilde{z}_1}{d_1 - \tilde{\lambda}_j} \\ \vdots \\ \dfrac{\tilde{z}_n}{d_n - \tilde{\lambda}_j} \end{bmatrix} = \begin{bmatrix} \dfrac{\tilde{z}_1}{\delta_{1,j}} \\ \vdots \\ \dfrac{\tilde{z}_n}{\delta_{n,j}} \end{bmatrix} = \tilde{\boldsymbol{z}} \oslash \boldsymbol{\delta}_j \quad \text{for } j = 1, \dots, n,$$

where $\oslash$ 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 dlaed3.f:

```
      w( i ) = sign( sqrt( -w( i ) ), s( i ) )
```

where `w` (`ztilde`) contains the "first k (`nsecular`) values of the final deflation-altered $\boldsymbol{z}$-vector", per dlaed2.

[2]Tisseur has upper limit $\prod_{\substack{j=1 \\ j \neq i}}^{i-1}$; Gu and Eisenstat [19] correctly has $\prod_{\substack{j=1 \\ j \neq i}}^{n}$.

Parallelization of eigenvectors

ScaLAPACK computes n_s roots on p processes as follows:
$n_c = n/n_{\text{pcol}}$ (`klc`) is the number of eigenvalues each process column computes.
$n_r = n_c/n_{\text{prow}}$ (`klr`) is the number of eigenvalues each process row within a process column computes.
Each process computes n_r eigenvalues, calling LAPACK's `laed4` for each. (Within each pcol, prow 0 (or drow?) computes any remainder. Globally, pcol 0 (or dcol?) computes any remainder.)

After each call to `laed4`, each process updates its $\bar{\boldsymbol{z}}$ vector of partial products, multiplying one more term:

$$\bar{z}_i = \frac{\prod_{k=1}^{n_r}(d_i - \tilde{\lambda}_{k_l})}{\prod_{\substack{k=1 \\ k_g \neq i}}^{n_r}(d_i - d_{k_g})} = \frac{\prod_{k=1}^{n_r}\delta_{i,i}}{\prod_{\substack{k=1 \\ k_g \neq i}}^{n_r}(d_{k_g} - d_i)} \quad \text{for } i = 1 \ldots n,$$

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 $\bar{\boldsymbol{z}}$, similar to `MPI_Reduce`. The root node finishes the computation of $\tilde{\boldsymbol{z}}$,

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

then broadcasts $\tilde{\boldsymbol{z}}$.

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

Now, each process computes its local portion of the $\boldsymbol{U}$ matrix, per the 2D block cyclic distribution. For each column j of $\boldsymbol{U}$ that pcol p owns, each process in pcol p redundantly re-computes the corresponding eigenvalue λ_j and $\boldsymbol{\delta}_j$ vector. Each process in pcol then redundantly computes the entire vector $\boldsymbol{u}_j = \tilde{\boldsymbol{z}}_j \oslash \boldsymbol{\delta}_j$, takes its norm, and normalizes and saves just the portion that the process owns. Note this calls `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 $\boldsymbol{\delta}$ in that fashion would avoid all the redundant `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 laed4 and why that might be necessary for stability.

Alternatively, it could skip all this redundant computation by saving the $\boldsymbol{\delta}_j$ vectors and communicating them, and doing a distributed computation of column norms to normalize the vectors.

3.1.12 Cost

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

$$\boldsymbol{QU} = \begin{bmatrix} \boldsymbol{Q}_1 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_2 \end{bmatrix} \begin{bmatrix} \boldsymbol{U}_1 \boldsymbol{U}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{Q}_1 \boldsymbol{U}_1 \\ \boldsymbol{Q}_2 \boldsymbol{U}_2 \end{bmatrix}.$$

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

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

with solution

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

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 8 Main divide & conquer driver

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

3.2.2 `stedc_solve`

Main divide & conquer driver.
Corresponds to ScaLAPACK `laed0`.

Algorithm 9 Main divide & conquer driver

```
function stedc_solve($\boldsymbol{D}$, $\boldsymbol{E}$, $\boldsymbol{Q}$; workspace $\boldsymbol{W}$, $\boldsymbol{U}$)
input: $\boldsymbol{A}$ represented by diagonal $\boldsymbol{D}$ and sub-diagonal $\boldsymbol{E}$ vectors.
output: $\boldsymbol{D}$ is (unsorted) eigvals and $\boldsymbol{Q}$ is eigvecs of $\boldsymbol{A}$.
    // Tear into subproblems
    for $i = b, 2b, \ldots, n-1$ with blocksize $b$
        subtract $\rho = |E_{i-1}|$ from $D_{i-1}$ and $D_i$
    end
    // Solve subproblems
    parallel for each block-col $i = 0, b, \ldots, n$; split over MPI ranks and OpenMP threads
        // todo: In 2DBC, seems highly load imbalanced
        // —only nodes assigned diagonal tiles do work.
        if $\boldsymbol{Q}_{i,i}$ is local then
            $i_2 = i + nb - 1$
            lapack::steqr( $\boldsymbol{D}_{i:i_2}$, $\boldsymbol{E}_{i:i_2-1}$, $\boldsymbol{Q}_{i,i}$ ) or stedc
                 to solve subproblem using serial algorithm
        end
    end
    gather and bcast $\boldsymbol{D}$ to all nodes
    // Merge subproblems
    for each level in divide & conquer tree, from leaf to root
        for each pair of subproblems, indexed $i, \ldots, i_1$ and $i_1 + 1, \ldots, i_2$
            $\rho = E_{i_1}$ // abs dealt with in merge routine
            stedc_merge( $\rho$, $D_{i:i_2}$, $Q_{i:i_2,\, i:i_2}$, $W_{i:i_2,\, i:i_2}$, $U_{i:i_2,\, i:i_2}$ )
        end
    end
end function
```

3.2.3 `stedc_merge`

Merges two subproblems.
Corresponds to ScaLAPACK `laed1`.

```
function stedc_merge($\boldsymbol{D}$, $\boldsymbol{Q}$; workspace $\boldsymbol{Q}_t$, $\boldsymbol{U}$)
input: eigvals $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ in $\boldsymbol{D}$, eigvecs $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$ in $\boldsymbol{Q}$ of subproblems
output: eigvals $\boldsymbol{D}$ and eigvecs $\boldsymbol{Q}$ of merged problem
    stedc_z_vector( $\boldsymbol{Q}$, $\boldsymbol{z}$ ) gets $\boldsymbol{z}$
    stedc_deflate( $\boldsymbol{D}$, $\boldsymbol{z}$, $\boldsymbol{D}_s$, $\boldsymbol{z}_s$, $\boldsymbol{Q}$, $\boldsymbol{Q}_t$, $\boldsymbol{P}_t$ ) deflates $n_d$ eigvals, leaving $n_s$ secular eqn eigvals
    stedc_secular( $\boldsymbol{D}_s$, $\boldsymbol{z}_s$, $\boldsymbol{U}$, $\boldsymbol{P}_t$ ) solves secular equation for eigvals $\boldsymbol{D}_s$ and eigvecs $\boldsymbol{U}$
    // todo: merge $\boldsymbol{D}$ and $\boldsymbol{D}_s$
    $\boldsymbol{Q}_{1,1:2} = \{\boldsymbol{Q}_t\}_{1,1:2}\boldsymbol{U}_{1:2}$
    $\boldsymbol{Q}_{2,2:3} = \{\boldsymbol{Q}_t\}_{2,2:3}\boldsymbol{U}_{2:3}$
    Copy with permutation deflated eigvecs $\{\boldsymbol{Q}_t\}_{1:2,4}$ to $\boldsymbol{Q}$
end function
```

3.2.4 `stedc_z_vector`

Gathers onto all nodes vector $\boldsymbol{z}$ that is last row of $\boldsymbol{Q}_1$ and first row of $\boldsymbol{Q}_2$,

$$\boldsymbol{z} = \boldsymbol{Q}^T \begin{bmatrix} e_{n_1} \\ e_1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{Q}_1^T e_{n_1} \\ \boldsymbol{Q}_2^T e_1 \end{bmatrix}.$$

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).

```
function stedc_z_vector( $\boldsymbol{Q}$, $\boldsymbol{z}$ )
input: eigvecs $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$ of subproblems in $\boldsymbol{Q}$
output: vector $\boldsymbol{z}$ is last row of $\boldsymbol{Q}_1$ and first row of $\boldsymbol{Q}_2$
    for each block-col $j = 0, \dots, n_t - 1$
        if $j < n_{t1}$ then
            $i = n_{t_1} - 1$    // last block-row of $\boldsymbol{Q}_1$
        else
            $i = n_{t_1}$    // first block-row of $\boldsymbol{Q}_2$
        end
        if $\boldsymbol{Q}_{i,j}$ is local then
            copy last or first row of $\boldsymbol{Q}_{i,j}$ to $\boldsymbol{z}_{i:i+b}$
            MPI_send $\boldsymbol{z}_{i:i+b}$ to root, if rank $\neq$ root
        else if rank == root then
            MPI_recv $\boldsymbol{z}_{i:i+b}$ from source
        end
    end
    MPI_Bcast $\boldsymbol{z}$ to all ranks
end function
```

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 $\boldsymbol{Q}$ according to the column type:

- column type 1: non-deflated eigvecs from $\boldsymbol{Q}_1$
- column type 2: non-deflated eigvecs updated by deflation
- column type 3: non-deflated eigvecs from $\boldsymbol{Q}_2$
- column type 4: deflated eigvecs

Locally within each rank,

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

Corresponds to ScaLAPACK `laed2`.

Algorithm 10 Deflation, part 1

```
function stedc_deflate( $\rho$, $\boldsymbol{D}$, $\boldsymbol{D}_s$, $\boldsymbol{z}$, $\boldsymbol{z}_s$, $\boldsymbol{Q}$, $\boldsymbol{Q}_t$ )
input: $\rho$ that tore subproblems,
       eigvals $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ in vector $\boldsymbol{D}$,
       $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ in $\boldsymbol{z}$,
       eigvecs $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$ of subproblems in $\boldsymbol{Q}$.
output: $\boldsymbol{D}$ has $n_d$ deflated eigvals,
        $\boldsymbol{D}_s$ has $n_s$ non-deflated eigvals for secular equation,
        $\boldsymbol{z}_s$ of length $n_s$ is updated $\boldsymbol{z}$ vector for secular equation,
        $\boldsymbol{Q}_t$ is $n_s$ updated eigvecs permuted into $\boldsymbol{Q}_{1,1:2}$, $\boldsymbol{Q}_{2,2:3}$, $\boldsymbol{Q}_{1:2,4}$,
        another permutation?
    // LAPACK secular equation solver (laed4) requires $\rho > 0$
    if $\rho < 0$ then
        $\rho = -\rho$
        $\boldsymbol{z}_2 = -\boldsymbol{z}_2$
    end
    // $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ are normalized; re-normalize so $\|\boldsymbol{z}\|_2 = 1$
    $\rho = 2\rho$
    $\boldsymbol{z} = \boldsymbol{z}/\sqrt{2}$
    compute permutation $P_s$ to sort $\boldsymbol{D}$
    // note lamch("e") is unit roundoff $u = \epsilon/2$
    tol = $4\epsilon \max(\|\boldsymbol{D}\|_{\max}, \|\boldsymbol{z}\|_{\max})$
    if $\rho \|\boldsymbol{z}\|_{\max} <$ tol then
        $n_s = 0$ return
    end
```

Algorithm 11 Deflation, part 2

```
    // Deflate eigvals
    // A *candidate* eigval is non-negligible (not type 1), but may have type 2 deflation.
    // $j_{s1}$ is candidate eigval; initially none ($-1$). $s$ indicates sorted permutation.
    // $j_{s2}$ is current eigval under consideration.
    $j_{s1} = -1$
    for $j = 0, \ldots, n-1$
        $j_{s2} = P_s[j]$
        if $|\rho z_{j_{s2}}| <$ tol then
            store $j_{s2}$ as deflated eigval (type 1)
        else if not first candidate eigval (i.e., $j_{s1} \geq 0$) then
            generate Givens rotation $\boldsymbol{G}$ to zero first entry of $\begin{bmatrix} z_{j_{s1}} \\ z_{j_{s2}} \end{bmatrix}$
            compute $D_{j_{s1},j_{s2}}$ off-diagonal from applying $\boldsymbol{G} \begin{bmatrix} D_{j_{s1}} & 0 \\ 0 & D_{j_{s2}} \end{bmatrix} \boldsymbol{G}^T$
            if $D_{j_{s1},j_{s2}} <$ tol then
                update $\begin{bmatrix} z_{j_s1} \\ z_{j_s2} \end{bmatrix} = \boldsymbol{G} \begin{bmatrix} z_{j_s1} \\ z_{j_s2} \end{bmatrix} = \begin{bmatrix} 0 \\ \tau \end{bmatrix}$
                update columns $\boldsymbol{Q}_{j_{s1},j_{s2}} = \boldsymbol{Q}_{j_{s1},j_{s2}} \boldsymbol{G}$    // involves MPI for remote columns
                update $\begin{bmatrix} D_{j_{s1}} & \\ & D_{j_{s2}} \end{bmatrix} = \boldsymbol{G} \begin{bmatrix} D_{j_{s1}} & \\ & D_{j_{s2}} \end{bmatrix} \boldsymbol{G}^T$    // off-diag $D_{j_{s1},j_{s2}}$ is negligible
                store $j_{s1}$ as deflated eigval (type 2)
            else
                store $j_{s1}$ as non-deflated eigval
            end
            $j_{s1} = j_{s2}$    // $j_{s2}$ becomes next candidate eigval
        else
            $j_{s1} = j_{s2}$    // $j_{s2}$ becomes first candidate eigval
        end
    end
    store $j_{s1}$ as non-deflated eigval
```

Algorithm 12 Deflation, part 3 (todo)

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

3.2.6 `stedc_secular`

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

```
function stedc_secular( $\boldsymbol{D}$, $\boldsymbol{z}$, $\boldsymbol{P}_u$, $\boldsymbol{\Lambda}$, $\boldsymbol{U}$ )
input: deflation-adjusted $\boldsymbol{D}$, $\boldsymbol{z}$, global permutation $\boldsymbol{P}_u$
output: eigvals $\boldsymbol{\Lambda}$ of merged problem, eigvecs $\boldsymbol{U}$ of merged problem (before multiplying by $\boldsymbol{Q}$)
    // Compute $\boldsymbol{\Lambda}$ and modified $\tilde{\boldsymbol{z}}$.
    $\tilde{\boldsymbol{z}} = 1$
    parallel for $j = 0, \ldots, n_s - 1$; split over MPI ranks
        lapack::laed4( $\boldsymbol{D}$, $\boldsymbol{z}$, $\lambda_j$, $\boldsymbol{\delta}_j$ ) solves secular equation for $\lambda_j$ and $\boldsymbol{\delta}_j$ vector
        for $i = 0, \ldots, n_s - 1$ (todo: OpenMP parallel?)
            if $i == j$ then
                $\tilde{z}_i \mathrel{*}= \delta_{ij}$
            else
                $\tilde{z}_i \mathrel{*}= \frac{\delta_{ij}}{D_i - D_j}$
            end
        end
    end
    MPI_Allreduce $\tilde{\boldsymbol{z}}$
    fix sign $\tilde{z}_j$ to match sign $z_j$, for $j = 0, \ldots, n - 1$
    MPI_Allgather $\boldsymbol{\Lambda}$
    permute $\boldsymbol{\Lambda} = \boldsymbol{P}_u \boldsymbol{\Lambda}$
    // Compute $\boldsymbol{U}$ via Löwner theorem.
    // All processes within a process column do this computation redundantly.
    // We could avoid that by communicating $\boldsymbol{u}_j$.
    parallel for $j = 0, \ldots, n_s - 1$; split over MPI 2DBC process columns
        Re-compute secular equation (laed4) to get $\boldsymbol{\delta}_j$ vector
        $\boldsymbol{u}_j = \tilde{\boldsymbol{z}} \oslash \boldsymbol{\delta}_j$ element-wise
        $\boldsymbol{u}_j = \frac{\boldsymbol{u}_j}{\|\boldsymbol{u}_j\|}$
        Store local part of $\boldsymbol{u}_j$
    end
end function
```

3.2.7 `stedc_sort`

Sorts eigenvalues $\boldsymbol{D}$ and applies same permutation to eigenvectors $\boldsymbol{Q}$.
Corresponds to ScaLAPACK `lasrt`.

```
function stedc_sort( $\boldsymbol{D}$, $\boldsymbol{Q}$, $\boldsymbol{Q}_{\text{out}}$ )
input: eigvals $\boldsymbol{D}$, eigvecs $\boldsymbol{Q}$
output: sorted eigvals $\boldsymbol{D}$, permuted eigvecs $\boldsymbol{Q}_{\text{out}}$
    compute permutation $\boldsymbol{P}_s$ to sort eigvals.
    compute inverse permutation $\boldsymbol{P}_s^{-1}$
    for each block-col $j = 0, \ldots, n_t - 1$
        // todo: these are bad descriptions
        fill pcols[ jj ] with destination process of column $\boldsymbol{P}_s^{-1}(j + jj)$ for $jj = 0, \ldots, jb - 1$
        fill mine[ jj ] with $\boldsymbol{P}_s^{-1}(jj)$ where pcols[ jj ] == mycol
        fill pcnts[ p ] = length( where( pcnts[ jj ] == p ) ) for $p = 0, \ldots,$ npcol
        fill poffset = prefix_sum( pcnt )
        if block-col $j$ is local then
            for $jj = 0, \ldots, j_b - 1$
                $j_g = j + jj$
                $k_g = \boldsymbol{P}_s^{-1}(j + jj)$
                if local then
                    local copy $\boldsymbol{Q}(:, j_g) \to \boldsymbol{Q}_{\text{out}}(:, k_g)$
                else
                    // Pack into workspace
                    copy $Q(:, j_g) \to$ work( :, poffset($p_k$) )
                    poffset($p_k$) += 1
                end
            end
            for $p = 0, \ldots,$ npcol$-1$
                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 $jj = 0, \ldots,$ length( mine )
                $k_g = mine(jj)$
                copy work( :, jj ) to $\boldsymbol{Q}_{\text{out}}(:, k_g)$
            end
        end
    end
end function
```

CHAPTER 4

Optimization

4.1 Hermitian to Hermitian band reduction (he2hb)

4.2 Hermitian to Hermitian band reduction (he2hb)

In this section, we present the performance optimization of the Hermitian to Hermitian band reduction `he2hb`. In [20] we show the optimization technique to extend the parallelization of the different steps in `he2hb` on CPU only. Where we introduce a new internal functions of the various operations of `he2hb`, such as `internal::he2hb_hemm<HostTask>`, `internal::he2hb_trmm<Target::HostTask>`, etc. As most of the operations in `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 `he2hb`. Figure 4.1 shows the performance of `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 `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 `he2hb` performance using different number of GPUs, the new implementaion with enabling the data transfer during the panel factorization achieves up to 20%

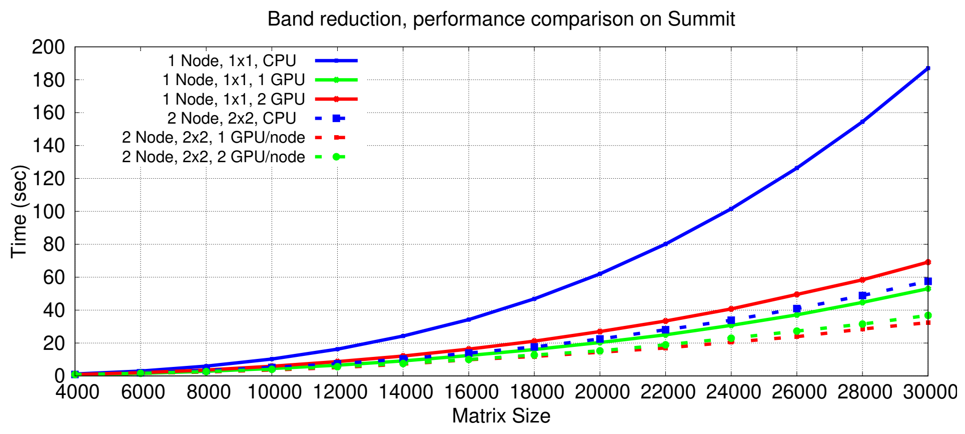


Figure 4.1: Performance of `he2hb` Using 1 and 2 nodes on Summit, 1×1 and 2×2 process grids.

improvement compared to the initial implementation.

Figure 4.4 shows the performance of `he2hb` in time and Gflops using 1 node on Summit.

4.3 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 6, 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.

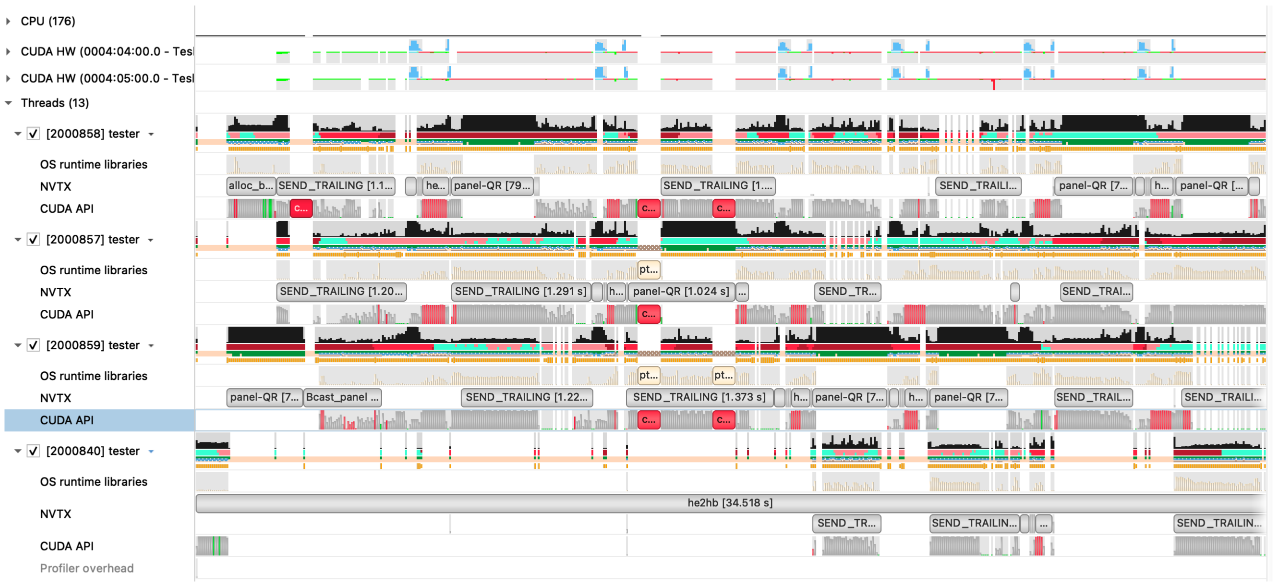


Figure 4.2: Overlap the panel factorization with subsequent data movements.

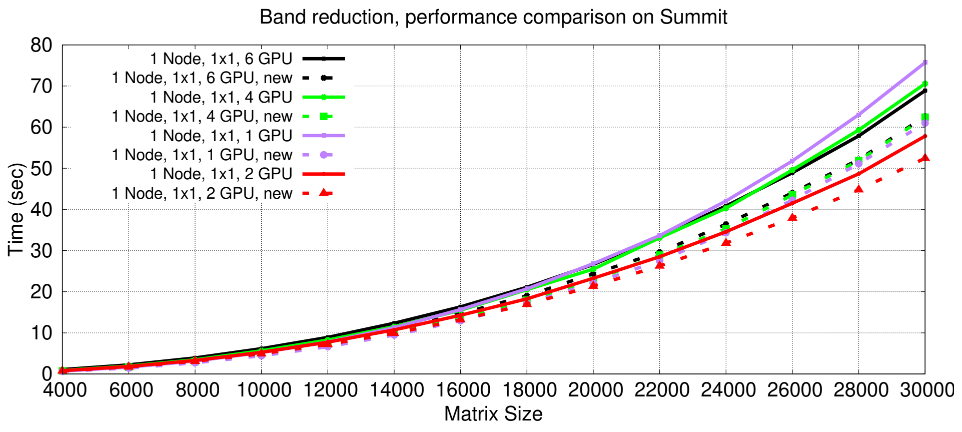
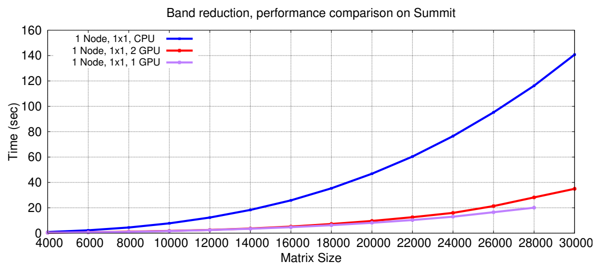
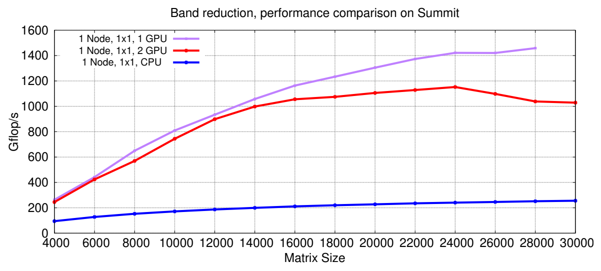


Figure 4.3: The performance impact by overlapping the panel factorization with subsequent data movements.



(a) in Time



(b) in Gflops

Figure 4.4: Performance results of Hermitian to Hermitian band reduction, using 1 node , 1×1 process grid. nb = 128,320, ib= 16, 48 for CPU, GPU tests, panel-threads=10

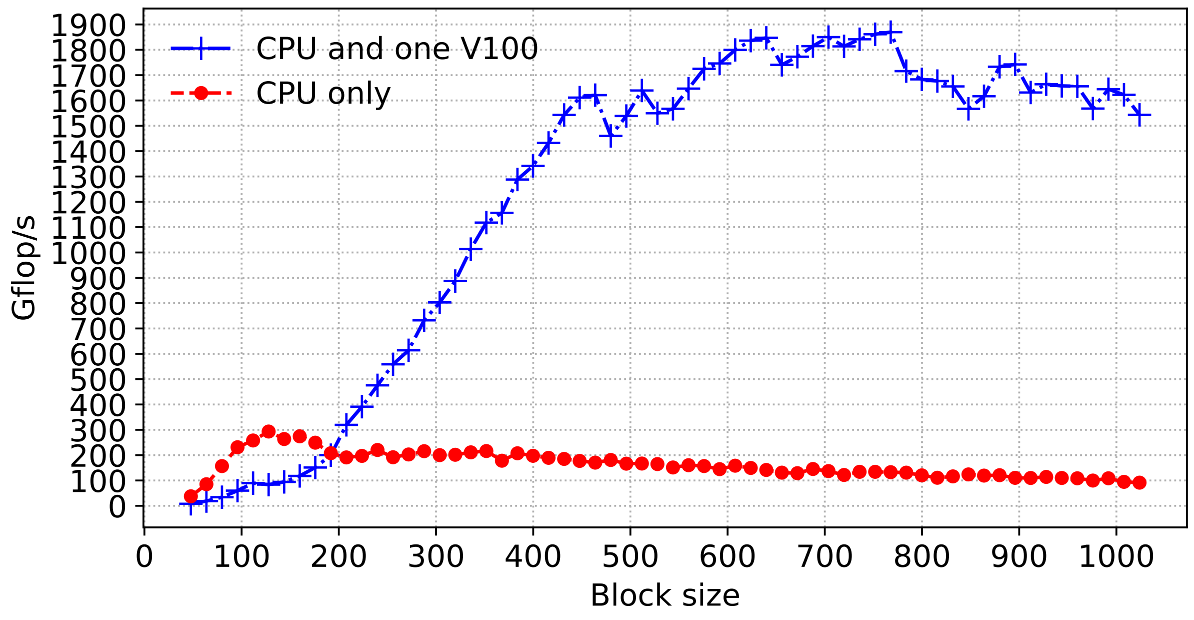


Figure 4.5: Performance of unmtr_hb2st on a host with two 20-core Intel Broadwell Xeon E5-2698 v4 CPUs and one NVIDIA V100 activated. N=16384. As seen in the figure, the device implementation provides up to 6x speedup.

CHAPTER 5

Performance

5.1 Environment

5.1.1 Hardware

Performance numbers were collected using the Summit system [1][2] 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

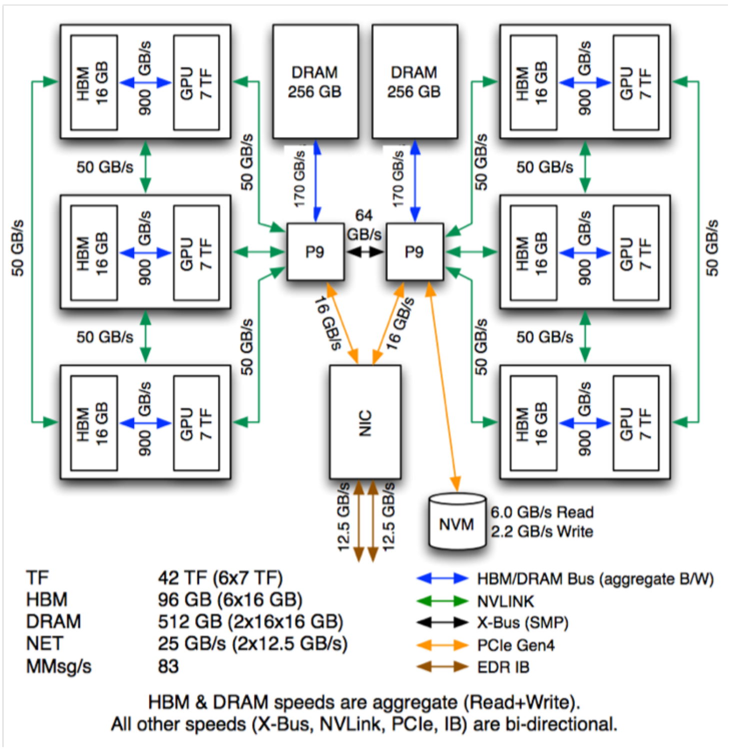


Figure 5.1: Summit node architecture.

- IBM Spectrum MPI 10.3.0.0,
- Netlib LAPACK 3.8.0, and
- Netlib ScaLAPACK 2.0.2.

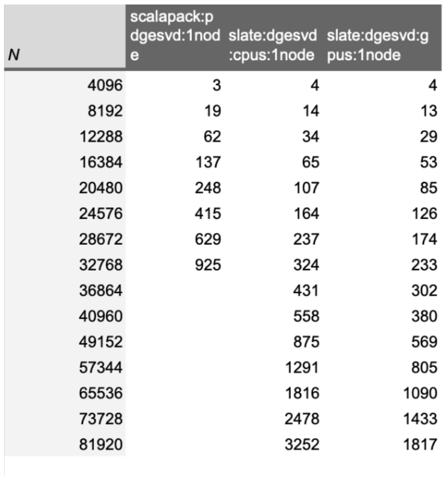
For the generalized Hermitian eigenvalues, these were updated to:

- GNU Compiler Collection (GCC) 8.1.1,
- NVIDIA CUDA 10.1.243,
- IBM Engineering Scientific Subroutine Library (ESSL) 6.1.0,
- IBM Spectrum MPI 10.3.1.2,
- Netlib LAPACK 3.8.0, and
- Netlib ScaLAPACK 2.0.2.

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



N	scalapack:pdgesvd:1node	slate:dgesvd:cpus:1node	slate:dgesvd:gpus:1node
4096	3	4	4
8192	19	14	13
12288	62	34	29
16384	137	65	53
20480	248	107	85
24576	415	164	126
28672	629	237	174
32768	925	324	233
36864		431	302
40960		558	380
49152		875	569
57344		1291	805
65536		1816	1090
73728		2478	1433
81920		3252	1817

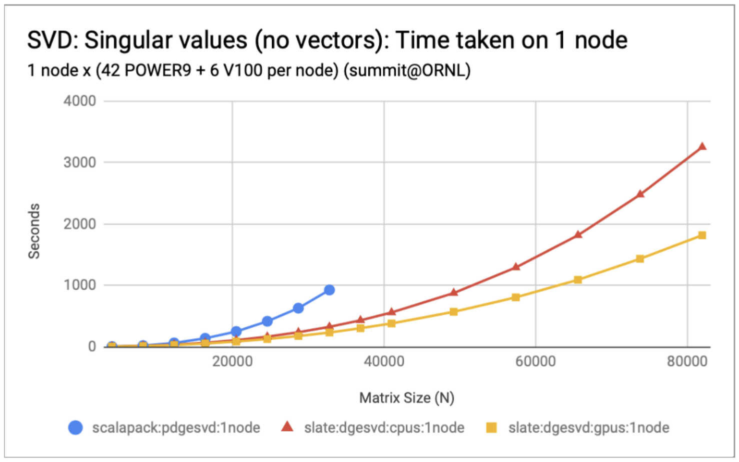


Figure 5.2: SVD performance comparison.

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.

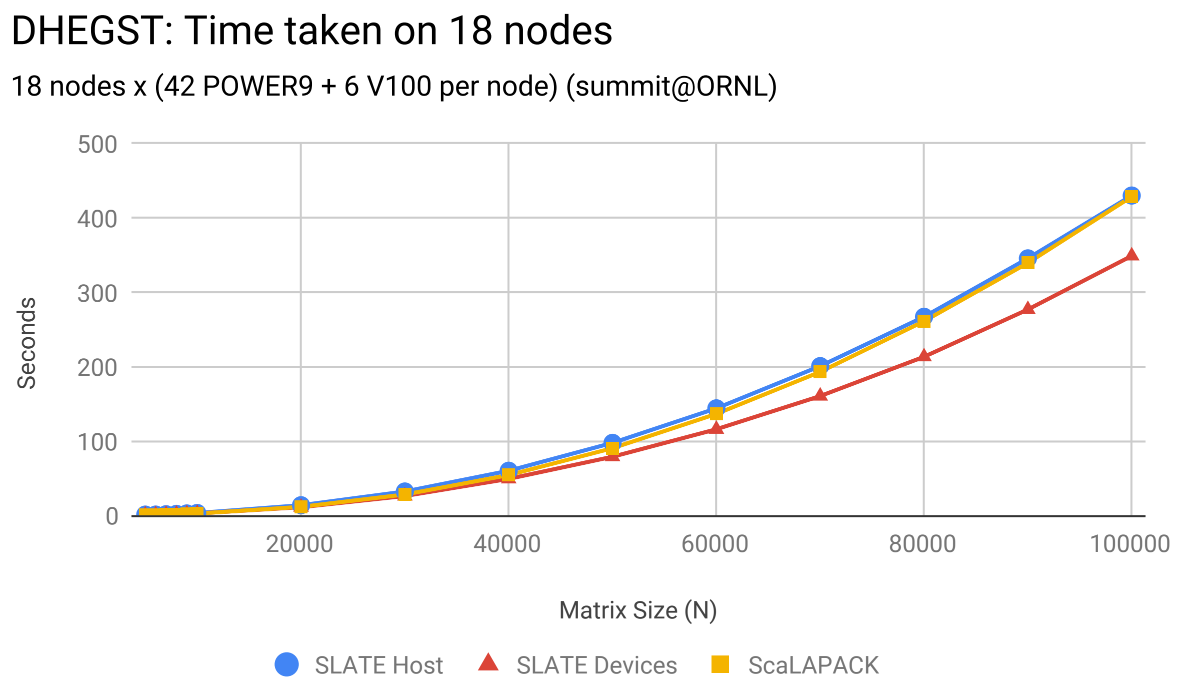


Figure 5.3: Generalized to standard eigenvalue performance comparison.

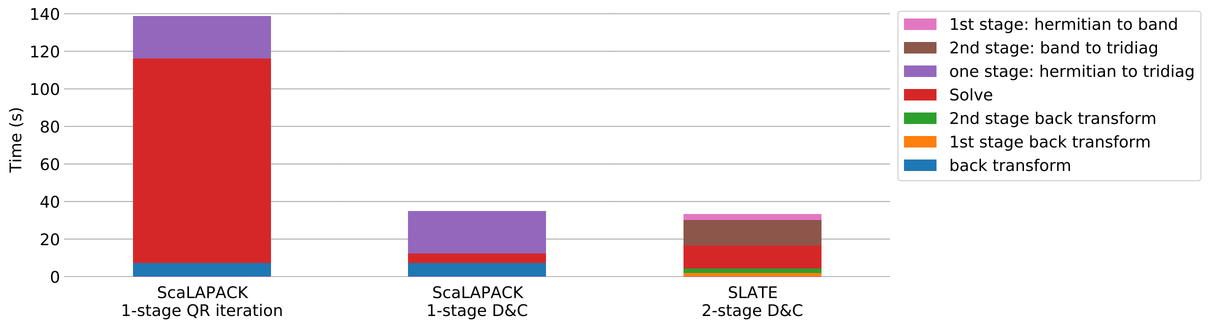


Figure 5.4: Profile of eigenvalue solver implementations showing each phase for N=12288. One node of Summit is used.

Bibliography

[1] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *ACM SIGARCH Computer Architecture News*, 14(2): 414–423, 1986.

[2] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[3] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[4] Magnus R Hestenes. Inversion of matrices by biorthogonalization and related results. *Journal of the Society for Industrial and Applied Mathematics*, 6(1):51–90, 1958.

[5] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.

[6] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. In *Linear Algebra*, pages 134–151. Springer, 1971.

[7] John GF Francis. The QR transformation: a unitary analogue to the LR transformation—part 1. *The Computer Journal*, 4(3):265–271, 1961.

[8] Vera N Kublanovskaya. On some algorithms for the solution of the complete eigenvalue problem. *USSR Computational Mathematics and Mathematical Physics*, 1(3):637–657, 1962.

[9] Gary W Howell, James W Demmel, Charles T Fulton, Sven Hammarling, and Karen Marmol. Cache efficient bidiagonalization using BLAS 2.5 operators. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):14, 2008. doi: 10.1145/1356052.1356055.

[10] Azzam Haidar, Jakub Kurzak, and Piotr Luszczek. An improved parallel singular value algorithm and its implementation for multicore hardware. In *Proceedings of the International*

Conference on High Performance Computing, Networking, Storage and Analysis (SC'13), page 90. ACM, 2013. doi: 10.1145/2503210.2503292.

[11] Azzam Haidar, Hatem Ltaief, and Jack Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 8:1–8:11. ACM, 2011. doi: 10.1145/2063384.2063394.

[12] Azzam Haidar, Stanimire Tomov, Jack Dongarra, Raffaele Solca, and Thomas Schulthess. A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine-grained memory aware tasks. *International Journal of High Performance Computing Applications*, 28(2):196–209, 2014. doi: 10.1177/1094342013502097.

[13] Robert Schreiber and Charles Van Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10 (1):53–57, 1989. doi: 10.1137/0910005.

[14] Jan JM Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik*, 36(2):177–195, 1980.

[15] Françoise Tisseur and Jack Dongarra. A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures. *SIAM Journal on Scientific Computing*, 20(6):2223–2236, 1999.

[16] Jack J Dongarra and Danny C Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM Journal on Scientific and Statistical Computing*, 8(2):s139–s154, 1987.

[17] Ren-Cang Li. Solving secular equations stably and efficiently. Technical Report LAPACK working note (LAWN) 89, University of California, Berkeley, April 1993.

[18] Jeffery D Rutter. A serial implementation of Cuppen's divide and conquer algorithm for the symmetric eigenvalue problem. Technical Report UCB/CSD 94/799, University of California, Berkeley, February 1994.

[19] Ming Gu and Stanley C Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM Journal on Matrix Analysis and Applications*, 16(1): 172–191, 1995.

[20] Kadir Akbudak, Paul Bagwell, Sebastien Cayrols, Mark Gates, Dalal Sukkari, Asim YarKhan, and Jack Dongarra. SLATE performance improvements: QR and eigenvalues, SWAN no. 17. Technical Report ICL-UT-21-02, Innovative Computing Laboratory, University of Tennessee, 4 2021. URL https://www.icl.utk.edu/publications/swan-017. revision 04-2021.